Lab 1. Multi-armed bandits

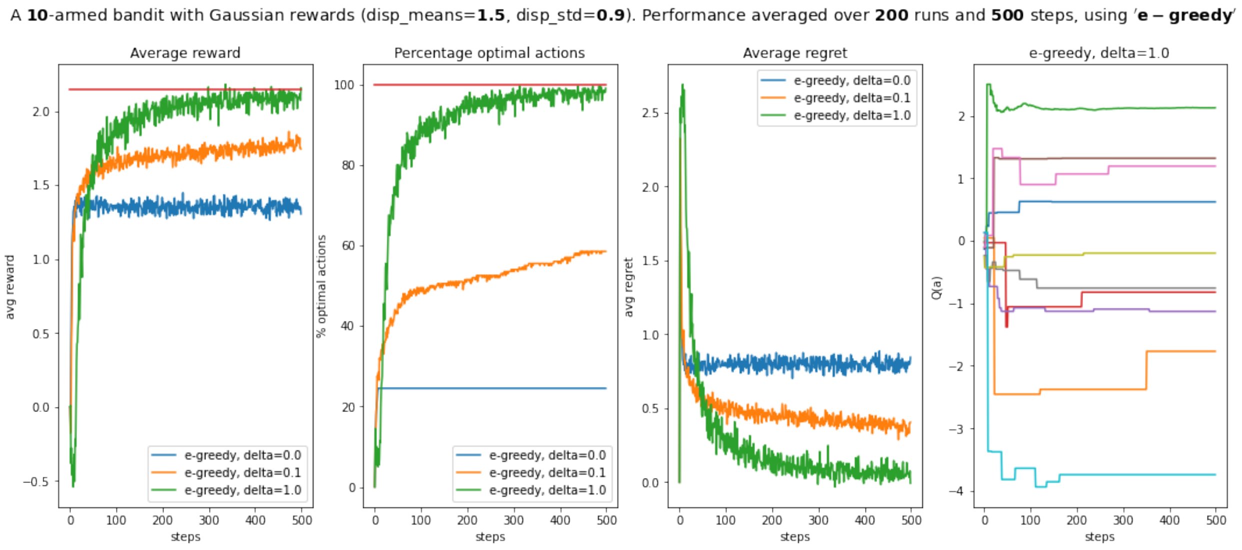
Alex Carrillo Alza
Aprenentatge per Reforç i Aprenentatge Profund, Ciència i Enginyeria de Dades (UPC)

October 6, 2020

1 ϵ-greedy algorithm

As opposed to the greedy action selection algorithm, the ϵ-greedy approach aims to avoid the agent getting stuck in a non-optimal action for ever. The above is achieved as every possible action will be sampled an infinite number of times, as t approaches infinity.

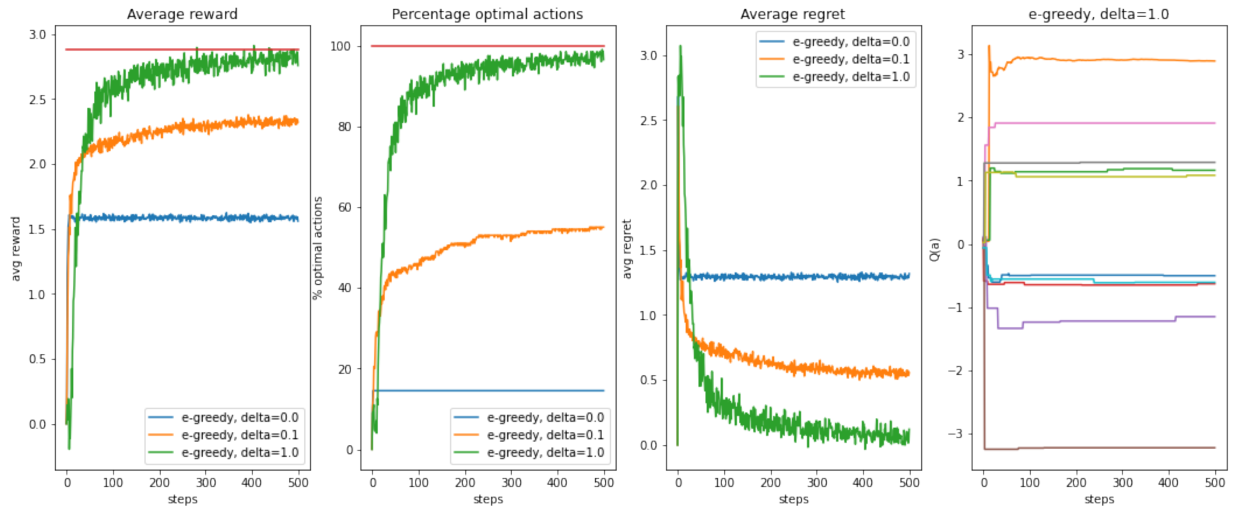
In order to better understand the behavior of it, we experiment with a 10-armed bandit with Gaussian rewards (and dispersion of means set to 1.5), which performance has been averaged over 200 runs and 500 steps. As requested, we focus on changing the values of the variance of the Gaussian, comparing values 0.9, 0.5, 0.05 and 0.001, on the dispersion on standard deviation:



