Reinforcement learning for cache-friendly recommendations : Partial report

Clément Bernard and Jade Bonnet

June 17, 2020

1 Introduction

Radio applications (like Deezer, Spotify) use recommendation algorithms to create playlists for the users that take into account the user's preferences and the relations between contents. Similarly, Youtube generates a sequence of videos guiding the user to a session. Nevertheless, these recommendation algorithms don't take into account the network (for instance the network latency). We proposed to solve this issue by implementing reinforcement learning algorithms that take into account both the network and the preferences of the user. The reinforcement learning algorithms aim to learn from experience by interactions with the user.

2 Problem

Catalogue We consider that the user can choose a content among a catalogue $\mathcal{K}$ of size K. This catalogue can be adapted to different configurations like videos or musics.

Recommendation System To know which content is related to another, we assume a matrix $\mathcal{U} \in \mathcal{R}^{\mathcal{K}\times\mathcal{K}}$ where $u_{i,j} \in [0, 1]$. Therefore, $u_{i,j} = 0$ means that the content j isn't related to the content i and $u_{i,j} = 1$ means that the content j is highly related to the content i.
Furthermore, we also assume that we recommend only 1 content after the user has watched one.

Caching cost Each content is associated with a cost, that can represent the network latency. For a content i, the cost associated is denoted $x_i \in \{0, 1\}$. If a content is cached, the cost is $x_i = 0$ and whenever $x_i = 1$ the content is non-cached.

User models We model two different types of users.

1. **Markovian User** : After this user has consumed a content i, he will :
   (a) Leave the simulation with probability α
   (b) Stay in the simulation with probability $1 - \alpha$ and :

        i. Pick the content we offer to him with probability a
        ii. Choose with probability $1 - a$ a content j among the catalogue with probability p_j, where $p_j \in [0, 1]$ and $\sum_{j=1}^{K} p_j = 1$. Therefore, it will ignore the recommender

2. **Specific User** : After this user has consumed a content i, he will :
    (a) Leave the simulation with probability α
    (b) Stay in the simulation with probability $1 - \alpha$ and :
        i. Accept the content j with probability $a_{i,j} = \frac{u_{i,j}}{max_i(u_i)}$
        ii. Otherwise, he will choose a content j like before : with probability p_j, where $p_j \in [0, 1]$ and $\sum_{j=1}^{K} p_j = 1$.

Recommendation process We aim to solve a recommendation issue by interacting with the user. Indeed, by using the reinforcement learning approach, the algorithm will interact with the user to learn how he works (like its preferences). Furthermore, the algorithm doesn't aim to only learn the user's behaviour : it will take into account the cached contents to create a trade-off between proposing cached content or related contents.

3 Reinforcement learning algorithms

3.1 Definition

Reinforcement learning This is a subclass of machine learning, such as *supervised learning* or *unsupervised learning.* Reinforcement learning is learning how to map situations to actions in order to maximise rewards. The learner is not told which action to do, but should discover which action results in higher rewards by trying them. The challenges of reinforcement learning are the fact that a given action has impacts in the immediate rewards but also in the next situations, and so the future rewards. Reinforcement learning is therefore a trade-off between learning by interaction and delayed rewards.

3.2 Elements of Reinforcement Learning

3.2.1 Markov Decision Processes

The finite Markov Decision Processes (MDPs) are a classical way to model sequential decision making, where the actions influence not immediate rewards but also future states and rewards. It helps us to create a mathematical model for our problem.
A reinforcement learning system has several main elements that include an Agent, Environment, State, Action, Reward, Policy, and Value Function.

Agent The learner that will make the decisionscof what actions to take. In this case the agent is the recommendation algorithm. The decisions he will take at time t is noted A_t

Environment Things that interact with the Agent. It will interact with the agent by returning a Reward R_{t+1} and the next States S_{t+1}. In our case, the environment is the user himself.

Reward Feedback on the actions taken by the agent, it can measure the success or failure of the actions taken. In our case, we give positive rewards if the content we suggest is either related or cached.

Action Content that is suggesting by the agent. This is an element among the catalogue $\mathcal{K}$.

State Content that is currently consumed by the environment (user). This is also an element among the catalogue $\mathcal{K}$.

In a MDP, the goal of the agent is to learn what actions will lead to best rewards through a finite sequence of interactions with the environment.

