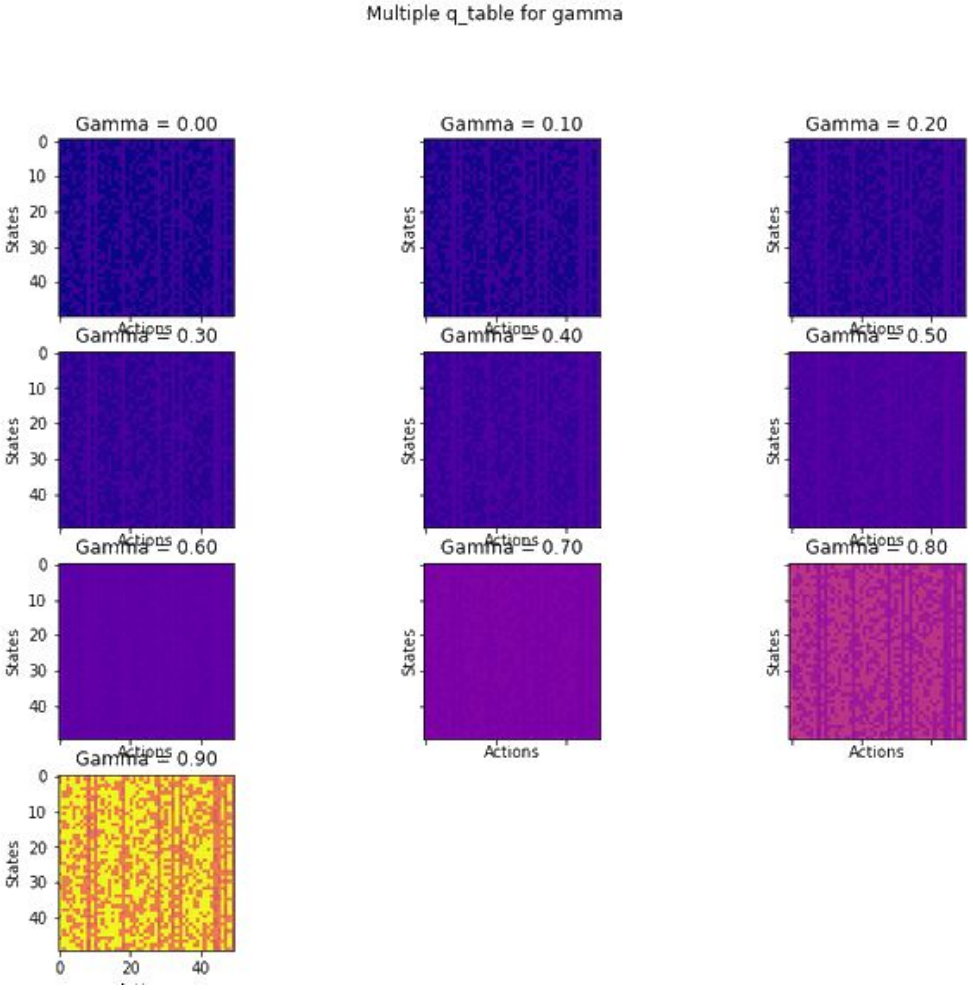
Report 05/05 : Reinforcement learning for Cache-Friendly Recommendations

