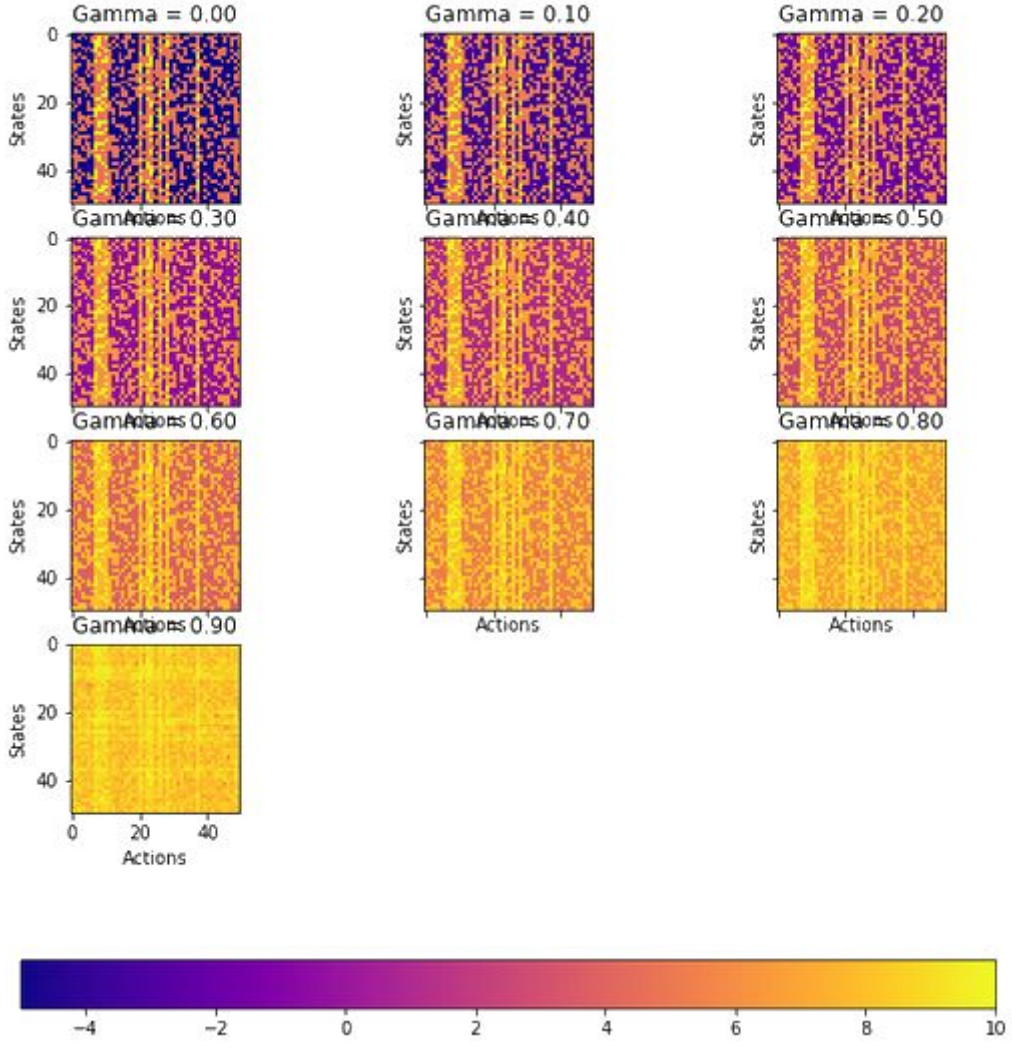
Report 19/05 : Reinforcement learning for Cache-Friendly Recommendations