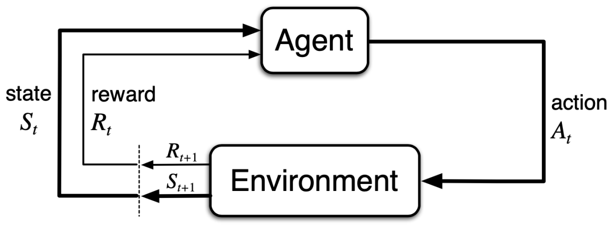


Figure 1: Agent-environment interactions in a MDP

The agent and the environment will interact at each sequence of time t = 0,1,2,3,4,... At each timestep t, the agent receives a state S_t and will offer an action A_t from what he knows. Then, the environment, as a result of this action, will return a reward R_t to this action and lead the agent to a new state S_{t+1}. It leads to a trajectory like this :

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, ...$$

In our problem, a timestep corresponds to a content that is consumed (for instance a video or a music for example).

State-transition probability With the previous notations, we can define a state-transition probability which describes the probability to be in a next state S_{t+1} given the current state S_t and an action A_t :

$$P(S_{t+1} = j | S_t = i, A_t = k) = \begin{cases} (1-\alpha)a & if \quad j = k \\ (1-\alpha)(1-a)p_j & if \quad j \neq k \end{cases}$$

3.2.2 Returns, policy and value function

So far we have defined our problem in term of MDPs, but we haven't defined yet the objective of the learner.

Return This is the cumulative sum of the rewards from step t. It is defined as follow :

$$G_t = R_{t+1} + R_{t+2} + ... + R_T$$

where T denotes the timestep where the simulation ends (the user has left the simulation for instance).

Expected return The previous formulation of the return doesn't take into account the fact that a current reward could have high impact at time t that the one at time T. To cope with it, we add a **discount** factor γ with $0 \leq \gamma \leq 1$. The expected reward is then defined as :

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-1} R_T = \sum_{k=0}^{T} \gamma^k R_{t+k+1}$$

The discount factor γ represents the value of future rewards : $\gamma = 0$ means the agent is 'myopic' and only considers maximizing current rewards whereas if γ is close to 1, the agent takes into account future rewards.

Policy Mapping from states to probabilities of selecting an action. If an agent is following a policy π at time t, then $\pi(a|s)$ gives the probability of choosing the action $A_t = a$ from state $S_t = s$.

Value function The value function of a state s under a policy π, denoted as $v_\pi(s)$ is defined as the expected return when starting from the state s and following the policy π :

$$v_\pi(s) = E_\pi(G_t|S_t = s)$$

Action-value function As described above, we define the value of taking action a in state s under a policy π, denoted $q_\pi(s, a)$, as the expected return from state s, taking the action a and following the policy π :

$$q_\pi(s, a) = E_\pi(G_t|S_t = s, A_t = a)$$

3.3 Q-Learning

3.3.1 Optimality

There are different policies that can handle a reinforcement learning problem. Nevertheless, some policies are better than others. But what does better mean in MDP ?

Order in term of policy A policy π is better than a policy π' if its expected return is greater than or equal to that of π' for each state.

Optimal policy There is always at least one policy which is greater than or equal to every other policies. Same idea for the action-value function. We denote them as follow :

$$v_*(s) = \max_\pi v_\pi(s)$$

and

$$q_*(s,a) = \max_\pi q_\pi(s,a)$$

3.3.2 Bellman optimality equations

The Bellman equations express the relationships between the value of a state and the value of the next states. Furthermore, the Bellman optimality equations express the fact that the value of a state under an optimal policy should be equal to the expected return for the best action from that state :

$$v_*(s) = \max_a q_{\pi_*}(s,a) = \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')]$$

There is the equivalent equation for the state action value function :

$$q_*(s,a) = \sum_{s',r} p(s',r|s,a)(r + \gamma \max_{a'} q_\pi(s',a'))$$

This is actually a system of equations for each state.
Once the optimal function of state-action pairs is determined, one can infer the optimal action to take from a given state s :

$$argmax_{a'} q_*(s,a')$$

Therefore, after determination of the optimal Q table, we know which content to recommend for the user for each content in the catalogue.

