

Simulation Notebook

In the real world, we don't know the value of each action that we are going to take

This is also the reason why we can't use Kelly's criteria in real life because we don't priorly know the winning or losing probability

This is where Reinforcement learning comes to the rescue. We don't know the correct action for every state beforehand. Here, we receive rewards based on a series of actions. It is only after running the iteration or cycle multiple times. That you are able to get some optimal results.

- The idea of K-armed bandit problem is to maximise reward
- But to achieve this goal we need to take best actions
- How to find the best actions?
 - We need to explore or exploit.

The need to balance exploration and exploitation is a distinctive challenge that arises in reinforcement learning.



PseudoCode of the simulation

```
1 fix the value of epsilon
2 initialise allMachineQValueMatrix matrix to be 0
3
4 for range in runs:
5
6     :-create an empty array (counterArray) to keep track the number of times an action was taken.
7
8     for trials in trial:
9
10        Step 1: generating a random number using random number generator between 0 and 1
11
12        Step 2: use that value for Action selection
13            if random Number is less than or equal to 1 - epsilon
14                then
15                    (exploit)
16                    :select the action which has got the max expected value
17            else
18                (explore)
19                    :choose any action with equal probability
20
21            :-store the action that you took above
22            :-increment the counter Array for that particular action
23
24        Step 3: reward Allocation
25            :-store the reward for the particular action that was taken
26
27        Step 4: Action Value Updation:
28            :- Update the allMachineQValueMatrix values for the next trial using the below incremental implementation
29            technique
30                NewEstimate <- OldEstimate + stepSize[Target - Old Estimate]
31
32 Step 5: plot the allMachineQValueMatrix by taking mean column wise for each bandit machine
```

Our very first reinforcement learning Code



```
In [204]: 1 "Discrete reward example"
2
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import random
6
7
8 def epsilonGreedyMethod(e,    numberofBanditArms      ,   runs   ,   trials):
9
10    allMachineQValueMatrix = np.zeros((numberofBanditArms,runs,trials))
11
12    for run in range(runs):
13
14        #starting a fresh game
15        actionSelectedCounter = np.zeros(numberofBanditArms)
16
17        for trial in range(trials):
18
19            #The np.random.uniform() function generates a random float number between a and b, including
20            #a but excluding b.
21            rand_num = np.random.uniform(0, 1)
22
23            ...
24            Step 1: Action selection:
25            e = epsilon = e.g. 0.1
26            probability of exploration = e = e.g. 0.1
27            probability of exploitation = 1 - e = e.g. 0.9
28
29            R is the reward
30            A is the action
31            ...
32
33
34            if rand_num <= 1 - e:
35                ...
36                exploit
37                ...
38
39
40
41
42            #storing the expected Q value for each arm for the current trial
43            QvalueOfBanditArms = np.zeros(numberofBanditArms)
44
45            for arm in range(numberofBanditArms):
46                QvalueOfBanditArms[arm] = allMachineQValueMatrix[arm][run][trial]
```



#0 #1 #2 Trials
 B_0 Run1 $\leftarrow [0, 0, 0]$,
Run2 $\leftarrow [0, 0, 0]$,
 B_1 Run1 $\leftarrow [0, 0, 0]$,
Run2 $\leftarrow [0, 0, 0]$

[0, 0, 0]

#finding the max expected value of an action (Qvalue) in the QvalueArrayForAllBanditArms

maxExpectedValue = max(QvalueOfBanditArms)

$$A_t \doteq \arg \max_a Q_t(a)$$

You will solely focus
on increasing your reward

#finding which actions are giving me this maxExpectedValue
shortListedActions = [action for action, estimateValue in enumerate(QvalueOfBanditArms)
if estimateValue == maxExpectedValue]

[0, 1]

[0, 0, 5]

#breaking the tie randomly if there are more than one selected actions

actionLength = len(shortListedActions)