The first thing we notice is that the overall performance seems quite good, since we almost finished with a $\sim 100\%$ percentage of optimal actions at the end of the iterations. Analogously, the regret seems to follow a decreasing tendency, which is good. Now, focusing on the three delta values tested, we observe that when $\delta = 0$ we are in a *purely greedy* case, where a_t is always $\text{argmax}_a \hat{Q}_t(a)$ and hence there is no exploration, so we can get stuck in a $a \neq a^*$ (exactly what happens in the plot).

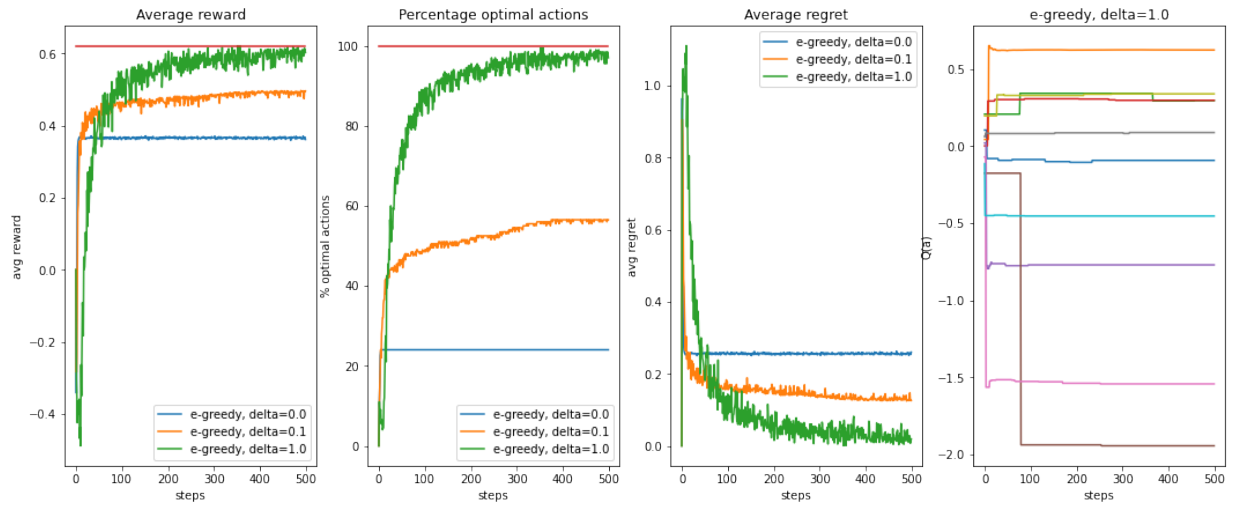
We observe that, as the variance is quite big, the estimate values of actions (shown in the fourth plot) appear very tangled, giving us the intuition that they are constantly updated (as they variate too much). For a `disp_std = 0.5` there is not so much difference, but we start to see a clearer estimate values of actions:

A **10**-armed bandit with Gaussian rewards (disp_means=**1.5**, disp_std=**0.5**). Performance averaged over **200** runs and **500** steps, using '$\mathbf{e-greedy}$'