Jade Bonnet and Clément Bernard

1) Normalize the q_table for different values of gamma

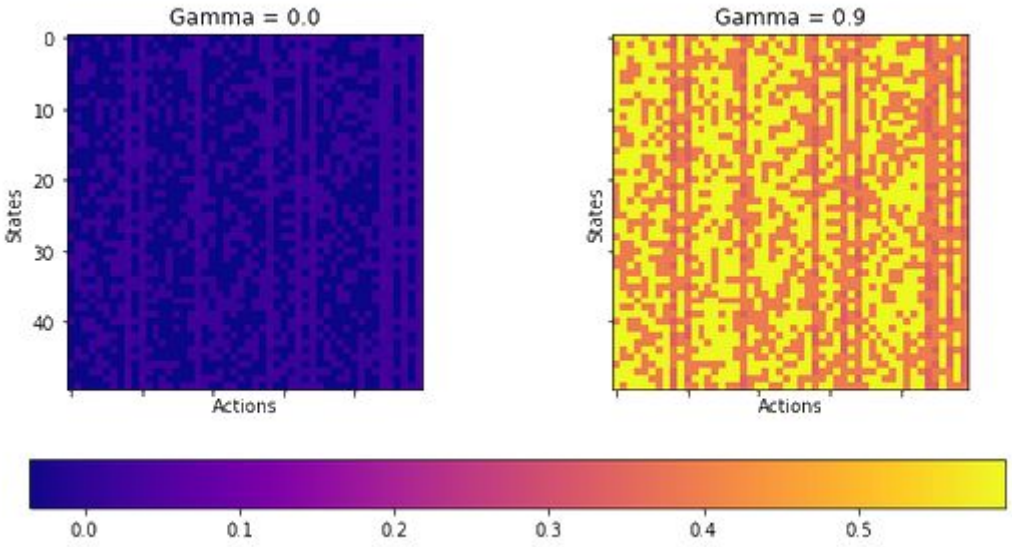
We normalized the q_tables to visualize them in a better way.



Normalized q_tables for different values of gamma (200 000 epochs)

We can see that the values of the q_table for high values of gamma are higher than for low values of gamma (as expected).

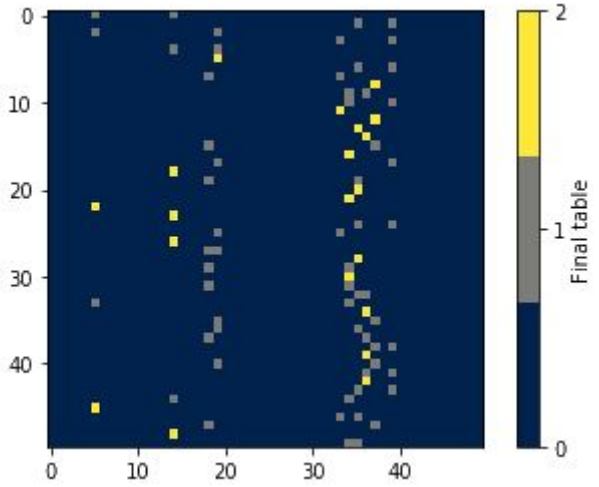
Here is the plot for gamma = 0 and gamma = 0.9 :



Normalized q_table for gamma = 0 and gamma = 0.9 (200 000 epochs)

2) Compare reward matrix row by row with q_table

We computed the final matrix which is basically the max of each row. We only took one of the maximum.
We then compared it for gamma = 0 and gamma = 0.9. We made +1 in the table if the action is the one that will be recommended.
In this case, the table will have values equal to +2 if it will be recommended by both the q_table with gamma = 0 and 0.9.