)

```

99
100
101
102     if actionLength > 1:
103
104         #choosing an action randomly from the shortListedActions
105         finalAction = random.choice(shortListedActions)
106
107     else:
108         finalAction = shortListedActions[0]
109
110     #incrementing the counter Array for that particular action
111     actionSelectedCounter[finalAction] += 1
112
113     [1, 0, 0]
114
115
116
117
118     A = finalAction [0]
119
120
121     else:
122         ...
123         explore
124         ...
125         in this block you will try to discover
126         unexplored states.
127
128
129     #selecting randomly from among all the actions with equal probability.
130     randActionIndex = np.random.randint(0,numberOfBanditArms)
131
132     #incrementing the counter Array for that particular action
133     actionSelectedCounter[randActionIndex] += 1
134
135     A = randActionIndex
136
137
138     Step 2: Reward Allocation
139
140
141     rand_num = np.random.uniform(0, 1)
142     if A == 0:
143         R = 2000
144
145
146
147
148
149
150     else:
151         if rand_num <= 0.6:
152             R = 5000
153         else:
154             R = 0
155
156
157
158
159
160
161
162
163     Step 3:
164     Action Value Updation:
165     NewEstimate <- OldEstimate + stepSize[Target - Old Estimate]
166
167     Matrix Updation For Performance Measurement
168     A is actual arm lever that you pulled down
169
170     #if condition to prevent index out of bound error.
171     if trial+1 != trials:
172         for arm in range(numberOfBanditArms):
173
174             if arm == A:
175                 Rn - Qn
176                 #updation of the expected value for Q for the arm that we pulled down
177                 update = (R - allMachineQValueMatrix[arm][run][trial])/actionSelectedCounter[A]
178                 allMachineQValueMatrix[arm][run][trial+1] = allMachineQValueMatrix[arm][run][trial]+update
179
180             else:
181                 #updation of the expected Q value for the other arms that we left in this trial
182                 if trial != 0:
183                     #updating the Q value of the arm of the next trial with the previous trial's value
184                     #if this arm was not pulled down
185                     allMachineQValueMatrix[arm][run][trial + 1] = allMachineQValueMatrix[arm][run][trial]
186
187
188
189 return allMachineQValueMatrix

```

$Q_{n+1} = Q_n + \frac{1}{n} [R_n - Q_n]$



Setup

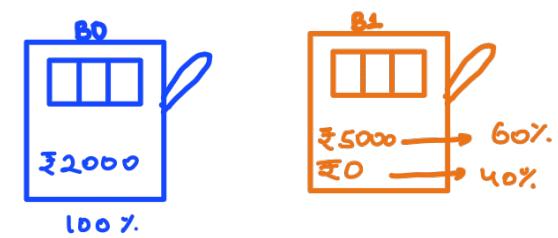
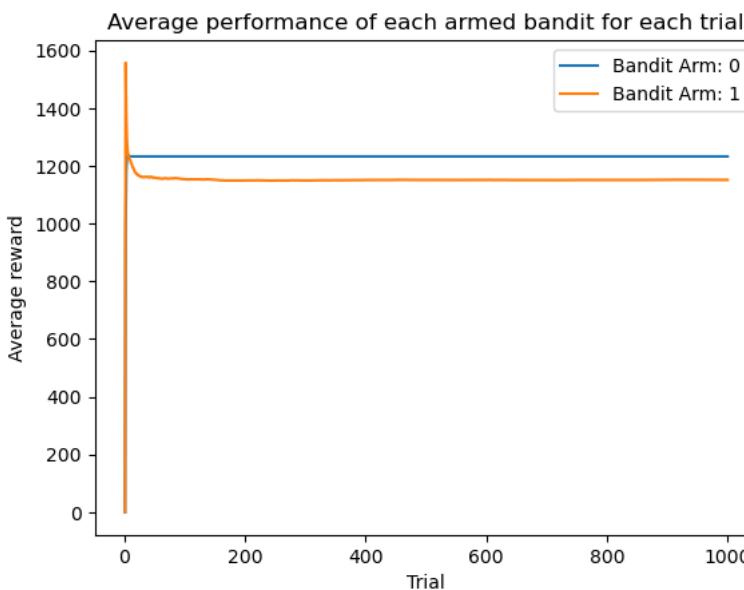
```
In [205]: 1 runs = 2000
2 trials = 1000
3
4 #1 run = 1000 trials
5 # we are running this for 2000 * 1000 trials!!!!
```

Discrete reward example

running discrete reward example with epsilon = 0

```
In [206]: 1 numberOfBanditArms = 2
2 allMachineQValueMatrix = epsilonGreedyMethod(e = 0, numberOfBanditArms = numberOfBanditArms
3 , runs = runs, trials = trials)
4
5 #print(allMachineQValueMatrix)
6
7 column_wise_mean = []
8 for banditArmIndex in range(numberOfBanditArms):
9     column_wise_mean.append(np.mean(allMachineQValueMatrix[banditArmIndex], axis=0))
10
11 timeStepArray = np.arange(1,trials + 1)
12
13
14 for i in range(numberOfBanditArms):
15     plt.plot(timeStepArray, column_wise_mean[i], label = "Bandit Arm: " + str(i))
16
17 plt.legend(loc = "best")
18 plt.title("Average performance of each armed bandit for each trial")
19 plt.xlabel("Trial")
20 plt.ylabel("Average reward")
21
```

Out[206]: Text(0, 0.5, 'Average reward')