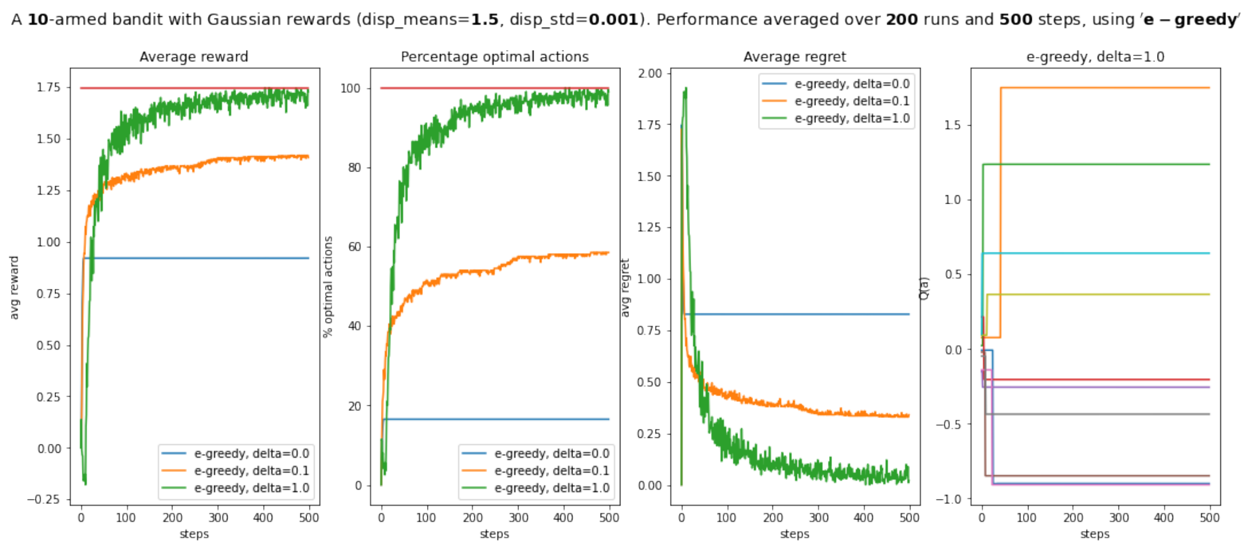


Following, with an smaller value on variance (*i.e.* `0.05`), we experiment a notable decrease of the regret (the scale has changed), which is a sign of better understanding or guess of the optimal arms. Also, the estimate values of actions become a bit clearer:

A **10**-armed bandit with Gaussian rewards (disp_means=**1.5**, disp_std=**0.05**). Performance averaged over **200** runs and **500** steps, using '$\mathbf{e-greedy}$'



Lastly, with a negligible value of variance (*i.e.* `0.001`), although the percentage of optimal actions chosen remains the same, the regret is reduced to almost half the value and the values of actions show a perfect choice of the optimal arm in the very first iterations:



The tendency we observe is that as the variance decreases, the arms become more and more deterministic somehow and therefore it is easier to find the optimum one. It is as if once we have tested each arm at least once we find which one is the best. The experiments also derive the conclusion that the performance of the algorithm depends on the similarity between the *pdf* of the optimal arm and the others. Hence, variance may lead to serious problems having similarly looking arms with different means.

