Question 2 Report

Contributors: Alexander Khrulev 500882732

Mahan Pandey 500881861

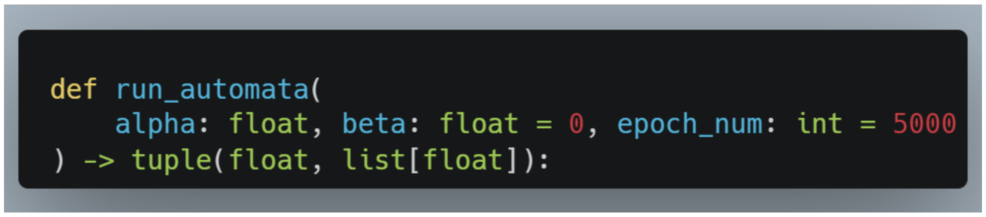
A Brief Algorithm Description

The following steps were used to run the experiment:

1. The arrays N_t, Q_t were initialized to 0 initially. They contain the number of times an action a was taken(at N_t[a]) and the average reward for an action (at Q_t[a])
2. The array of probabilities q_t(denoted as probabilities in code) was randomly generated from a Continuous Uniform distribution (0,1)
3. The optimal action was determined as the one which has the highest probability of success, i.e optimal_action=argmax(q)
4. The array of probabilities p_t was initiated to 0.1
5. For every iteration, we would get an action based on the corresponding probabilities in p_t. Afterwards, we can learn what signal will be generated by the environment based on the corresponding probability q_t[action] stored in the probabilities array.
6. Finally, we would adjust the probabilities based on the last signal. If the action was successful, its corresponding p_t value would be increased, while the rest would be decreased. Otherwise, the opposite would happen.
7. Ideally, we would hope that the p_t[optimal_action] would converge to or be very close to 1, indicating that the automata successfully learned to take the most optimal action. As we see later, this is not always the case.

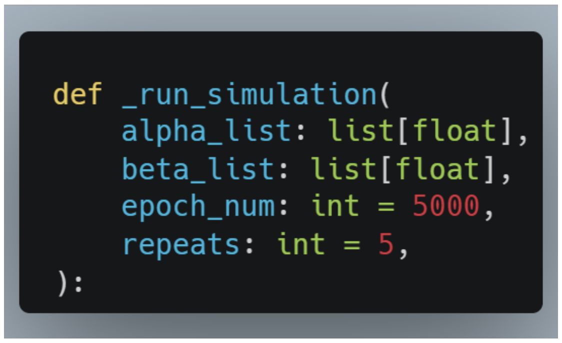
Implementation Details

3 functions were written for the question.Their signatures are presented below.
The first one let's users run a single simulation based on the required alpha, beta, and epoch_number.



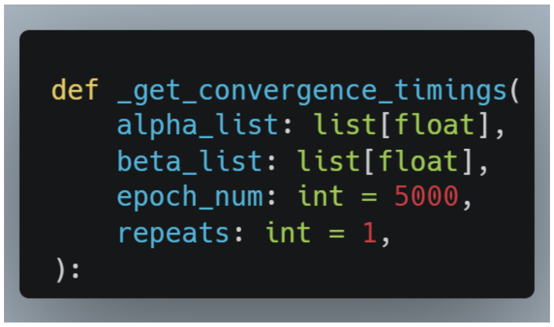
```
def run_automata(
    alpha: float, beta: float = 0, epoch_num: int = 5000
) -> tuple(float, list[float]):
```

The second function let's users run a single simulation the desired number of times to better predict automata's behavior, which is stochastic and highly dependent on the initial values of q_t=probabilities array. This would allow us to make better conclusions based on the test data.