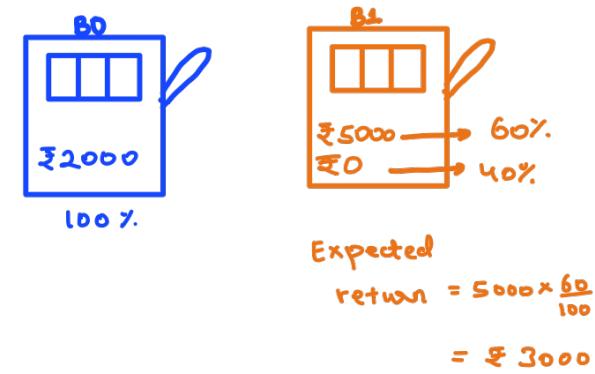
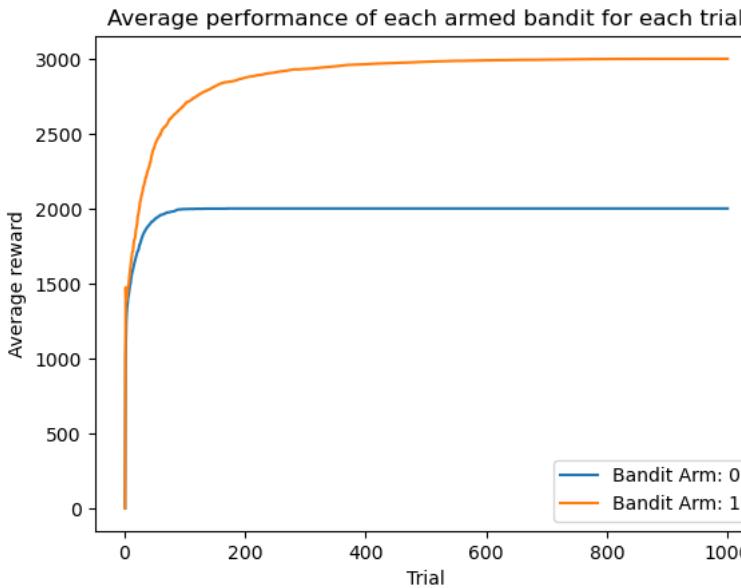
running discrete reward example with epsilon = 0.1

```

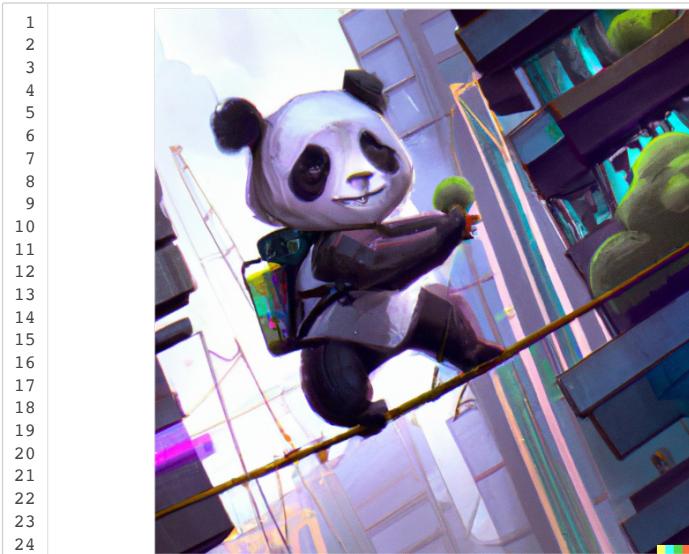
In [207]: 1
          2
          3 allMachineQValueMatrix = epsilonGreedyMethod(e = 0.1, numberOfBanditArms = numberOfBanditArms
          4 , runs = runs, trials = trials)
          5
          6 #print(allMachineQValueMatrix)
          7
          8 column_wise_mean = []
          9 for banditArmIndex in range(numberOfBanditArms):
          10     column_wise_mean.append(np.mean(allMachineQValueMatrix[banditArmIndex], axis=0))
          11
          12 timeStepArray = np.arange(1,trials + 1)
          13
          14
          15 for i in range(numberOfBanditArms):
          16     plt.plot(timeStepArray, column_wise_mean[i], label = "Bandit Arm: " + str(i))
          17
          18 plt.legend(loc = "best")
          19 plt.title("Average performance of each armed bandit for each trial")
          20 plt.xlabel("Trial")
          21 plt.ylabel("Average reward")
          22

```

Out[207]: Text(0, 0.5, 'Average reward')



Lets Level up the game



Continuous Probability Reward Distribution

In []:

1
2

```

In [215]: 1 """
2 Prerequisite:
3
4 Know the terms:
5 (1) Action = A (pulling the arm of a bandit machine)
6 (2) Reward = Rt (actual reward selected at time t)
7 (3) Value:
8   (a) True Value = q*(a) of an action (it is the actual value)
9   (b) Expected Value = Q*(a) of an action (it is a value based on stochastic method)
10 """
11
12 """
13 Step 1: For each bandit problem, the true action value q*(a), a = 1,...,10 were selected according to
14      a normal distribution with mean 0 and variance 1.
15 (in book the value of k is 10. This times number of actions is 10. This means that we need 10 data points.)
16 """
17 #setting up parameters
18 runs = 2000
19 trials = 500
20 numberofBanditArms = 3
21
22 #setting the true values of each Bandit Arm that is q*(a)
23 qValuesOfBanditArms = np.random.normal(5, 1, numberofBanditArms )
24 qValuesOfBanditArms

```

Out[215]: array([5.65257546, 3.9338925 , 4.27232989])

Visualising the Continuous Reward Distribution

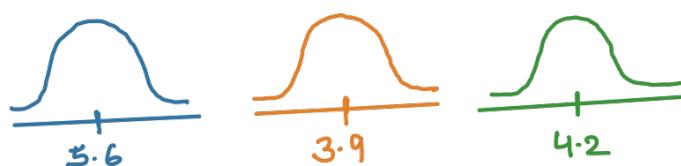
```

In [216]: 1 dataPoint = []
2 arms = len(qValuesOfBanditArms)
3 for x in range(arms):
4     dataPoint.append(np.random.normal(qValuesOfBanditArms[x], 1, 10000))
5
6 plt.title("K-armed Bandit Problem with k = " + str(arms)+ "")
7 plt.grid(True)
8 sns.violinplot(data=dataPoint, split=True, orient='v', width=0.8, scale='count')
9 plt.xlabel("Actions")
10 plt.ylabel("Reward Distribution")
11 plt.show()
12
13

```



Code	Rewards	Rewards	Rewards
6	4	4.7	
8	5	3	
4	6	2	
5	4.9	8	



In [217]:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import random
4
5
6
7 def epsilonGreedyMethod(e, numberOfBanditArms, qValuesOfBanditArms, runs, trials):
8
9     allMachineQValueMatrix = np.zeros((numberOfBanditArms, runs, trials))
10
11    for run in range(runs):
12
13        #starting a fresh game
14        actionSelectedCounter = np.zeros(numberOfBanditArms)
15
16        for trial in range(trials):
17
18            rand_num = np.random.uniform(0, 1)
19
20            ...
21
22            Step 1: Action selection:
23            e = epsilon
24            probability of exploration = e
25            probability of exploitation = 1 - e
26
27            R is the reward
28            A is the action
29            ...
30
31
32            if rand_num <= 1 - e:
33                ...
34                exploit
35                ...
36
37
38            #storing the expected Q value for each arm for the current trial
39            QvalueOfBanditArms = np.zeros(numberOfBanditArms)
40            for arm in range(numberOfBanditArms):
41                QvalueOfBanditArms[arm] = allMachineQValueMatrix[arm][run][trial]
42
43
44            #finding the max expected value of an action (Qvalue) in the QvalueArrayForAllBanditArms
45            maxExpectedValue = max(QvalueOfBanditArms)
46
47
48            #finding which actions are giving me this maxExpectedValue
49            shortListedActions = [action for action, estimateValue in enumerate(QvalueOfBanditArms)
50                                  if estimateValue == maxExpectedValue]
51
52
53            #breaking the tie randomly if there are more than one selected actions
54            actionLength = len(shortListedActions)
55            if actionLength > 1:
56
57                #choosing an action randomly from the shortListedActions
58                finalAction = random.choice(shortListedActions)
59
60            else:
61                finalAction = shortListedActions[0]
62
63            #print("the agent pulled the arm of bandit machine number ", finalAction)
64            actionSelectedCounter[finalAction] += 1
65
66            A = finalAction
67
68            reward added for action
69            ...
70
71    else:
72        ...
73        explore
74        ...
75
76        #selecting randomly from among all the actions with equal probability.
77        randActionIndex = np.random.randint(0, numberOfBanditArms)
78        #rewardofRandAction = rewardListForEachBanditArmsAction[randActionIndex]
79        actionSelectedCounter[randActionIndex] += 1
80        #print("actionSelectedCounter: ", actionSelectedCounter)
81
82        A = randActionIndex
83
84
85    ...
86
87    Step 2: Reward Allocation
88    When learning method applied to that k bandit problem selected action A at time step t,
89    the actual reward R, is being selected from a normal distribution with q(A at time t) and variance 1
90    ...
91    R = np.random.normal(qValuesOfBanditArms[A], 1, 1)
92    ...
93
94
95
96
97
98

```

mean *variance*

[5.6, 3.9, 4.2] *no. of data points*