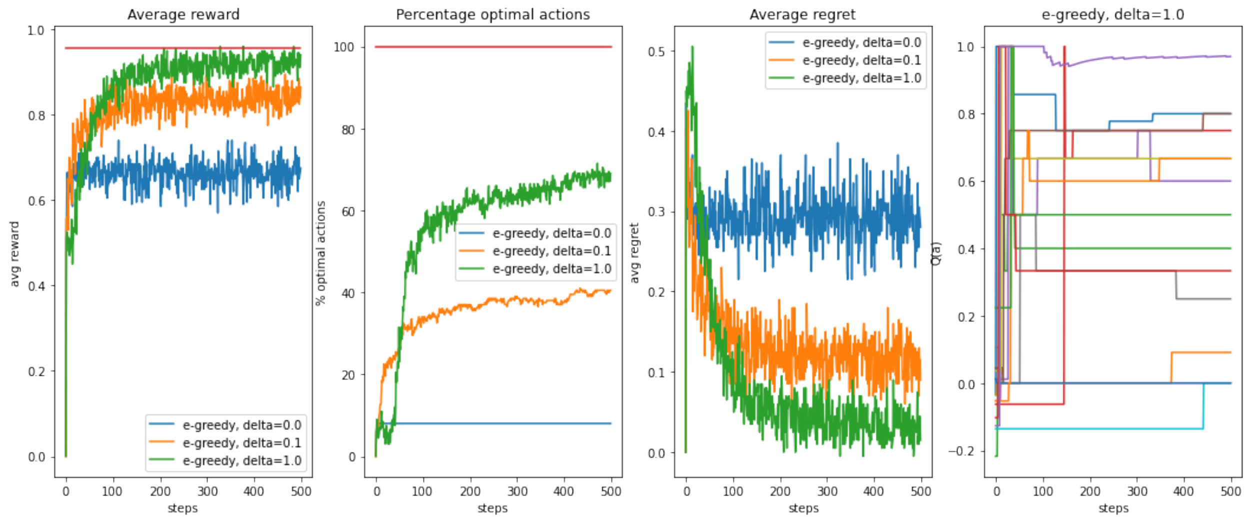
2 Application of an m-armed bandit

On the one hand, the first idea we came up with was the application of an armed bandit to finance, more specifically, in the task of portfolio management. It can be said that the prices of an acquired (or sold) share or action of a company, *i.e.* the *rewards* for each share, can be modeled with a Gaussian distribution. However, we notice that this scenario is indeed non-stationary, as the observed return on shares over time cannot be said to be an independent variable over time.

On the other hand, the Bernoulli distribution models the probability of a single event occurring with probability p, *i.e.* get heads on a single p-coin flip. This is the special case of the Binomial distribution where $N = 1$. In the armed bandit scenario, this distribution can be used to approximate a hit or miss event, such as if a user clicks on a headline, ad or recommended product. This application is indeed *stationary* as we can think that the decisions that lead a person to click a certain link can be said to be an independent variable in time.

2.1 Programming it

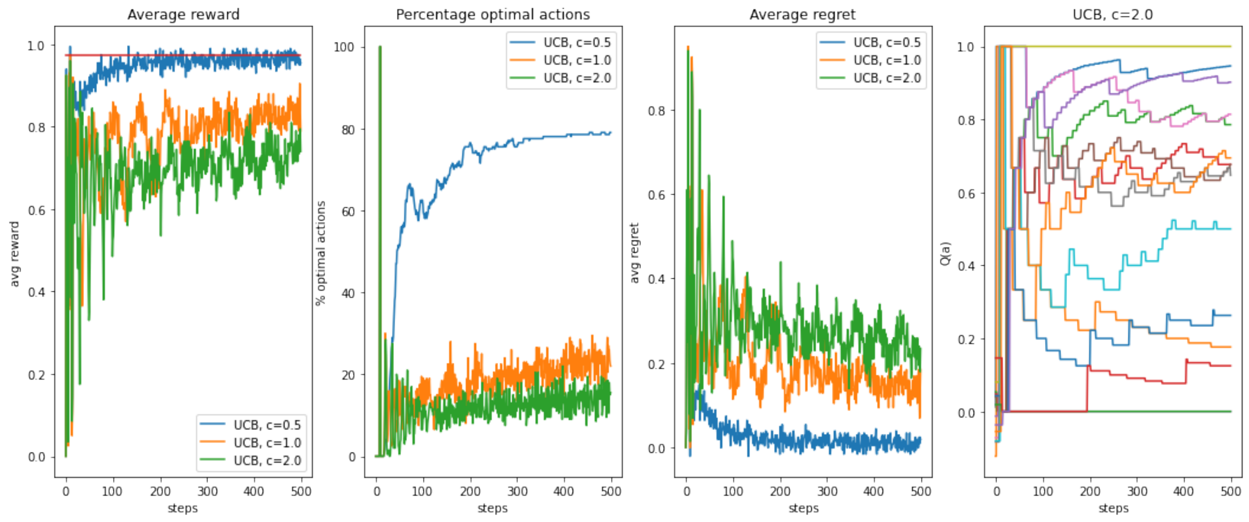
To exemplify this kind of application we adapted the code provided to simulate a Bernoulli bandit, that is, setting a probability to each arm that will generate a Bernoulli reward.