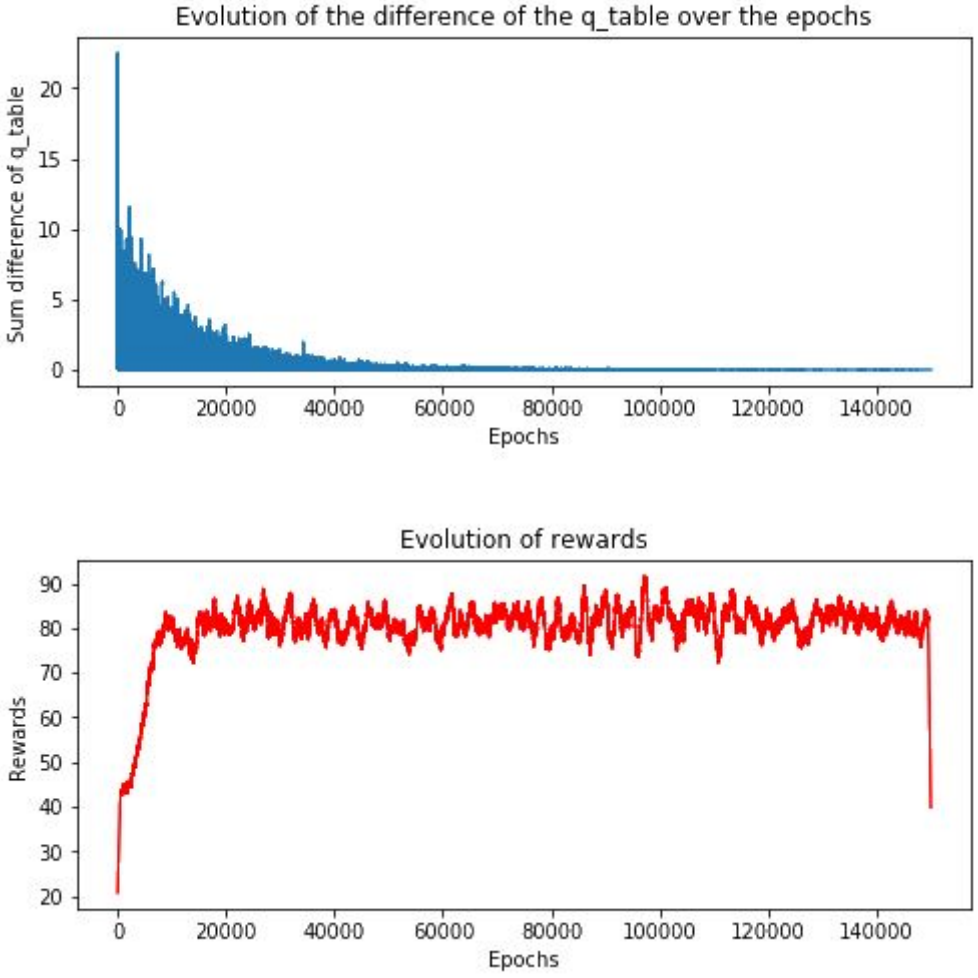


Comparison of final reward for gamma = 0 and gamma = 0.9

In this case, there is 25% of similitude.

3) Find convergence criteria

As there is some noise in the sum of rewards through the epochs, using a batch could have been a solution. Nevertheless, we've found another metric to see when the algorithm has converged. Here is the evolution of the total difference between two q_table over the epochs :

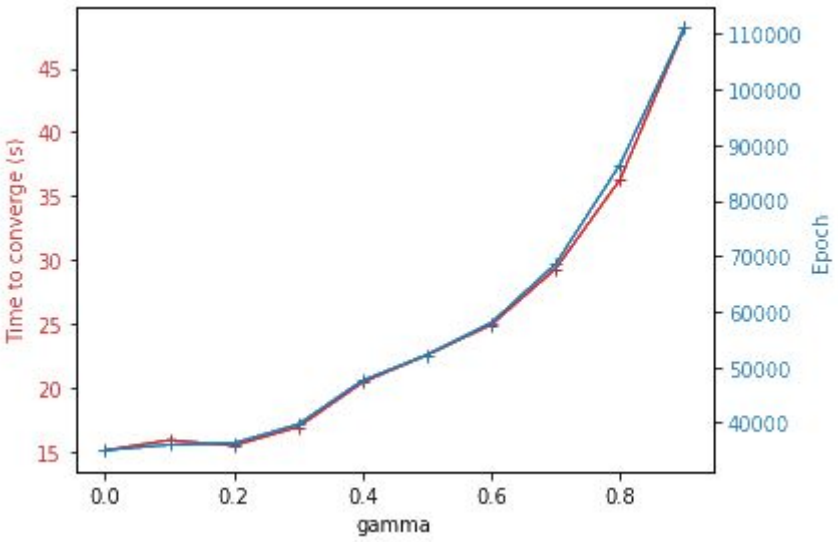


Evolution of the sum difference of q_table over the epochs with rewards

Therefore the criteria to converge will be that the sum of difference of rewards over 100 epoch is less to 0.05.