```

99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138

```

Step 3:
Action Value Updation:
NewEstimate <- OldEstimate + stepSize[Target - Old Estimate]
...
Matrix Updation For Performance Measurement
A is actual arm lever that you pulled down
...
#if condition to prevent index out of bound error.
if trial+1 != trials:
 for arm **in** range(numberOfBanditArms):
 if arm == A :
 #updation of the expected value for Q for the arm that we pulled down
 update = (R - allMachineQValueMatrix[arm][run][trial])/actionSelectedCounter[A]
 allMachineQValueMatrix[arm][run][trial+1]=allMachineQValueMatrix[arm][run][trial] + update
 else:
 #updation of the expected Q value for the other arms that we left in this trial
 if trial != 0:
 #basically we are updating the Q value of the arm of the previous trial
 #if this arm was not pulled down
 allMachineQValueMatrix[arm][run][trial + 1] = allMachineQValueMatrix[arm][run][trial]
 return allMachineQValueMatrix

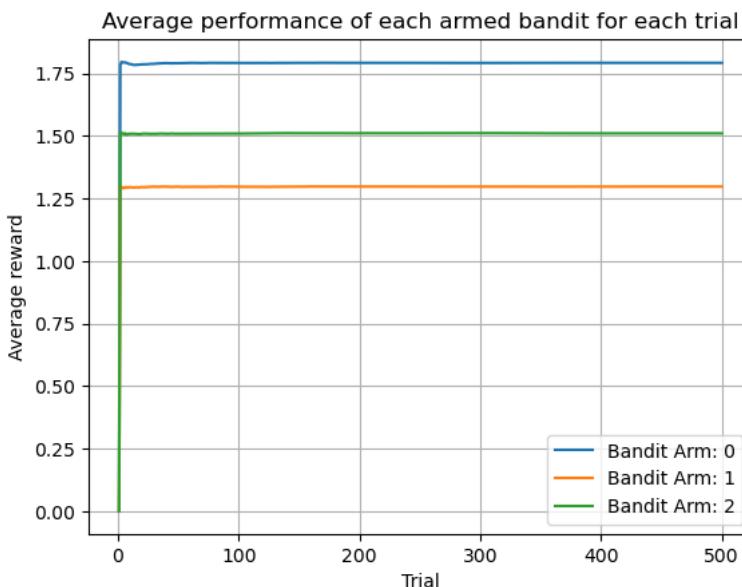
running continuous probability reward example with epsilon = 0

In [218]:

```

1  allMachineQValueMatrix = epsilonGreedyMethod(e = 0, numberOfBanditArms = numberOfBanditArms
2                                     ,qValuesOfBanditArms=qValuesOfBanditArms,runs = runs,trials = trials)
3  #print(allMachineQValueMatrix)
4
5
6
7  column_wise_mean = []
8  for banditArmIndex in range(numberOfBanditArms):
9      column_wise_mean.append(np.mean(allMachineQValueMatrix[banditArmIndex], axis=0))
10
11 timeStepArray = np.arange(1,trials + 1)
12 #print(column_wise_mean)
13 timeStepArray
14
15 for i in range(numberOfBanditArms):
16     plt.plot(timeStepArray, column_wise_mean[i], label = "Bandit Arm: " + str(i))
17
18 plt.legend(loc = "best")
19 plt.title("Average performance of each armed bandit for each trial")
20 plt.xlabel("Trial")
21 plt.ylabel("Average reward")
22 plt.grid(True)
23 plt.show()
24

```



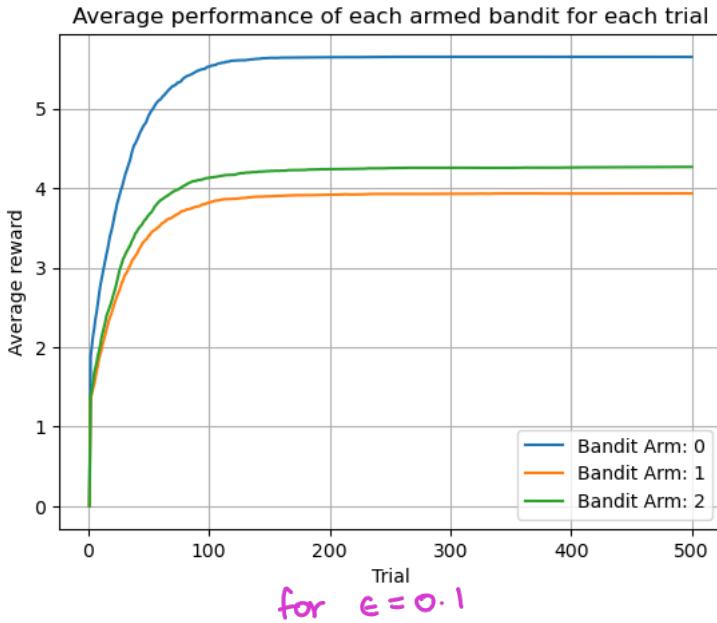
[5.6, 3.9, 4.2]

running continuous probability reward example with epsilon = 0.1

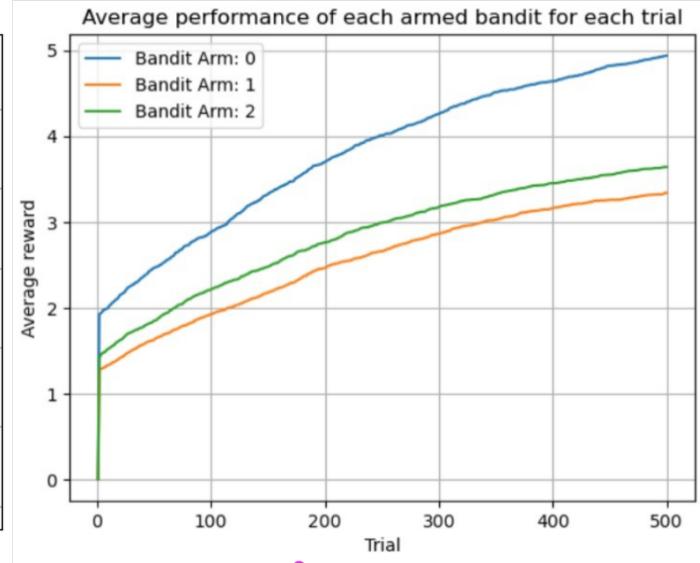
In [219]:

```
1 allMachineQValueMatrix = epsilonGreedyMethod(e = 0.1, numberofBanditArms = numberofBanditArms
2                                     ,qValuesofBanditArms = qValuesofBanditArms,runs=runs,trials = trials)
3 #print(allMachineQValueMatrix)
4 column_wise_mean = []
5 for banditArmIndex in range(numberofBanditArms):
6     column_wise_mean.append(np.mean(allMachineQValueMatrix[banditArmIndex], axis=0))
7
8 timeStepArray = np.arange(1,trials + 1)
9
10
11
12
13 for i in range(numberofBanditArms):
14     plt.plot(timeStepArray, column_wise_mean[i],label = "Bandit Arm: " + str(i))
15
16 plt.legend(loc = "best")
17 plt.title("Average performance of each armed bandit for each trial")
18 plt.xlabel("Trial")
19 plt.ylabel("Average reward")
20 plt.grid(True)
21 plt.show()
```

true expected values = [5.6, 3.9, 4.2]



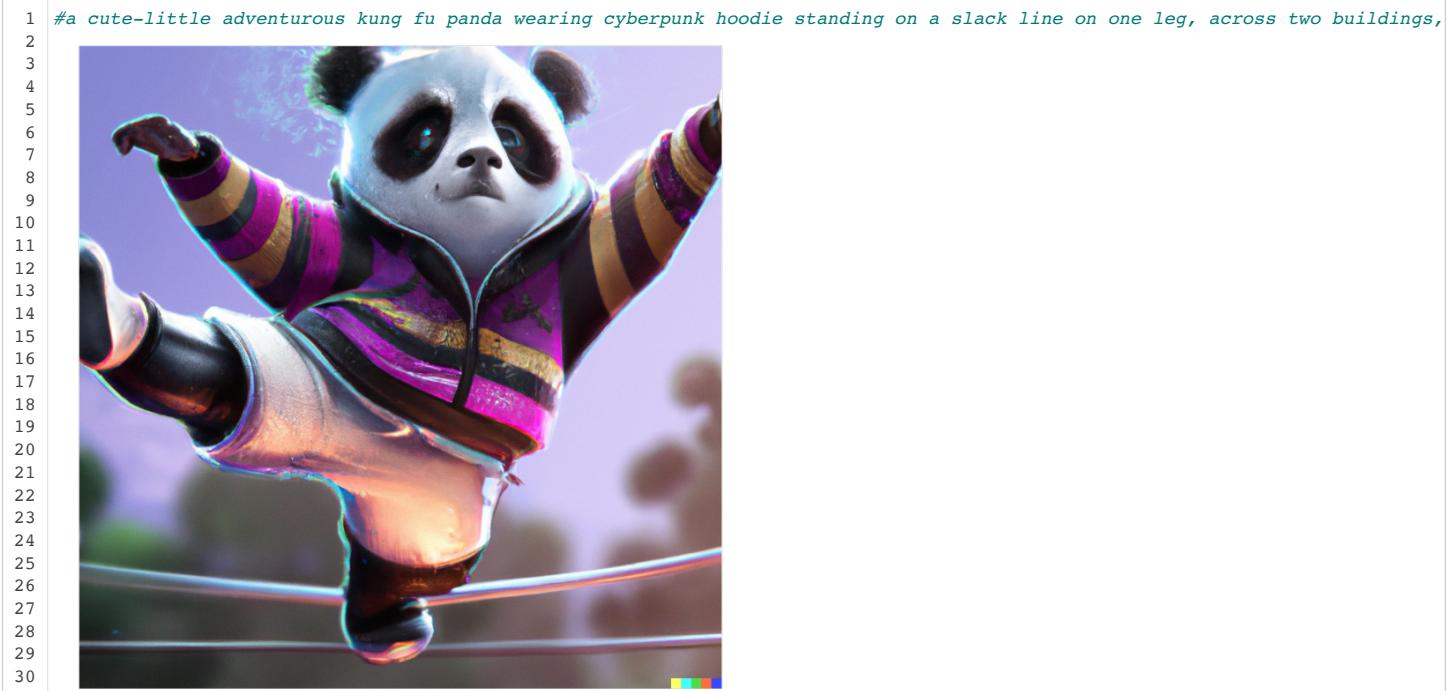
for $\epsilon = 0.1$



for $\epsilon = 0.01$

Let us now make things more interesting

In [220]:



Code

In [221]:

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import random
4
5
6
7 def epsilonGreedyMethodMem(e, numberofBanditArms, qValuesofBanditArms, runs, trials):
8
9     allMachineQValueMatrix = np.zeros((numberofBanditArms, runs, trials))
10
11    for run in range(runs):
12
13        #starting a fresh game
14        actionSelectedCounter = np.zeros(numberofBanditArms)
15
16        for trial in range(trials):
17
18            rand_num = np.random.uniform(0, 1)
19
20            ...
21
22            Step 1: Action selection:
23            e = epsilon
24            probability of exploration = e
25            probability of exploitation = 1 - e
26
27            R is the reward
28            A is the action
29            ...
30
31
32            if rand_num <= 1 - e:
33                ...
34                exploit
35                ...
36
37
38            #storing the expected Q value for each arm for the current trial
39            QvalueOfBanditArms = np.zeros(numberofBanditArms)
40
41            for arm in range(numberofBanditArms):
42                QvalueOfBanditArms[arm] = allMachineQValueMatrix[arm][run][trial]