3.3.3 Q learning algorithm

To learn how to approximate the optimal state action pairs, we can approximate directly q_*. To do so, we use this following formula :

$$Q_{t+1}(S_t,A_t) = Q_t(S_t,A_t) + \alpha(R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q_t(S_t,A_t))$$

It leads to the Q learning algorithm :

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Figure 2: Q learning algorithm

3.3.4 Exploration and Exploitation (ϵ-greedy)

In the above algorithm, the selection of an action by the agent (and so the content to recommend) is chosen in a specific way. Indeed, there is a trade-off between :

1. **Exploration** : with probability ϵ, the agent will pick randomly a content among the catalogue. The purpose is to enable the agent to continue to explore the possible actions in order not to avoid some decisions.

2. **Exploitation** : with probability $1 - \epsilon$, the agent will pick the action that maximises the q values : $argmax_a Q(S_t, a)$

This is very important to keep this trade-off : avoiding the exploitation will lead to random recommendations whereas avoiding the exploration will lead to eventually bypass good actions.
Furthermore, one can consider to start with a high ϵ to advantage the exploration (because we have no knowledge of the expected rewards yet) and then gradually decreasing the ϵ as the agent learns more and more about the environment.
Nevertheless, we don't consider this approach because it is equivalent to assume that the user doesn't change and he will keep his preferences. We don't want to assume these and so we keep ϵ constant.

3.3.5 Convergence criteria

There are two ways to consider the end of the algorithm, and therefore to get the final Q values (which should correspond to the optimal Q values) :

1. **Number of epochs** : We simulate user experiences and in the same time we update the q-table for a number of times predefined. Then, we risk to have situation where the q-table values don't update anymore and we compute for nothing.

2. **Threshold** : We use a criteria to say whether the Q-table values have converged. To do so, we consider the maximum difference of q-tables values over an episode. That is to say, we let the user consume contents, and whenever he has terminated, we take the maximum difference between the previous q-table and the new one obtained after the simulation. If this value is less than a threshold (new hyperparameter), then we stop the process and we consider the q-tables as optimal.

3.3.6 Hyperparameters

There are some hyperparameters to take into account in the Q learning algorithm.

1. **Learning rate** α : it describes how fast we want to adjust the q values. This is as a classic supervised algorithm with gradient descent : a too high value of α leads to divergence whereas a too low value leads to increase the convergence time.

2. **Discounted** γ : It deals with the rewards that the agent considers to update the Q values. Indeed, if $\gamma = 0$ the agent will prefer actions with high current rewards whereas γ close to 1 will lead the agent to consider actions that lead to future high rewards. For instance, if $\gamma = 0.9$, it will take into account 10 future episodes.

Furthermore, there are also hyperparameters for the environment (and so the user we use). If the user is a Markovian one, the hyperparameters are the following :

1. **Cached contents** : Number of cached contents. This is what takes into account the network : the cached contents shouldn't cost anything to the user in order to consume it. On the other hand, a non-cached content can, if this is a popular one, lead to latency and decrease the user's experience.

2. **Related contents** : Number of related contents for each state. For instance, if the catalogue is a list of music, the related contents can be musics from same genre or artist. We keep this number constant for each state, and it shouldn't be more than 10 percent of the total size of the catalogue.

3. α : Probability to leave the experience. It corresponds to the end of a user experience. For instance, it could correspond to the time spent on Youtube watching consecutive videos.

4. **a** : Probability that the user follows our recommendations. This coefficient is set constant for sake of simplicity, but should evolve through the process. Indeed, if we recommend contents that the user dislikes, he won't trust the recommender and so will choose himself the future contents.

Note that for the specific user described above, the only difference is in the a coefficient.

3.4 Results

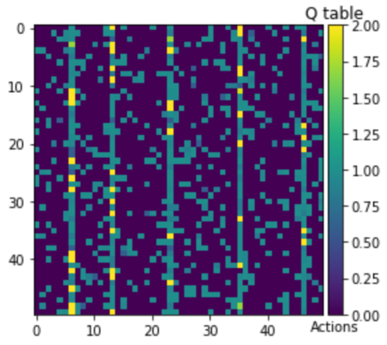
3.4.1 Q-table

We tried to solve two different problems :

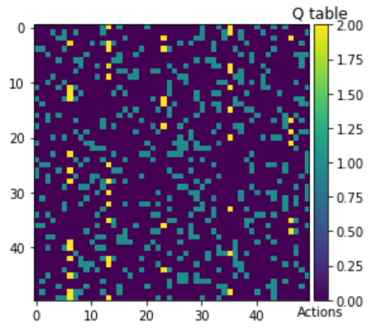
1. **Classic recommender** : In the exploration part, we consider all the contents with uniform probability.

