Semester Project Spring 2020

“Cache-Friendly Recommender with reinforcement learning”

TABLE OF CONTENTS:

1. Introduction

The project aims at defining the cache-friendly problem seen from the Reinforcement Learning (RL) perspective. The recommendation system needs to accomplish its goal and recommend to the user not only the contents of their interest as also the content acceptable by the network, in terms of caching cost. So that the final implementation should be beneficial to both the user and the network.

Concerning the implementation in Python, we have the environment with the learning agent using the different types of user behavior and Q-learning approaches.

2. Reinforcement Learning:

Reinforcement learning allows an agent to automatically determine the ideal behavior within a specific context in order to maximize its performance. Reward feedback is required for the agent to learn its behaviour. Markov decision process (MDP) consists of the following features:

- Set of possible states(S). Every state where the agent can be in.
- Set of possible actions (A). Action that can be taken being in a state S
- A real value reward function R(S,A). Reward for being in state S
- Model. Transition T where being in a state S and taking an action A takes us to state S'. The probability of reaching the state S' if action A is taken in state S.
- Policy. Solution to MDP that indicates the action A to be taken in from a state S

2.1 Modelisation:

We have a discrete but possibly infinite set of steps : a new step is defined when the user (or customer) of a service (e.g Youtube) chooses an item (e.g video, song…). The user can choose this item from recommendations of the service, or not. This can go on for an undetermined amount of steps.

Environnement :

The environment is composed of :

- Items : A list of objects (it could be songs in Spotify, or videos in Youtube…). We denote those objects by the denomination items (one video = one item).

Those items have several properties, such as a cost -it can be binary (0 if the item is cached, 1 if not), or real (>=0).
We also added the similarities between all of the items, as another feature of those items. Those similarities are randomly generated in our code, but could be computed in a straightforward way in the case of real datasets (like Youtube).
Additional features might very likely be added, for those realistic datasets.
In our code, we created a minimalistic debug dataset of python objects with costs and similarities only.

- A user (the customer of the service (e.g Spotify customers) : a good Q learning model should make very few assumptions on the user, as the behaviour of the user is an entire part of the environnement the RL agent will have to deal with. In fact, a good RL agent should adapt to the behaviour of the user.
  To test our model and theory, and train the agent, we have implemented "simplified" versions of a potential user.
  It would also therefore be very interesting to try training the agent with a real person (a real human being), to test our model, when we begin to have a solid model and implementation.

  We report on our 2 most interesting users in the paragraph below:

  1) Type 'similar': Chooses an item in the recommendations if it has a "good enough" quality score with the currently consumed item. Here, the quality is estimated by the users by the similarity score, that has to be above a given minimum score. Else, chooses a random item, not in the recommendations.
     Ex: let's say the "minimum acceptable score" of the user is 0.8. It means that :
     - If there is a recommended item that shares a similarity score with the currently chosen item above 0.8, the user will choose this item in the recommendations. The algorithm does a for loop along the recommended items - which means that the first item along the index with the required similarity is chosen, even if other items recommended later have a higher similarity.
     - If there is not (all of the recommended items have, with the currently consumed item, a similarity score below 0.8), then the user chooses randomly an item in the item list.

  2) Type "SimilarWithSubset" :This user is an improved version of our 'Similar' user, scalable to bigger catalogs, and also more "realistic", as the algorithm is more complex. The sum of the similarities of the recommendation items with the item the customer has just chosen has to be higher than a certain value, for the recommended items to be considered by the user. To be precise:
     - If the sum of similarities is high enough, the customer chooses an item of the recommendation, randomly. The random choice is weighted by probabilities, for each of the recommended items. We took the similarities of the recommended items, and applied a softmax to the similarities to convert them into probabilities.

    - If the sum is not high enough. A random subset of the items is drawn and the customer chooses, with a uniform probability, one of the 3 items of the subset with the best similarity with the currently consumed item. We made sure in our code that the customer does not consume the same item two times in a row.