43
44            #finding the max expected value of an action (Qvalue) in the QvalueArrayForAllBanditArms
45            maxExpectedValue = max(QvalueOfBanditArms)
46
47            #finding which actions are giving me this maxExpectedValue
48            shortListedActions = [action for action, estimateValue in enumerate(QvalueOfBanditArms)
49                                  if estimateValue == maxExpectedValue]
50
51            #breaking the tie randomly if there are more than one selected actions
52            actionLength = len(shortListedActions)
53            if actionLength > 1:
54
55                #choosing an action randomly from the shortListedActions
56                finalAction = random.choice(shortListedActions)
57
58            else:
59                finalAction = shortListedActions[0]
60
61            #print("the agent pulled the arm of bandit machine number ", finalAction)
62            actionSelectedCounter[finalAction] += 1
63
64            A = finalAction
65            ...
66            reward added for action
67            ...
68
69        else:
70            ...
71            explore
72            ...
73
74        #selecting randomly from among all the actions with equal probability.
75        randActionIndex = np.random.randint(0, numberofBanditArms)
76        #rewardOfRandAction = rewardListForEachBanditArmsAction[randActionIndex]
77        actionSelectedCounter[randActionIndex] += 1
78        #print("actionSelectedCounter: ", actionSelectedCounter)
79
80        A = randActionIndex
81
82
83        ...
84        Step 2: Reward Allocation
85        When learning method applied to that k bandit problem selected action A at time step t,
86        the actual reward R, is being selected from a normal distribution with q(A at time t) and variance 1
87        ...
88
89        R = np.random.normal(qValuesofBanditArms[A], 1, 1)
90
91
92        ...
93        Step 3:
94        Action Value Updation:
95        NewEstimate <- OldEstimate + stepSize[Target - Old Estimate]
96        ...
97
98        Matrix Updation For Performance Measurement