4) Time of convergence for different rewards

We computed the time to converge and the number of epochs required for each value of gamma. Here is the result :

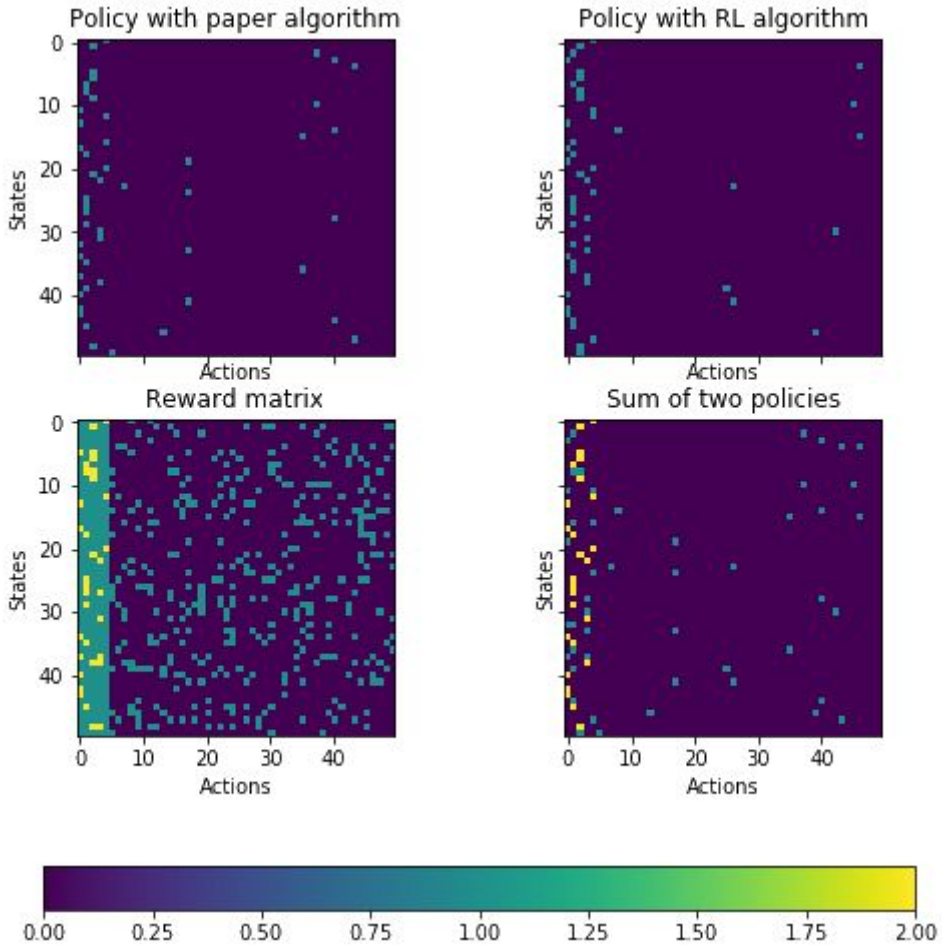


Time and epoch to converge for different values of gamma

We observed that the final table obtained is the same after this epoch.

5) Check U matrix with the classic solution

We discussed with Theodoros. He gave us a U matrix, the cached contents, and the policy obtained by his algorithm.
We then compared to our policy obtained using the same parameters with gamma = 0.9.
Here is the plot of the results :