Jade Bonnet and Clément Bernard

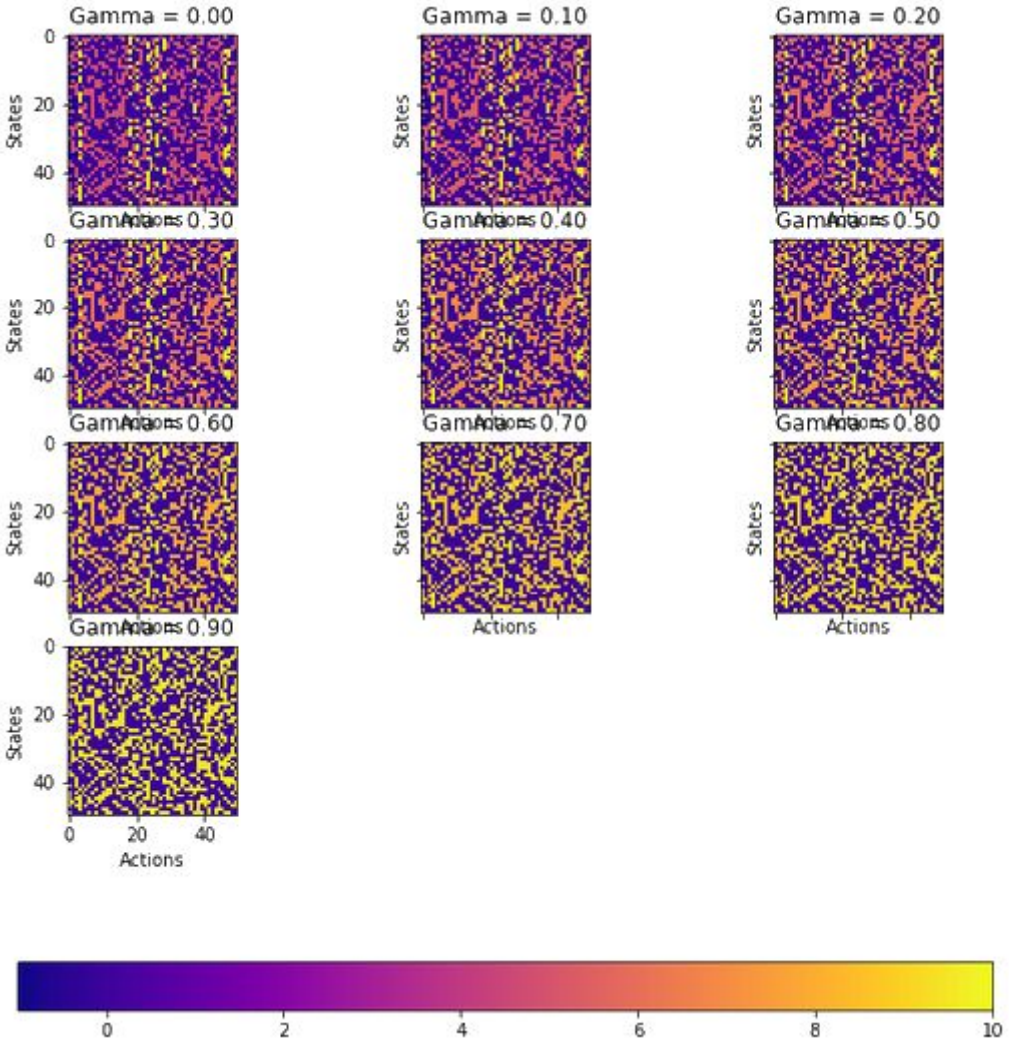
1) Q learning algorithm

a) Normalize the q_table for different values of gamma

We normalized the q_tables by dividing the table by $\frac{1}{1-\gamma}$.
The aim is to see the value per step.



Normalized q_tables for different values of gamma (algorithm v1)