Roughly, the utility of this application (simulating 15 different ads to try or products) seems to be good overall, with success rate between 60-70% for selecting the best optimal actions (or ads). This is at least with a high value of delta (1), *i.e.* always exploring for new actions to make, which make sense in this application and taking into account the high number of arms involved.

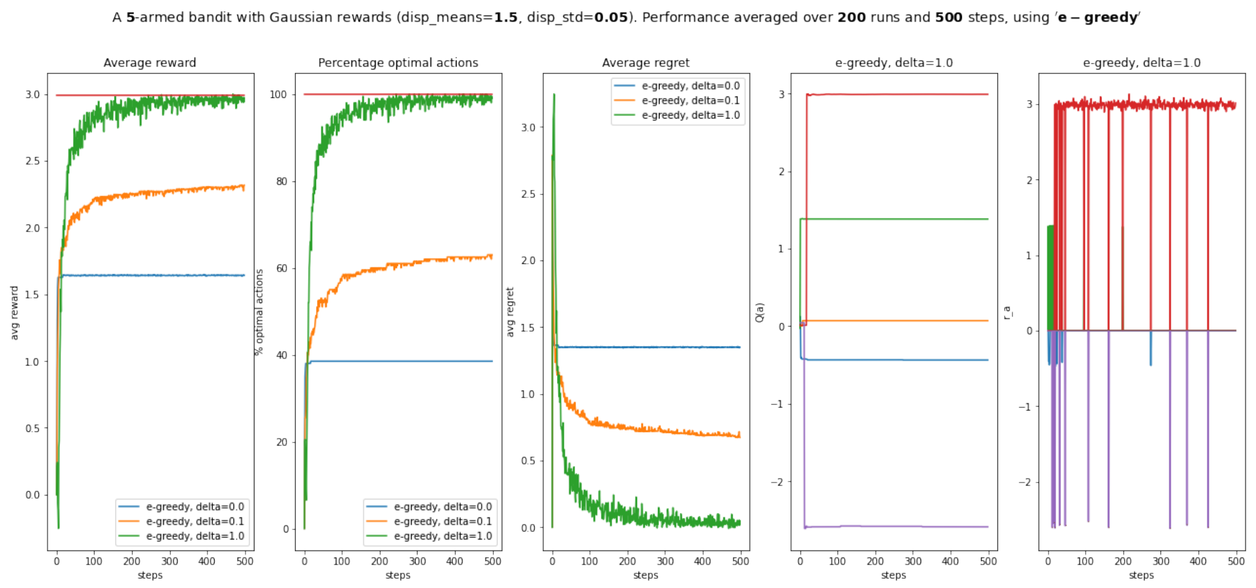
2.2 Check UCB performance

Now, we test the *upper-confidence-bound* (UCB) action selection strategy, which aims to have more potential for optimality, as its the lemma is "Optimism in the face of uncertainty" and it naturally balances exploration and explotation. We observe that percentage of optimal selected actions has increased to nearly a 80% and the regret has decreased, at least with the least exploratory strategy, *i.e.* $c = 0.5$:



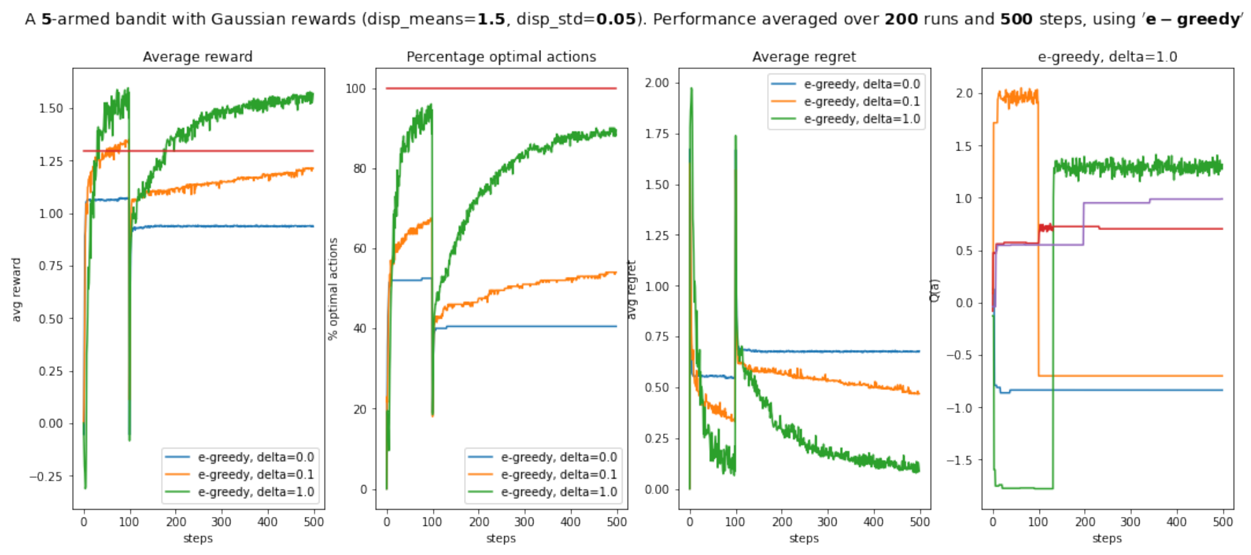
2.3 Rewards for every action

Next, we want to highlight the meaning of the fourth plot, *i.e.* the evolution of the estimated $Q(a)$ on a single run. [Notice we are back to a Gaussian bandit with only three arms, in order to clearly show what is is explained.] The fifth plot shows the actual rewards for every action on a single run, and it can be seen that each update of an arm returning a reward somehow matches the plot on its left, the steps of which estimation correspond to those rewards:

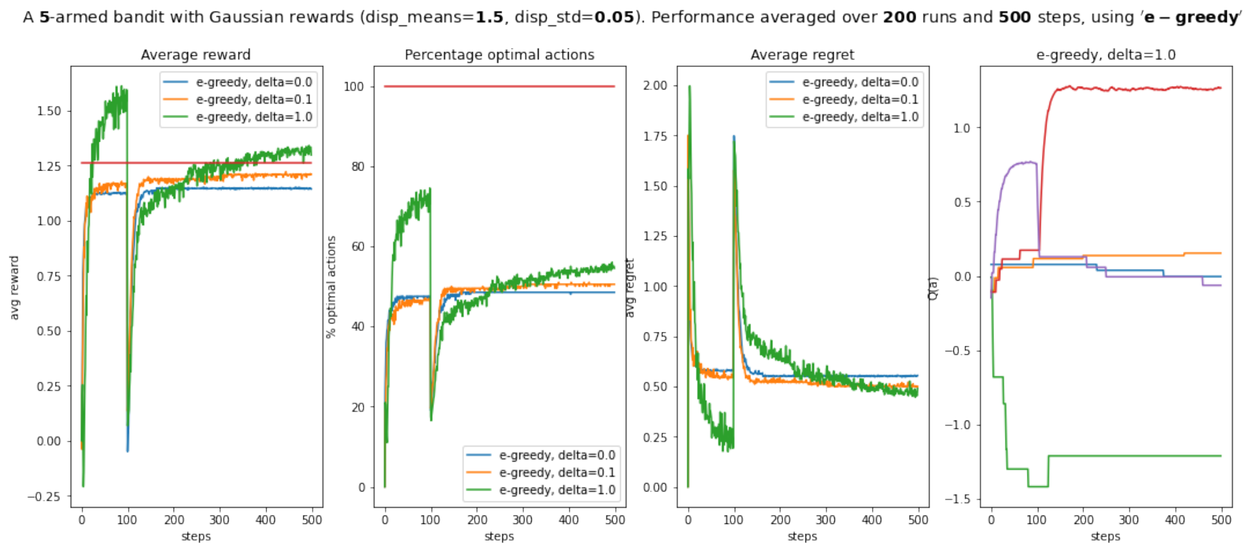


3 Non-stationary environment

Finally, we study a different scenario on which we make the environment *non-stationary* and we include the appropriate changes in the code to allow the algorithms track the best decisions.