2. **Restricted recommender** : In the exploration part, we only consider the related contents (we use the non null values in the U matrix).

Hence, we expect different behaviours for the final q-tables.
Note that we have also generated an environment where the catalogue has a size of 50. The cached contents are the contents 6, 13, 23, 35 and 46.
We also consider that a content that is related and cached will lead to a reward of +2, whereas a content that is either related or cached will lead to a reward of +1. Finally, a content neither related nor cached will lead to a reward of 0.
Here are the results of a q-table after applying the Q-learning algorithm for 100 000 epochs , with a random U matrix and cached list :



(a) Classic recommender



(b) Restricted recommender

Figure 3: Q-tables for $\gamma = 0$

To understand what these plots mean, it requires to read it line by line. In fact, for a given state (a line in these plots), the values we got in this line are the expected rewards for recommending this action (content). We can see that, as the cached contents are fixed, the vertical lines correspond to these contents. The expected rewards are usually high as the immediate reward, whatever state the environment is, is +1. Moreover, whenever a q-value is yellow (which means the q value associated is very high), the content is usually related and cached.

The difference between a classic recommender and the restricted one is that the restricted recommender tends to not consider every content in the catalogue. It leads, as shown above, to make the q-table values to 0 for contents that are cached (because they are not related to the specific state).

3.4.2 Discounted γ parameter

To understand the role of γ, we can focus on the optimal Q functions obtained for extreme values of γ. Here is the result, with the same environment defined above :

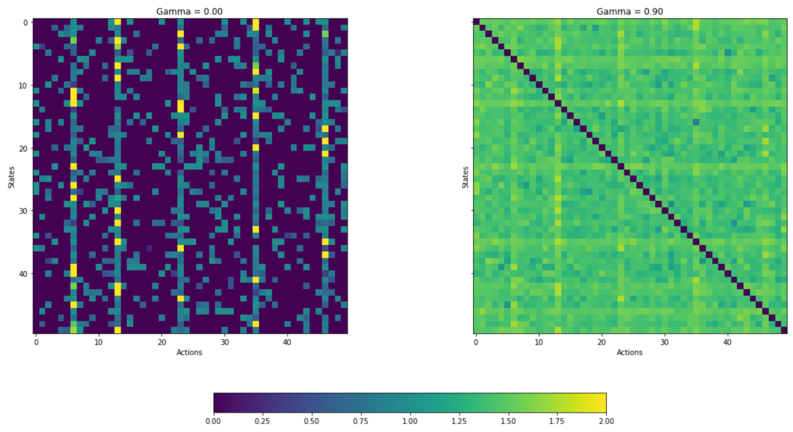


Figure 4: Q-table for $\gamma = 0$ (left) and $\gamma = 0.9$ (right) without constraints on the recommendations

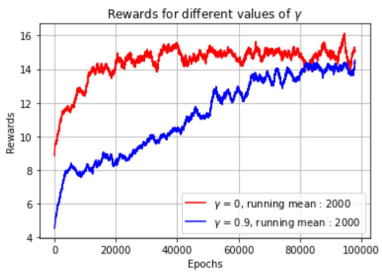
On one hand, the q-table for $\gamma = 0$ is predictable. This can be found with both the U matrix and the cached list. Indeed, it only considers immediate rewards, which is directly linked to the fact that a content is either related or cached.
On the other hand, for $\gamma = 0.9$, we see that this is more noisy. The Q-table values are very close. As it considers future rewards, even a not cached and not related content can lead to high rewards. That's why the values are so close. The diagonal is equal to 0 because we can't recommend the content that the user is currently consuming.

3.4.3 Rewards and penalties

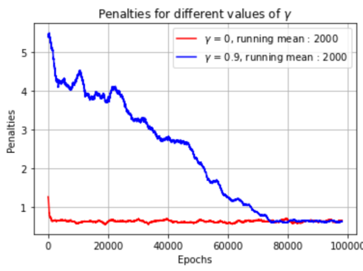
Rewards As the agent starts without any knowledge on the user, he will start by recommending bad actions (contents that will lead to low rewards). Through the epochs, the q-table will start to be completed and then the agent will be able to recommend good contents. It means that, if we look at the rewards, it will start to be very low (because the agent recommends bad contents) and will progressively increase.