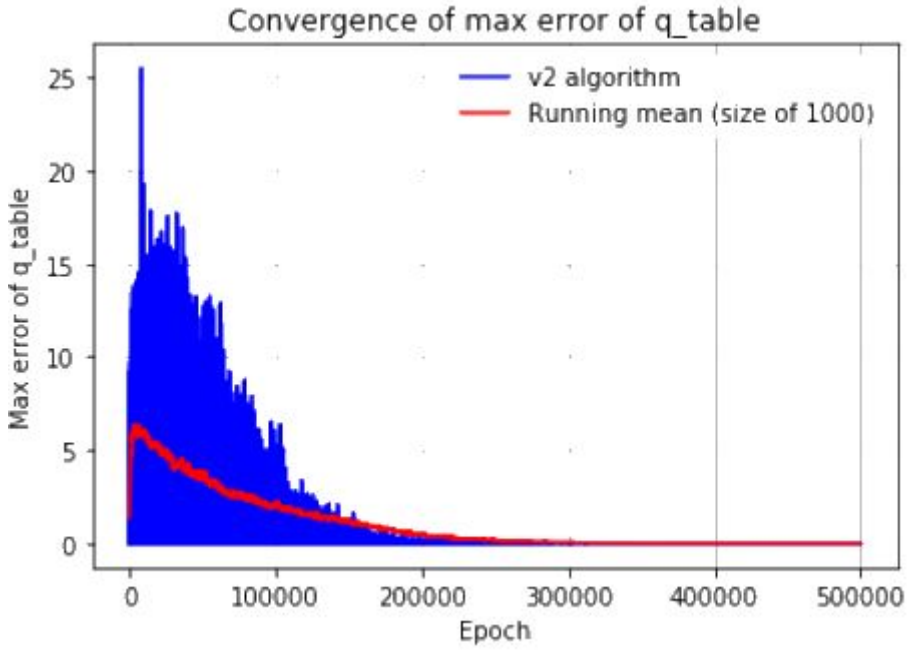


Normalized q_tables for different values of gamma (algorithm v2)

The second plot shows that, for low value of gamma, the agent tends to recommend content with high current rewards whereas for gamma = 0.9, it can recommend contents which don't always lead to high current reward.

b) Find convergence criteria

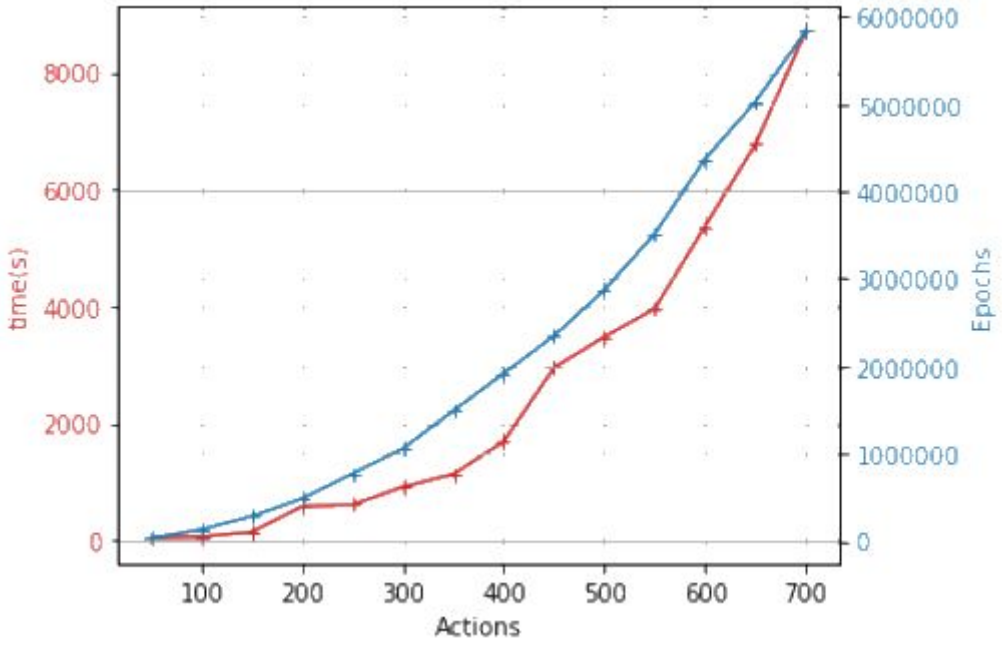
As suggested, we used the maximum difference between the q_table as a criteria of convergence. Here is the plot of this error depending on the epochs :