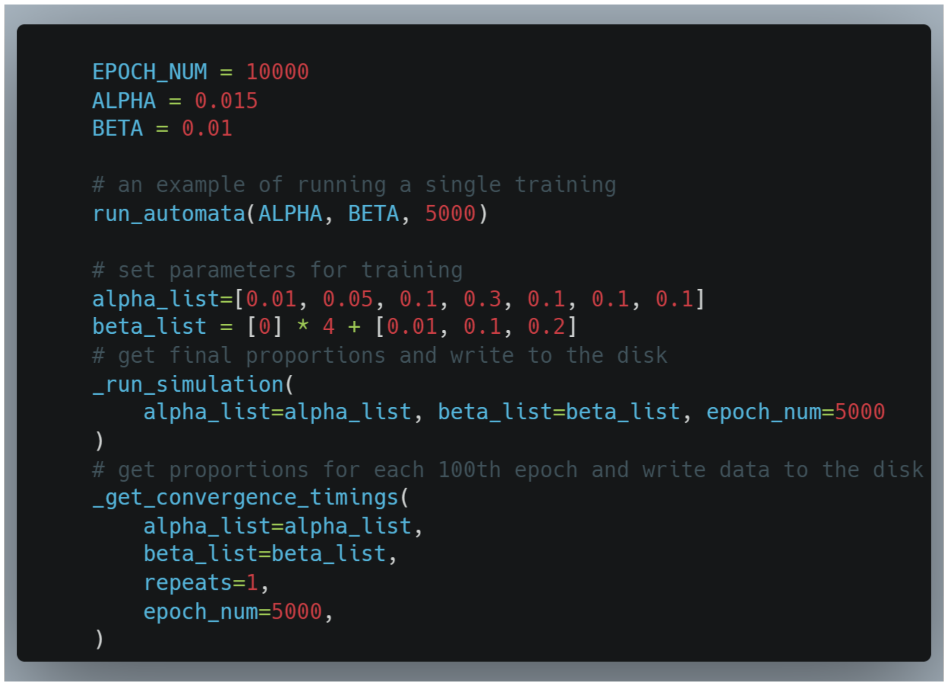
```python
def _run_simulation(
    alpha_list: list[float],
    beta_list: list[float],
    epoch_num: int = 5000,
    repeats: int = 5,
):
```

Finally, the third function can be used to get a better understanding over the convergence rate and speed, as it records the proportion of cases when the optimal action was chosen every 100 epochs. This way, it is possible to see if the reward converged fast enough, see the fluctuations, etc.



```python
def _get_convergence_timings(
    alpha_list: list[float],
    beta_list: list[float],
    epoch_num: int = 5000,
    repeats: int = 1,
):
```

Their implementation details can be found in the code. Nevertheless, the sample usage is presented below for convenience.



```
EPOCH_NUM = 10000
ALPHA = 0.015
BETA = 0.01

# an example of running a single training
run_automata(ALPHA, BETA, 5000)

# set parameters for training
alpha_list=[0.01, 0.05, 0.1, 0.3, 0.1, 0.1, 0.1]
beta_list = [0] * 4 + [0.01, 0.1, 0.2]
# get final proportions and write to the disk
_run_simulation(
    alpha_list=alpha_list, beta_list=beta_list, epoch_num=5000
)
# get proportions for each 100th epoch and write data to the disk
_get_convergence_timings(
    alpha_list=alpha_list,
    beta_list=beta_list,
    repeats=1,
    epoch_num=5000,
)
```

Practical Results

Overall proportion of Cases

One of the easiest parameters that can be used to check if an automaton is performing well is just to see if the proportion of times when the optimal action was taken out of the overall number of iterations is high. To be precise, we use the following formula:
_proportion = total_optimal_num / epoch_num * 100, where epoch_num is the total number of iterations to train the automata for. This parameter i**ndicates the proportion of cases when the automata chose to do the optimal action.**

This way, we'll see how well the automaton is taking the optimal action. The following table has been obtained after running the function _run_simulation with the parameters displayed on the previous page.

alpha	beta	Proportion on Last Epoch
0.01	0	43.78
0.01	0	91.54
0.01	0	48.12
0.01	0	68.94
0.01	0	81.86

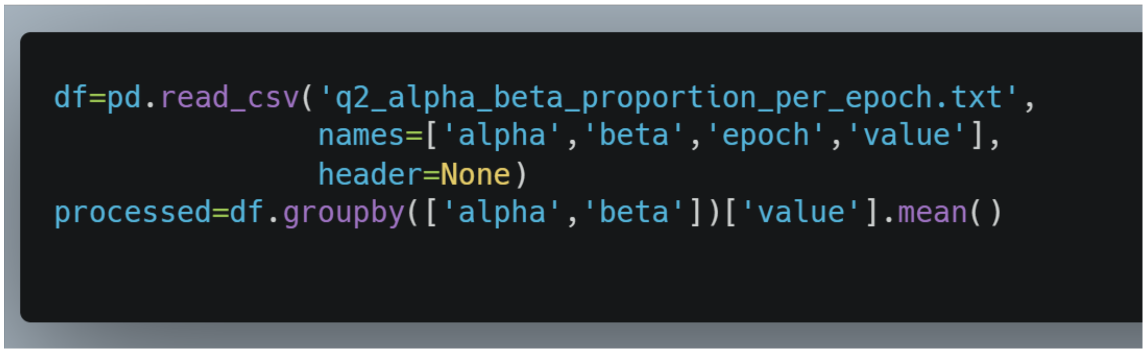
0.05	0	97.1
0.05	0	90.32
0.05	0	93.12
0.05	0	0.5
0.05	0	98.64
0.1	0	0.2
0.1	0	0.12
0.1	0	0.04
0.1	0	96.6
0.1	0	0.64
0.3	0	99.8
0.3	0	5.4
0.3	0	3.1
0.3	0	0.4
0.3	0	4.9
0.1	0.01	13.52
0.1	0.01	94.18
0.1	0.01	65.96
0.1	0.01	79.9
0.1	0.01	0.1
0.1	0.1	21.02
0.1	0.1	68.72
0.1	0.1	66.6
0.1	0.1	23.38
0.1	0.1	44.96
0.1	0.2	18.14
0.1	0.2	13.92
0.1	0.2	26.28
0.1	0.2	15.52
0.1	0.2	28.44