In order to do so, we added a change in the means and probabilities, for our implementation of Gaussian and Bernoulli bandits respectively, in the middle of the race of iterations (in our case, once in the whole loop of steps). And, as requested, we change the function of estimating $Q_{t+a}(a)$ by using an alpha parameter on it: $\hat{Q}_{t+a}(a) = \hat{Q}_t(a) + \alpha(r_t - \hat{Q}_t(a))$. In the plot above, we tried a factor of $\alpha = 0.9$ and on the plot below we tried $\alpha = 0.1$:



As seen in theory class, an alpha near to 1 will provide more memory to the estimation, giving less weight to recent rewards than to long-past rewards. And a value of alpha near 0, will have indeed less memory. Comparing both plots, we can see that in the first one it is possible to counteract the change in time of the rewards and the relapse return to almost its original value (looking at the percentage of optimal actions for e-greedy with delta 1). In opposition, in the second plot it is difficult to choose the optimal action again.

Note we can even change the value of alpha over the iterations (concretely conditions are met for $\alpha_t = 1/t$) in order to ensure convergence with probability 1. However, the condition is not met if the step size is fixed. As a conclusion, we intuit that we need alpha to be constant so that the algorithm is reactive and it is able to adapt to the new changes of the non-stationary scenario.

4 Colab notebook and code

All code, parameters and generated plots used in this report can be seen at this Colab notebook.:

https://colab.research.google.com/drive/1rUwK-0fmCVqsckzJv8DLSLWpdcXvYhTk?usp=sharing

However, a glimpse on the scripts used is provided below, just in case.

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

np.random.seed(1234)

'''Insert title, and axis labels to plots'''
def insert_labels(labels, ax):
    if 'title' in labels:
        ax.set_title(labels['title'])
    if 'xlabel' in labels:
        ax.set_xlabel(labels['xlabel'])
    if 'ylabel' in labels:
            ax.set_ylabel(labels['ylabel'])

def generate_plot(r, ba, q, rewards, re, m, NRuns, NSteps, strg, par, bandit, meanA='', dispMeansA='', 
    str_legend = ['{}, {}={}'.format(strg, 'c' if strg == 'UCB' else 'delta', p) for p in par]
    #str_legend = ['{}, {}={}'.format('UCB' if strg else 'e-greedy', 'c' if strg else 'delta', p) for p

    conf_1 = {'title': 'Average reward', 'xlabel': 'steps', 'ylabel': 'avg reward'}
    conf_2 = {'title': 'Percentage optimal actions', 'xlabel': 'steps', 'ylabel': '% optimal actions'}
    conf_3 = {'title': 'Average regret', 'xlabel': 'steps', 'ylabel': 'avg regret'}
    conf_4 = {'title': str_legend[-1], 'xlabel': 'steps', 'ylabel': 'Q(a)'}
    conf_5 = {'title': str_legend[-1], 'xlabel': 'steps', 'ylabel': 'r_a'}

    if show_rewards:
      fig, ax = plt.subplots(1,5, figsize=(22, 9))
      for i, (data, conf) in enumerate(zip([r, ba, re, q, rewards],[conf_1, conf_2, conf_3, conf_4, con
          for j, d in enumerate(data):
              ax[i].plot(d)
              insert_labels(conf, ax[i])
          if i < 3:
              ax[i].legend(str_legend)
          if i ==4:
              ax[i].plot(d, marker=',', linestyle = 'None')
    else:
      fig, ax = plt.subplots(1,4, figsize=(18, 7))
      for i, (data, conf) in enumerate(zip([r, ba, re, q],[conf_1, conf_2, conf_3, conf_4])):
          for j, d in enumerate(data):
              ax[i].plot(d)
              insert_labels(conf, ax[i])
          if i < 3:
              ax[i].legend(str_legend)

    if bandit == 'gaussian':
      ax[0].plot(np.ones(len(d))*np.max(meanA)) # Include the optimum reward in the displa
      fig.suptitle("A " + r"$\bf{" + str(m) + "}$" + "-armed bandit with Gaussian rewards (disp_means="
                   "). Performance averaged over " + r"$\bf{" + str(NRuns) + "}$" + " runs and " + r"$\bf
                   fontsize=14)

    elif bandit == 'bernoulli':
       ax[0].plot(np.ones(len(d))*np.max(probs))

    if strg == 'e-greedy':
      ax[1].plot(np.ones(len(d))*100)



def run_bandits(m = 10, dispMeansA = 1.5, dispStd = 0.05,
                NRuns = 200, NSteps = 500, strategy = 'e-greedy', params = None, bandit = 'gaussian', s