Penalties We can also consider the quality of each decision of the agent. To do so, we consider what we call the *penalty*. This value is increased each time the agent recommend a content neither related nor cached (a content that leads to a reward of 0). In this case, we expect the penalty to be the contrary of the rewards : it will start very high because the user won't know anything about the environment, and then start to learn and so the penalty will decrease.

Note that for both the penalties and the rewards, we take the running mean to plot it because these values are very noisy (due to the fact that the user doesn't listen to the agent every time and also due to the exploration part).



(a) Rewards



(b) Penalties

Figure 5: Rewards and penalties for $\gamma = 0$ and $\gamma = 0.9$ without restrictions on recommendation

We used different values of γ for the plots. In fact, when $\gamma = 0$, the solution is easier to find compared to high values of γ (because it should consider future rewards and not immediate rewards). We see that, in average, after convergence, the agent suggests contents such as the total reward per episode is around 15 (the user is consuming in average 10 contents). Furthermore, for $\gamma = 0.9$, the agent starts by suggesting 5 contents that are neither related nor cached in a session of 10 contents in average. The start is really bad but it converges to a penalty of 0 (which means no bad contents recommended).

3.4.4 Specific user

We've tried to simulate a Markovian user, who listens to us with some probability, otherwise he picks randomly a content among the catalogue. There is another specific user that we tried to learn from. This is the user that accepts the content j from the state i with probability $a_{i,j} = \frac{u_{i,j}}{max_i(u_i)}$. This user is made to compare the policy to an optimization solution. The environment is composed of a catalogue of size 30, and there is only the 3 first contents that are cached. In the optimization algorithm, the process was run in 100 episode in average, which corresponds in our case to $\gamma = 0.99$.
We got 28 out of 30 contents that are similar in terms of policy :

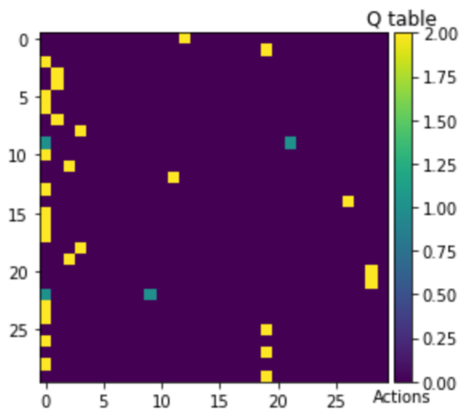


Figure 6: Sum of the optimised policy and our policy with constraints with Q-learning algorithm

In this plot, we represent the sum of our policy for the given user and the policy after application of the optimised algorithm. It tells us the final content to recommend when the user is consuming a specific content (so for each line). Each time the plot above has value equal to 2 (in yellow), it means both our algorithm and the optimised algorithm has predicted the same action for the given content that the user is consuming. On the other hand, if the value is 1 (and therefore there are two values equal to 1 for this line), it means the algorithms will recommend something different (which happens 2 times out of 30). It means our algorithm has similarity with a classic optimized approach (note that this is probably not exactly the same because of convergence issues).

3.4.5 Time of convergence

Criteria of convergence As explained above, the criteria to say whether the agent has finished to learn or not is either a maximum number of episode or the maximum difference between q-tables over an episode. The last criteria should decrease when the epochs increase :

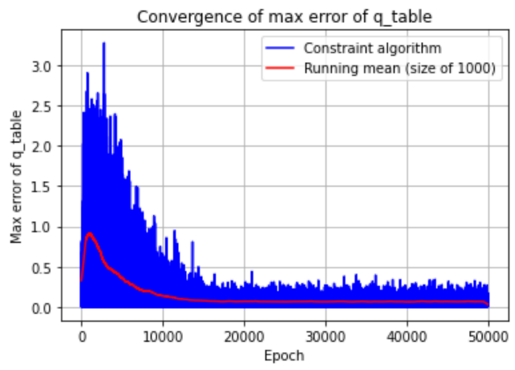


Figure 7: Evolution of the maximum difference of q-table ($\gamma = 0.9$)

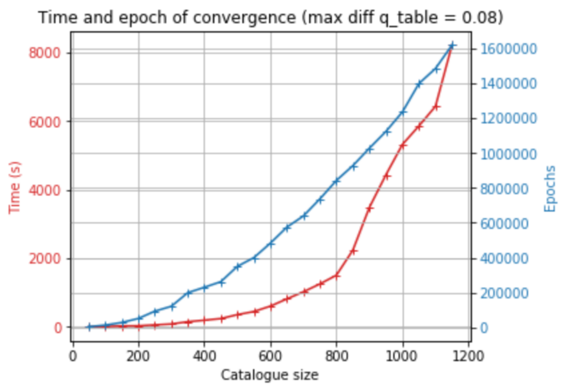


Figure 8: Time and epochs for convergence depending on the catalogue size

Therefore, we decided to take as a threshold 0.08.