Evolution of the max error of q_table over the epochs

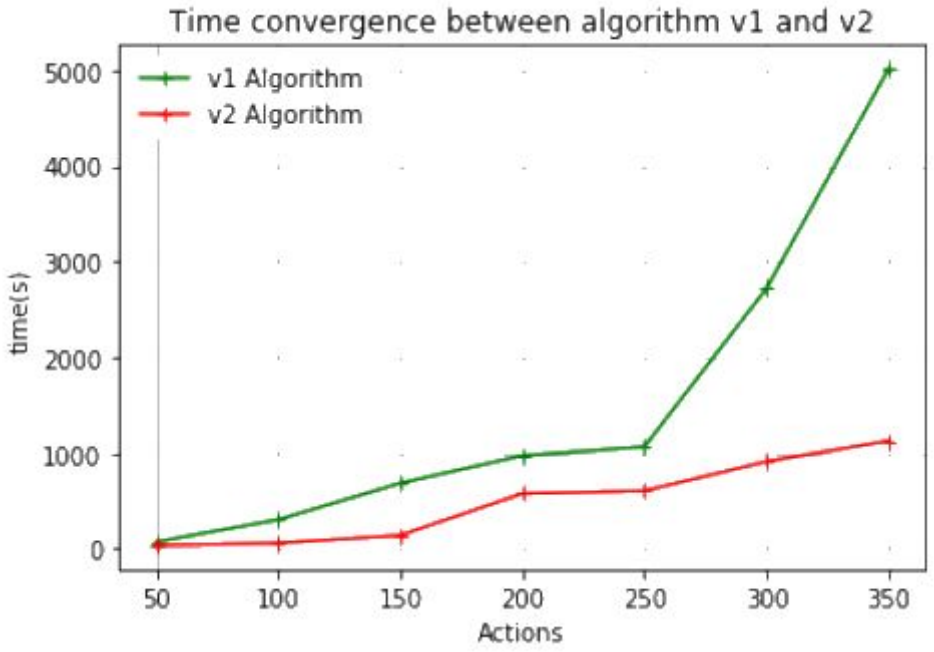
c) Time of convergence

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 (v2 algorithm)

We used the v2 algorithm because it is faster that the first one, as it only explores over the related contents. Here is the comparison between the v1 and v2 algorithm in term of time of convergence :



Time of convergence for algorithm v1 and v2

d) Comparison with a specific user

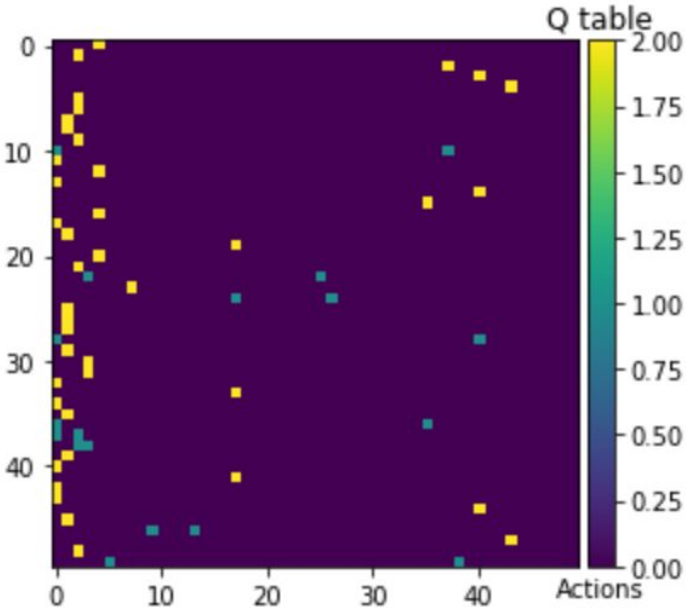
We changed the behaviour of the user as follow :

When the user is in state **i**, he is likely to accept the content **j** that we suggest with the probability : $\alpha_{i,j} = \frac{u_{i,j}}{max_i(u_i)}$

We also make the rewards only on the cached contents.

We didn't have the same policy than the paper had. We don't know why. We got 82 % of similarity for both experiences with the v2 algorithm (that only recommend related contents)

```
Pourcentage of similarity 82.0 %
```



Sum of the two policy (Theo's policy and our policy)

2) Deep q learning

We used Keras interface to build our neural network.

a) Memory

For the training of our Neural Network, we used what is called **experience replay**.
We stored the agent's experience at each time step in an object called **'Memory**'.
At each time step, we store the experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$
We only memorized the **mem_size** last experience. We randomly sampled it to train the network. We use it to break the correlation between each consecutive sample.

b) Architecture

We chose a neural network with two hidden layers. Here is how we defined it :

```
Model: "sequential_29"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_87 (Dense)             (None, 24)                48
_________________________________________________________________
dense_88 (Dense)             (None, 24)                600
_________________________________________________________________
dense_89 (Dense)             (None, 50)                1250
=================================================================
Total params: 1,898
Trainable params: 1,898
Non-trainable params: 0
_________________________________________________________________
```