- Recommendations : this defines only the spot where N videos are put when they are recommended. The choice of these N videos is up to the RL agent. The Recommendations object alone has no link with the decision algorithm at all, it is just the subset of items that the user sees each time, when he/she has to make another choice. This is the only part of the environnement the RL agent can change/influence directly, by choosing what videos to recommend

Agent :

When defining the environnement, we used the term RL agent. It's task is to choose which items to recommend, so that the user can see them in recommendations. The final goal will be to maximise a reward, taking into account a certain state we are in at the moment. Let's define it more :

- State : The state will allow the agent to make a decision. After the user chooses a new item, this is when the state is computed, before the computation of any recommendation (at each new step). It will always take into account the current set of items (with costs and properties). We have decided to make a scalable code, M last memory assumption, where M is an input parameter:
  → M last Memory assumption: The list of the M last chosen items by the user is kept, and only those last M items. This seems like the best option, as we can avoid space / complexity issues of the Full Memory assumption (the whole previous trajectory is kept), but also improve the model by taking advantage of the series (from the perspective of markov chains). We will have to do parameter tuning to choose the right *M*.
- Reward : The agent will get a new reward at each step. As stated in the paper, the agent should be rewarded for two main reasons : resource fetching cost (it will be simplified by "item cost" in our model), and recommendation quality (in terms of content quality). The latter is trickiest, we might have several ways of defining it.
  The two aspects of the reward are:
  →Fetching Cost : The chosen item has a low cost (e.g : it is cached). Subtracting the cost of the chosen item at each step, which means negative rewards, should be sufficient to get the agent to choose low cost items. This part of the reward is quite straightforward.
  → Quality with trust : We used this idea to implement an estimate of quality.
  The idea is to give a positive reward if the user picks something in the recommendations at the new step (which means that he "trusts" the recommendations.
  Indeed, the user will more likely trust the recommendations, if the quality is high.

If the recommender adapts the quality of the recommended videos, and gets the user to look at the recommendations, it is great, because we have more control on the fetching cost, if the user picks a recommended item (whereas we have no control at all, if the user picks directly an item in the list of items). It also helps better learn choice patterns that have nothing to do with similarity, in case we want to adapt the Q learning algorithms and user behavior.

Based on those two previous aspects to take into account in the reward, we mainly did our test with the “trust” reward system. The “trust” reward system is defined by:

$$reward = \alpha T - \beta C$$

C is the caching cost of the item. T stands for “trust” and is equal to 1 if the user picked an item from the recommender (and thus trusted the recommender), and to 0 if not. β and α are hyperparameters, set to 1 and 1 for the moment.Therefore, “trust” is an implicit way to measure content quality. Indeed, some of our users, and especially the one of type “similar”, picks an item in the recommendations only if a minimum quality is satisfied. So both the cost and the content quality are taken in account in the reward.