Time of convergence What becomes interesting is the time of convergence of the algorithms. Indeed, here the catalogue size is 50. But in reality, the catalogue size can be million of contents. To have an idea of the time required, we looped over the size of the catalogue and we stored the time required to converge (for $\gamma = 0.9$). We set the size of the related contents to 5% of the catalogue size and the number of cached contents to 4 % of the catalogue size. We increased the size of the catalogue and reported the number of episodes and the time until convergence.

Therefore, for a catalogue size of 1200, it took 2 hours and around 1 600 000 episodes until convergence. Furthermore, if we want it to be realistic, we need to add the time of consumption of a content by the user. Indeed, for a music, it would require 3 minutes in average, and for videos it will be even more. So, for musics, it will require 400 hours until convergence, and for only 1200 songs. Thus, we need a way that requires less time to compute in order to improve the quality of the user's experience.

3.5 Deep Q-learning

3.5.1 Function approximation

As we want to represent a real data set of contents, we need to have a large amount of states in the catalogue, and therefore in the q-table too. To do so, the function approximation seems appropriate. It will use a neural network to approach the Q values. It takes as input a state and produces the q values for every action. The aim of the neural network is to learn the parameters that will lead to generate the q-tables.

We denote the new q-table as : $Q(S_t, A_t, w)$ in order to have a notation where we know that this is an approximation with neural network.

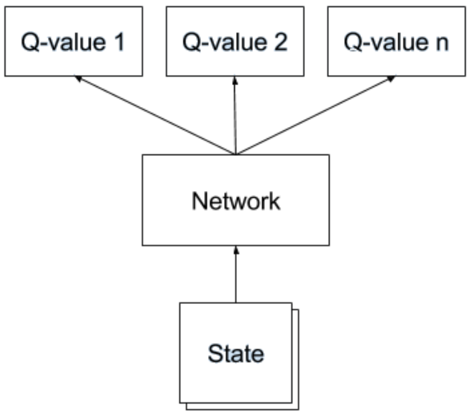


Figure 9: Function approximation summary

3.5.2 Training part

To train the neural network, it requires a target such as in supervised learning. Indeed, the parameters of the neural network can be update thanks to the gradient of the loss according to these parameters.

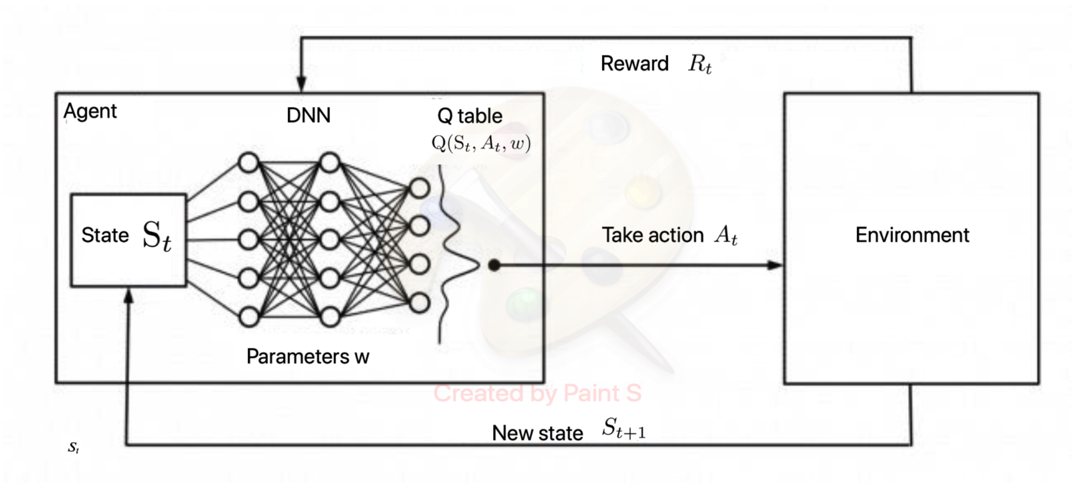


Figure 10: Summary of deep q learning for reinforcement learning

Target What we want to approximate is the Q-table, that is to say the expected rewards from a state for each action available. Therefore, as for the Q learning algorithm, the target is the following :