Clearly, we can see the stochastic component of our algorithms. For example, for alpha=0.1 and beta=0, the automata reaches a very respectable result of 96.6 % once, while 4 times it reaches less than 1 %. The same is true for alpha=0.3, where the automata reached 98.8 % once but less than 10% in other cases.

The reason for that is the fact that the algorithm is highly dependent on the initial values of q_t(denoted as probabilities in the code). If the probability for the optimal action is much higher than the rest of probabilities, the algorithm will successfully converge to it. Otherwise, it will keep fluctuating between different values that are close to each other, ending up not choosing the best possible action.
At the same time, one could argue that the linear reward-penalty algorithm is superior to the linear-inaction reward algorithm because it is reaching relatively better results *overall* compared to the linear reward algorithm. It is evident that the reward-penalty algorithm is more robust to the randomness introduced to q, as it is either performing better *on average* in the worst case scenario or just very well. On the other hand, the linear-inaction algorithm is either performing very poorly (choosing the optimal action in less than 10% of cases) or very well (choosing the optimal action more than 90% of the times). In this experiment, the best parameters for alpha and beta were determined to be 0.1 and 0.01 respectively, making sure that the automaton is penalized for choosing the wrong action but still can explore it more. If the penalty is too high or 0, the exploratory aspect is negligible,for ex. because the automata reached a very high value of p_t for the first chosen action, which might not be optimal. This creates some kind of feedback loop, where the algorithm is unable to explore other potentially better actions because it also increases the p_t for one action too high. Additionally, if the penalty is too high the algorithm will not be eager to explore other alternatives to the action that might be producing good results. For example, If one action has a probability of success as 56% and the other 60%, the automaton might end up using the former only because it got penalized severely a few times on the latter action, even though it is more optimal and better long term.

Check Convergence Rate

In order to actually see the convergence rate, the data for every 100th epoch has been recorded. Due to the stochastic nature of both algorithms, it should be noted that these results are not fully representative. Ideally, each case must be run at least a few dozen times to get a better picture. For the sake of time, this step was not done.
Additionally, due to the sheer amount of data, the following code was used to obtain the average rate for every algorithm.


```
df=pd.read_csv('q2_alpha_beta_proportion_per_epoch.txt',
               names=['alpha','beta','epoch','value'],
               header=None)
processed=df.groupby(['alpha','beta'])['value'].mean()
```

The results are presented below. For the original data, please consult the file "q2_alpha_beta_proportion_per_epoch.txt".

alpha	beta	mean_rate
0.01	0	29.2544
0.05	0	45.6724
0.1	0	47.5924
0.1	0.01	48.1732
0.1	0.1	11.5624
0.1	0.2	9.55

As in the previous part, the best convergence rate has been achieved by the algorithm with alpha=0.1, beta=0.01, followed by alpha=0.1, beta=0.1. Evidently, these parameters let the algorithm keep the right balance between exploration and exploitation in the fastest way possible. According to the results, the best balance is achieved with a small value of beta and a relatively high value of alpha.

Conclusion

The best results for alpha and beta were found to be 0.1 and 0.01 , respectively, followed by 0.1 and 0. It is important to note that the algorithms' performance is highly dependent not only on the alpha and beta but also on the randomly generated array of probabilities q_t. The reason for that is the fact that if the probabilities in q_t are close to each other, both algorithms might not converge to the required optimal action because they might have already tried an action with a slightly smaller probability of success, and already increased its p_t value. This is a serious disadvantage that the UCB algorithm doesn't have, as according to our results it almost always converges to 100%(it ends up choosing an optimal action in the majority of cases). Thus, it might be preferable to use UCB in the general case, while using the linear automatas only in specific cases with known initial probabilities q_t, where the most_optimal action can be easily identified.