  # "alpha":        time constant for incremental estimation of Q in time-varying environment
```

```
if bandit == 'gaussian':
  # "m":            number of actions
  # "dispMeansA":   dispersion in the values of means for every action
  # "dispStd":      dispersion in the values of variances for every action

  meanA = np.random.randn(m)*dispMeansA   # means for every action
  stdA = np.random.rand(m)*dispStd        # std deviations for every action
  bestAction = np.argmax(meanA)           # index of the best action
  optimum = np.max(meanA)
elif bandit == 'bernoulli':
  probs = np.random.random(m)
  bestAction = np.argmax(probs)           # index of the best action
  optimum = np.max(probs)


avg_r = []
avg_ba = []
avg_regret = []

# Set default params to test.
if params is None:
  if strategy == 'e-greedy':
    params = np.array([0, 0.1, 1])                              # test values of 'delta' in e-greedy
  elif strategy == 'UCB':
    params = np.array([0.5, 1, 2])                              # test values of 'c' in UCB

for p in range(len(params)):                                       # parameters for the method
  r = np.zeros((NRuns, NSteps))                                 # instantaneous rewards
  BA = np.zeros((NRuns, NSteps))                                # identifies if best action has been se
  regret = np.zeros((NRuns, NSteps))
  for i in range(NRuns):
    Q = np.zeros((m, NSteps))                                 # average reward per action
    Q[:,0] = np.random.randn(m)*0.1                           # initialization of Q
    actual_r = np.zeros((m, NSteps))
    ta = np.zeros((m))                                        # times each action is selected
    for j in range(1, NSteps):

      if not stationary:
        if j == (NRuns // 2):  # Change means or probs. in the middle of the race --> non-stationary
          if bandit == 'gaussian':
            meanA = np.random.randn(m)*dispMeansA   # means for every action
            stdA = np.random.rand(m)*dispStd        # std deviations for every action
            bestAction = np.argmax(meanA)           # index of the best action
            optimum = np.max(meanA)
          elif bandit == 'bernoulli':
            probs = np.random.random(m)
            bestAction = np.argmax(probs)           # index of the best action
            optimum = np.max(probs)

      # e-greedy
      if strategy == 'e-greedy':
        I = np.argmax(Q[:,j-1])                             # select best action
        if np.random.rand() > min(1, m*params[p]/j):           # e-greedy with decaying epsilon
```

```
            a = I
        else:
            randIndex = np.random.randint(m-1)          # select an action other than greedy one
            a = randIndex + (randIndex >= I)

      # UCB
      elif strategy == 'UCB':
        U = p * np.sqrt(2*np.log(j) / ta + 0.0001)

        if p == 0.5:
          print(U)
        a = np.argmax(Q[:,j-1] + U)

      ta[a] += 1
      if bandit == 'gaussian':
        r[i,j] = meanA[a] + np.random.randn()*stdA[a]   # obtain the gaussian reward
      elif bandit == 'bernoulli':
        r[i,j] = 1 if np.random.random() < probs[a] else 0

      actual_r[a,j] = r[i,j]

      Q[:,j] = Q[:, j-1]  # update Q function
      if stationary:
        Q[a,j] += 1/ta[a] * (r[i,j] - Q[a, j])
      else:
        if alpha is None:
          al = 1 / j
        else:
          al = alpha
        Q[a,j] += al * (r[i,j] - Q[a, j])

      BA[i,j] += bestAction == a
      regret[i,j] = optimum - r[i,j]

  avg_r.append(np.mean(r.copy(), axis=0))
  avg_ba.append(np.mean(BA.copy(), axis=0)*100)
  avg_regret.append(np.mean(regret.copy(), axis=0))

if bandit == 'gaussian':
  generate_plot(r=avg_r, ba=avg_ba, q=Q, rewards=actual_r, re=avg_regret, m=m, NRuns=NRuns, NSteps=NS
elif bandit == 'bernoulli':
  generate_plot(r=avg_r, ba=avg_ba, q=Q, rewards=actual_r, re=avg_regret, m=m, NRuns=NRuns, NSteps=NS
```