$$R(S_t, A_t) + \gamma \max_a Q(S_{t+1}, a, w)$$

At the beginning of the training, the next maximum reward predicted by the model will be wrong (the model isn't trained yet). The only thing that is sure is the actual reward R_t, which should slowly adjust the network parameters.

Loss To update the weights, we need a loss. As the problem is a regression problem, we can use the Mean Square Error between the target and the actual output of our network. It gives the following formula :

$$loss = (R(S_t, A_t) + \gamma \max_a Q(S_{t+1}, a, w) - Q(S_t, A_t, w))^2$$

Note that the q-table for the prediction of the target and for the actual prediction is the same. There exists a version where we use two different neurons : one for the target and one for the actual prediction. This is the double q learning algorithm. Here we use the simple q-learning algorithm.

Memory replay To build a batch that isn't correlated, what is usually done is the experience replay (or memory replay). This is done by storing each experience (which corresponds to an episode) : the State, Action, Reward, Next state and if the experience ended after this or not. In our case, if the experience had ended it was because of the user leaves the experience.

3.5.3 State representation

For the classic q-learning algorithm, we represent a content as a number. For instance, the content 15 was represented by an integer '15'. Nevertheless, we can't do this anymore. Indeed, if we were use this representation, the neural network would make the same amount of epochs than the q-learning algorithm. The aim of using function approximation is to let the neural network approximate the states, and therefore to find similarities between states that are 'close' in terms of actions to suggest. To do so, we need to change the representation of the states.

One hot encoding For a given state $S_t = i$, we replace this state by a list of contents where there is 0 everywhere except for the i position :

$$\Phi(S_t = i) = [0, ..., 1, ...0]$$

U matrix For the state $S_t = i$, we replace this state by the corresponding line of the U matrix :

$$\Phi(S_t = i) = u_i s$$

As the U matrix has values (that correspond to similarity between two contents) between 0 and 1, the algorithm would have to learn that a content that has U values equal to 0.88 or 0.76 will lead to same rewards. So another way to represent it better would be to make 1 whenever the U values isn't null and 0 otherwise :

$$\Phi(S_t = i) = (\delta_{i,j})_{1 \leq j \leq n} \text{ where } \delta_{i,j} \begin{cases} 1 & \text{if } u_{i,j} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

Rewards For the state $S_t = i$, we replace by a list of contents where there is 0 if the content is neither related nor cached, +1 if it is either related or cached and +2 if the content is both related and cached :

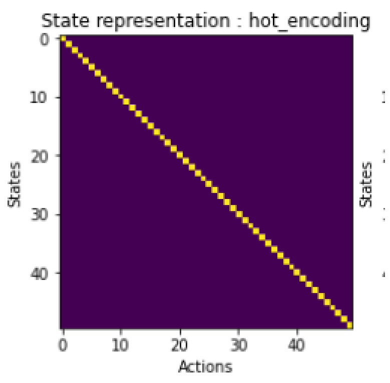
$$\Phi(S_t = i) = (\delta_{i,j})_{1 \leq j \leq n} \text{ where } \delta_{i,j} \begin{cases} 2 & \text{if j is cached and related} \\ 1 & \text{if j is cached and not related or related and not cached} \\ 0 & \text{otherwise} \end{cases}$$

Valuable Finally, to help even more the algorithm to find the q-table values, we can try to add information about what the following action would lead to. We replace the state $S_t = i$ by $\delta_{i,j}$ where :