```

```

99 A is actual arm lever that you pulled down
100
101 #if condition to prevent index out of bound error.
102 if trial+1 != trials:
103     for arm in range(numberOfBanditArms):
104         if arm == A :
105             #updation of the expected value for Q for the arm that we pulled down
106             update = (R - allMachineQValueMatrix[arm][run][trial])/actionSelectedCounter[A]
107             allMachineQValueMatrix[arm][run][trial+1] = allMachineQValueMatrix[arm][run][trial]+update
108         else:
109             #updation of the expected Q value for the other arms that we left in this trial
110             #basically we are updating the Q value of the arm of the previous trial if this arm was not pulled down
111             allMachineQValueMatrix[arm][run][trial + 1] = allMachineQValueMatrix[arm][run][trial]
112
113     else:
114         if run + 1 != runs:
115             for arm in range(numberOfBanditArms):
116                 allMachineQValueMatrix[arm][run + 1][0] = allMachineQValueMatrix[arm][run][trial]

```

Magic Lines

Run 1

	B ₁	B ₂	B ₃
#1	0	0	1
#2	0	0	-0.8
#3	0	3	-0.8

Run 2

	B ₁	B ₂	B ₃
#1	0	5	-0.8
#2	0	-0.1	-0.8
#3	2	-0.1	-0.8

memory of Run 1

1 way

2nd way

Will the memory way make the graph converge faster?

```

140 return allMachineQValueMatrix
141
142
143

```

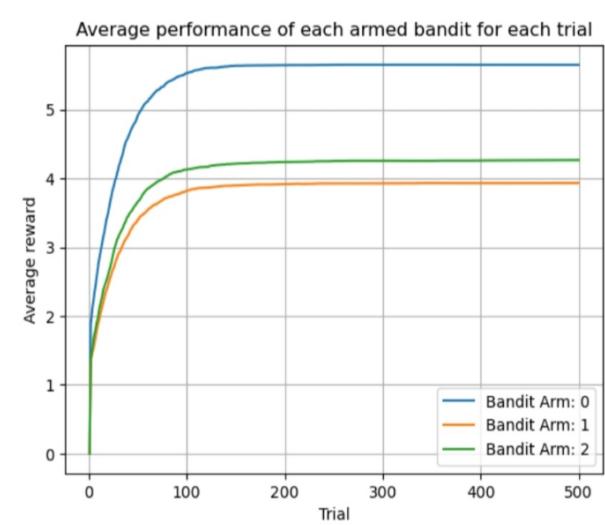
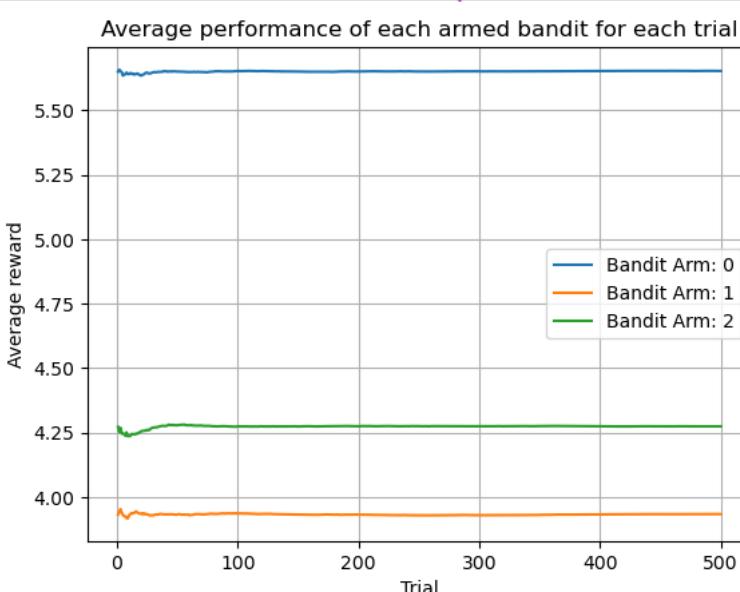
```

In [222]: 1 runs = 2000
2 trials = 500
3 allMachineQValueMatrix = epsilonGreedyMethodMem(e = 0.9, numberOfBanditArms = numberOfBanditArms
4 ,qValuesOfBanditArms=qValuesOfBanditArms,runs=runs,trials=trials)
5 #print(allMachineQValueMatrix)
6 column_wise_mean = []
7 for banditArmIndex in range(numberOfBanditArms):
8     column_wise_mean.append(np.mean(allMachineQValueMatrix[banditArmIndex], axis=0))
10
11 timeStepArray = np.arange(1,trials + 1)
12
13
14
15 for i in range(numberOfBanditArms):
16     plt.plot(timeStepArray, column_wise_mean[i],label = "Bandit Arm: " + str(i))
17
18 plt.legend(loc = "best")
19 plt.title("Average performance of each armed bandit for each trial")
20 plt.xlabel("Trial")
21 plt.ylabel("Average reward")
22 plt.grid(True)
23 plt.show()

```

with

without



In []:

1

image credits: DALLE-2