Comparison with the paper's algorithm

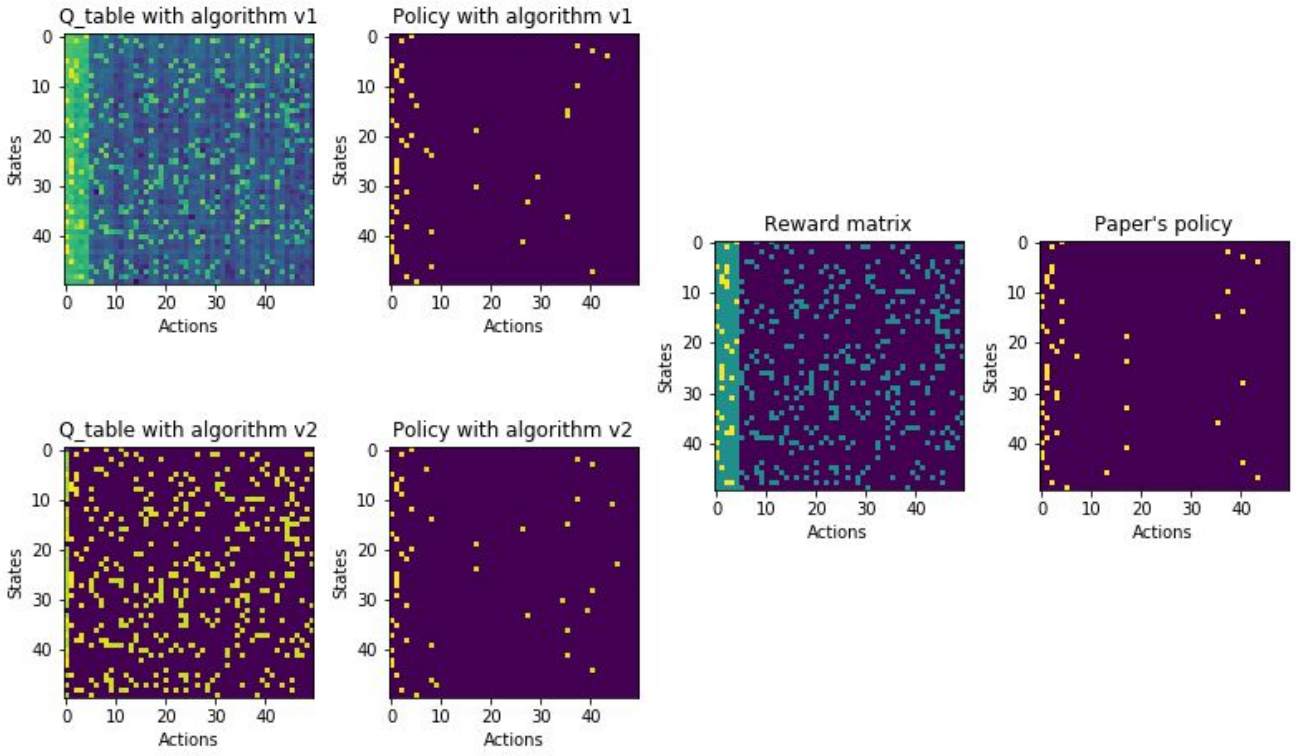
We did it twice (that is to say with two different policies and U matrix) and we get a similitude of 1) 52% and 2) 42%.
Actually, our algorithm relies on the rewards we set, whereas the paper's algorithm doesn't use a system of rewards as we did. It could explain the percentage of similitude.

WE NOTICED A MISTAKE IN OUR CODE :
In the ‘step’ function (which simulates a step of our user), here is the behavior of our environment :

```
def step(self, action,state) :
    # Returns the state, reward after taking the action in input
    # done is a boolean to say whether the user quits the game or not
    # We want to return the state where will be the user when we suggest him "action"
    # Knowing he is in the current 'state'

    if (random.uniform(0, 1) < self.to_leave) :
        # The user stops to play
        new_state,reward, done = None,None,True
        return new_state,reward, done
    else :

        # Else the user will choose among the contents
        if (random.uniform(0,1)< self.alpha ) :

            # Then the user chooses a content among the recommended contents

            recommended_contents =  self.recommended[state]
            index_state = random.randrange(0,len(recommended_contents),1)
            new_state = recommended_contents[index_state]
            reward = self.find_reward(action,state)
        else :
            # The user picks a content randomly in the catalogue

            new_state = get_random_state(self.p0)
            reward = self.find_reward(action,state)


        done = False

    return new_state, reward, done
```