Summary of the network

c) Target

What we want to approximate is the following target :

$$target = R(S_t, A_t) + \gamma max_{a'} \hat{Q}\ (S_{t+1}, a')$$

d) Loss

Therefore we defined the loss as follow :

$$loss = (\ R(S_t, A_t) + \gamma max_{a'} \hat{Q}\ (S_{t+1}, a') - Q(S_t, A_t)\ )^2$$

e) Replay

We trained the neural net with experiences that came from the memory.

```
def replay(self, batch_size):
    '''
    Fit the network with a batch of experiences
    Compute the target from this batch
    The target is computed as follows : r + gamma * max(Q(s',a'))
    '''
    minibatch = self.memory.sample(batch_size)
    # Loop over the batch
    for exp in minibatch:

        target = exp.reward
        if not exp.done:
            target = (exp.reward + self.gamma * np.amax(self.q_network.predict(np.array([exp.next_state]))[0]))
        # Compute the prediction
        target_f = self.q_network.predict(np.array([exp.state]))
        # Replace the prediction by the target
        target_f[0][exp.action] = target
        # Fit the neural network with the new target to update the weights
        self.q_network.fit(np.array([exp.state]), target_f, epochs=1, verbose=0)
```

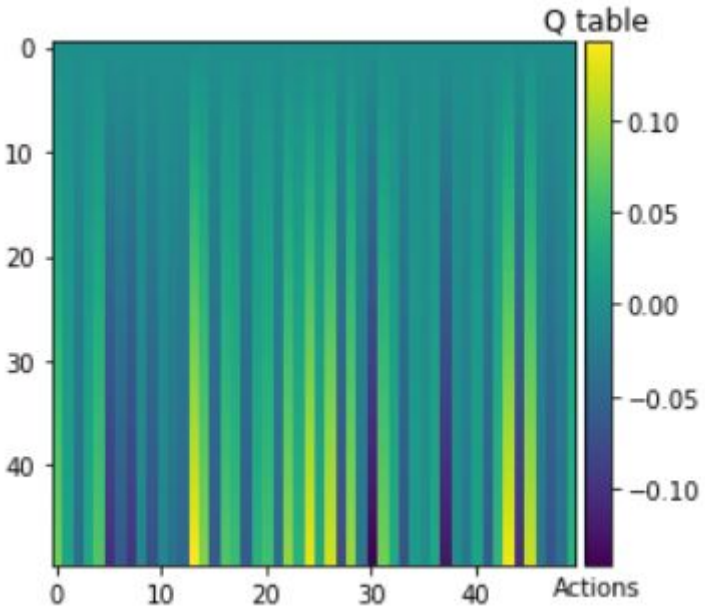
f) *Act*

The agent is choosing an action with the epsilon-greedy way :

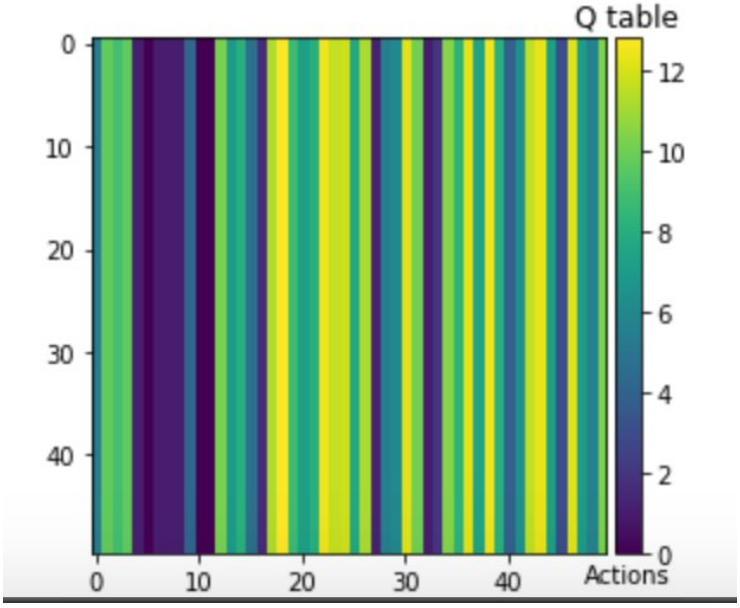
```
def act(self, state) :
    '''
    EPSILON-GREEDY Decision : Exploitation or Exploration
    '''
    if np.random.rand() <= self.epsilon:
        # Exploration
        return random.randrange(self.n_actions)
    else :
        # Exploitation
        act_values = self.q_network.predict(state)
        return np.argmax(act_values[0])  # returns action
```

g) Results

Our network can predict the Q value. Nevertheless, we have not achieved yet to make it converge. Indeed, it seems that it tends to recommend the same thing for each state, which is not what we except. Maybe this is due to the architecture. Here is a comparison between before the training and after (500 epochs).



Result of the network at initialisation



Result of the network after 500 epochs