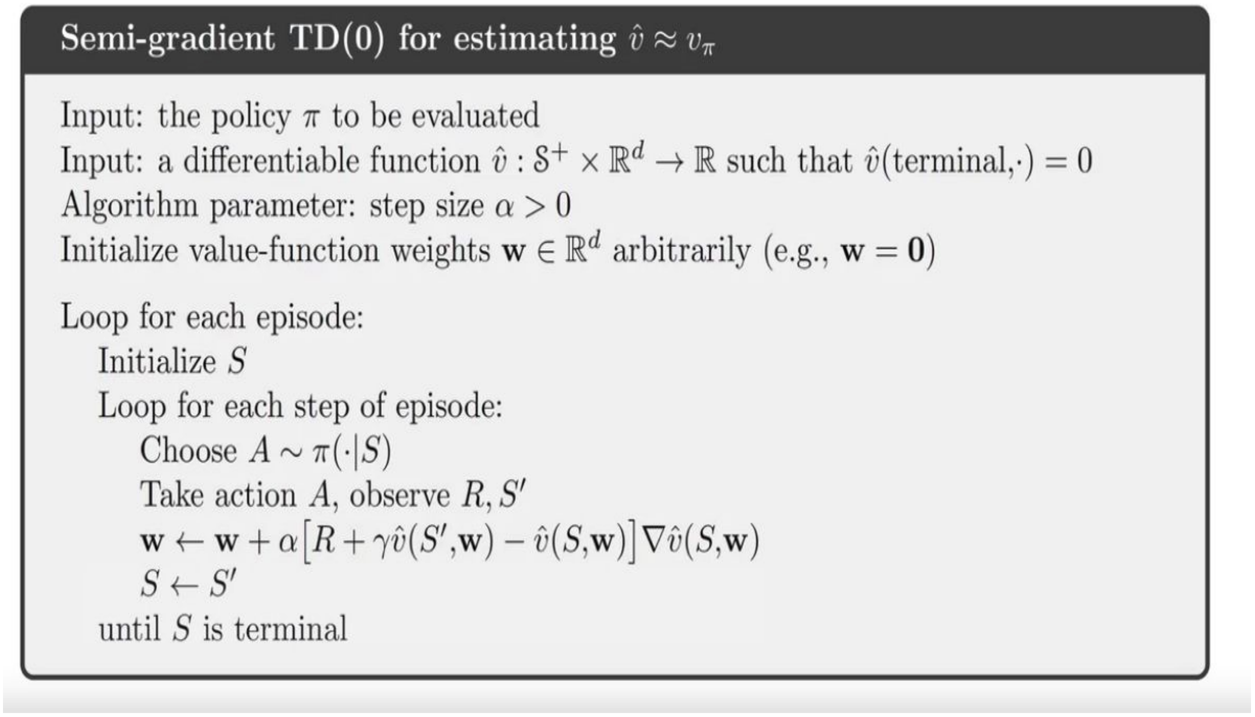
2.2 Q learning algorithms

We did several algorithms for our RL agent. We created several classes, for a few different tabular approaches and function approximation approaches. The function approaches are successfully passing the basic sanity check tests, provided that a correct input is provided.

One major problem with our implementations is that we are making huge time consuming loops.
To tackle this issue, we began a little bit to work on improving two of them into something more scalable to huge catalogs. Our first try was by preselecting a defined subset for each loop.
We also had the time for a small speed up, by keeping in memory for a few steps the value of some states - to avoid some for loops-, in the deep learning based function approximated algorithm.

Basic algorithm

All of our Q learning classes follow a semi-gradient TD learning process (with a state-action value function instead of a state-value function).
This picture shows the main principle of the algorithm :



```
Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
   Initialize $S$
   Loop for each step of episode:
      Choose $A \sim \pi(\cdot|S)$
      Take action $A$, observe $R, S'$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
      $S \leftarrow S'$
   until $S$ is terminal
```

(picture from *Reinforcement Learning - An Introduction - second edition - Richard S. Sutton and Andrew G. Barto*)

Input features of the state-action-value models

Considering the function approximation, we created several Q-learning algorithms that required an update of the input data structure for successful operation.

Input features update: The input features of the state-action feeded to the state-action value model is represented differently than at the beginning. If the earlier tabular versions, only an array of the ids of the items were taken into account. Indeed, it did not work back then, and the results were very random, because the ids of the items do not represent anything logical about the item in itself, it was just indexing.