‘step’ function from our environment

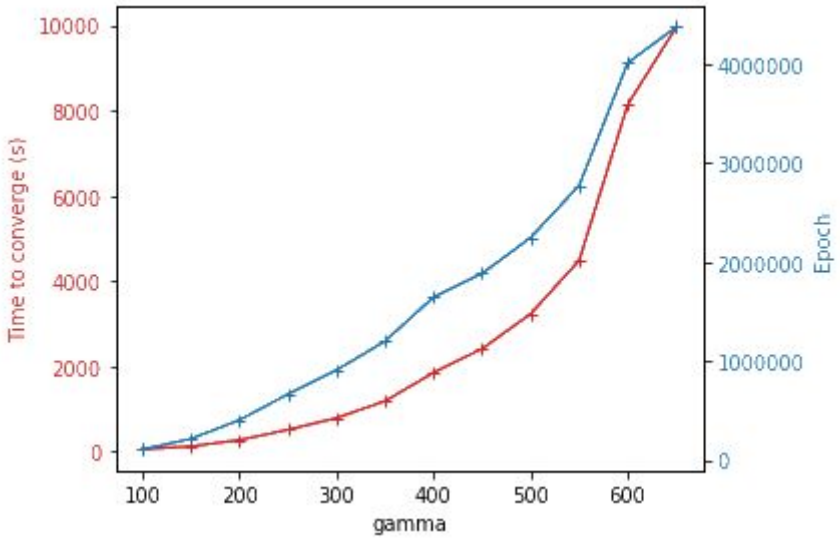
Indeed, if the action comes from the exploration part, the user won’t consume this content. We changed this and we didn’t know if, for exploring, we should consider either every content in the catalogue or the one that are related.
We tried both as **v1 (search all actions)** and **v2 (only considering the most related)**.



Comparison policy between paper's algorithm and v1/v2 algorithm

6) "Kill the computer"

We are currently running our algorithm to see how much we can compute.
Here is a result :

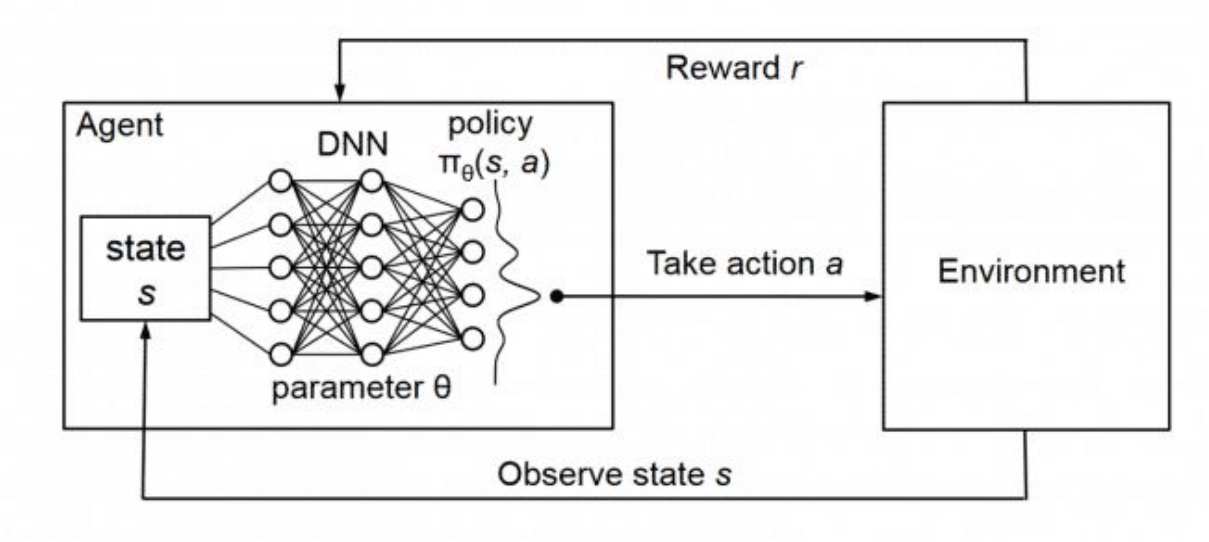


TIme and epoch to converge for different values of states/actions

We had some issues to train more (800 and more) because it took more than 4 hours.

7) Deep q learning

As the time to converge for more than 1000 states is really high, we can't rely only on this algorithm.
Hence, a new way to deal with it is to use a function approximation, for instance a neural network, which will take as input a state and will approximate the Q-values for each action based on that state.



Deep q learning schema

We started some tutorials to understand it.