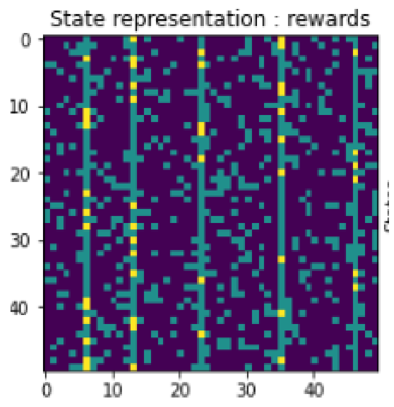
1. $\delta_{i,j} = 0$ if $u_{i,j} = 0, x_j = 1, \forall k \in [1, K], u_{j,k} \neq 0 \wedge x_k = 1$
2. $\delta_{i,j} += 1$ if $u_{i,j} \neq 0$
3. $\delta_{i,j} += 1$ if $x_j = 0$
4. $\delta_{i,j} += 1$ if $\exists k \in [1, K], u_{j,k} \neq 0 \wedge x_k = 0$

In other words, we replace the state by actions where we add 1 whenever the action is either cached, related, or if the next state it leads to has at least one related content that is cached.

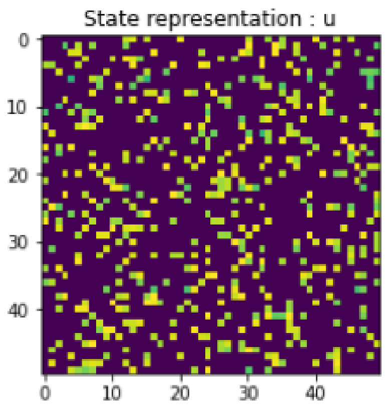
To visualise the previous representations, Figure 11 is a plot of the representation. For each line of these matrices there is the new representation of the given state.



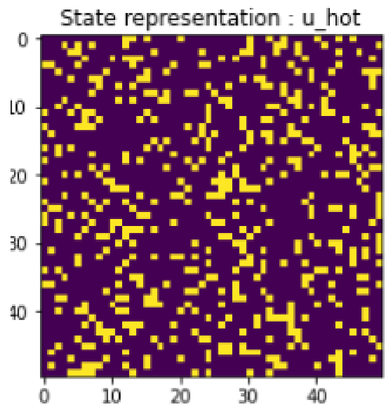
(a) One hot encoding



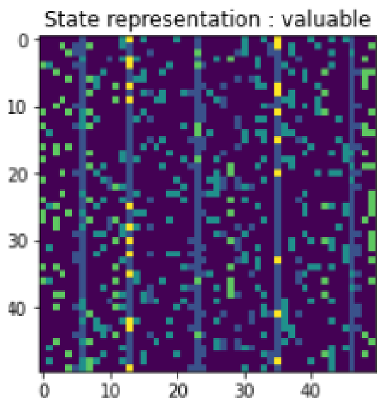
(b) Rewards



(c) U matrix



(d) U one hot encoded matrix

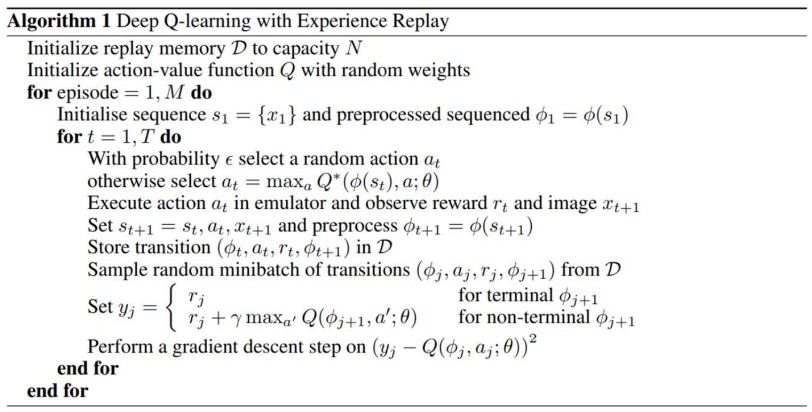


(e) Valuable

Figure 11: Different state representations for deep q learning algorithm

3.5.4 Algorithm

To summarize, we used the following algorithm to get the q-table for our problem
:



```
Algorithm 1 Deep Q-learning with Experience Replay
Initialize replay memory $\mathcal{D}$ to capacity $N$
Initialize action-value function $Q$ with random weights
for episode $= 1, M$ do
    Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
    for $t = 1, T$ do
        With probability $\epsilon$ select a random action $a_t$
        otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
        Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
    end for
end for
```

Figure 12: Deep Q learning pseudo code

3.5.5 Results

TO DO

References

[1] Reinforcement Learning I: Introduction, Richard S. Sutton and Andrew G. Barto, 1998.