Now the state that we use to train our model consists of the following parameters:

- costs of items of the state
- costs of items of the actions
- similarity between the currently consumed item and the items of the action

Concrete example of the new input features: if we have a state memory of 3 and a number of items to recommend of 2, the input will have the following shape :
[cs_1, cs_2, cs_3, ca_1, ca_2, s_1, s_2], with:

- cs_i is the cost of the item i in the state array (cached or not cached)
- ca_i is the cost of the item i in the action array
- s_i is the similarity score between the last item of the state array (which is the currently consumed item), and the item i in the action array

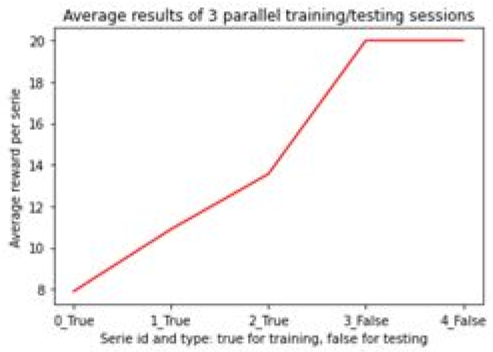
As both the costs and the similarities are accessible by the agent, it is a plausible input type. It is also quite going along with user behaviour types such as similar, as both cost and similarities are taken into account. It is also more convenient than one hot encoding, as it is a kind of input that can easily be used in larger datasets. Also, contrary to one hot encoding of item ids as an input, the architecture with this input type is usable in case of changing datasets such as Youtube or spotify.

However, it also has some drawbacks. Indeed, this input is not usable in the case of the ‘specific’ user behaviour. However, this is not the main user behaviour that we want to use. It is indeed not very realistic, we would prefer to use the ‘similar’ user, and improved versions of this user.

Function approximation Q-learning algorithms list :

The list of the Q-learning with function approximation classes, with tests, is given below:

1) Linear Q-learning: Approximation of a linear function. It takes input and multiplies it by the weight matrix. At the output, we get an estimation of the state-action value.



Input parameters:

- Number of items: 4

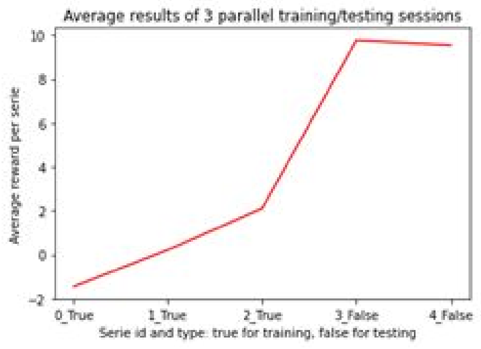
- Number of recommended items: 1

- State memory: 1

X-axis: Steps of the training/testing process.

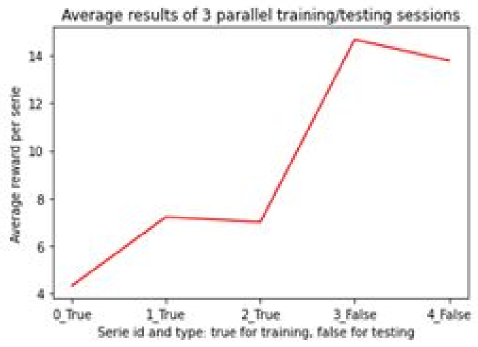
Y-axis: Average reward of 3 parallel agents. Parallel agents are agents that go through the same learning process with the same parameters.

2) Simple Deep Q-learning: One hidden layer neural network, where the size of the hidden layer can be modified.



Input parameters:

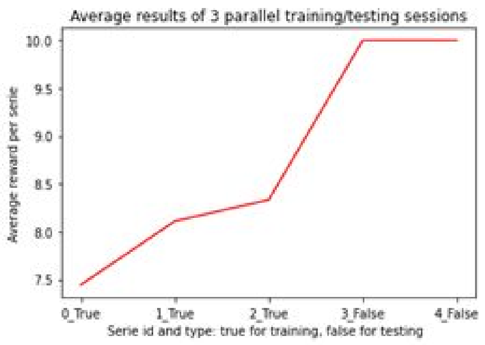
- Number of items: 4
- Number of recommended items: 1
- State memory: 1
- Hidden Layer Size: 10



Input parameters:

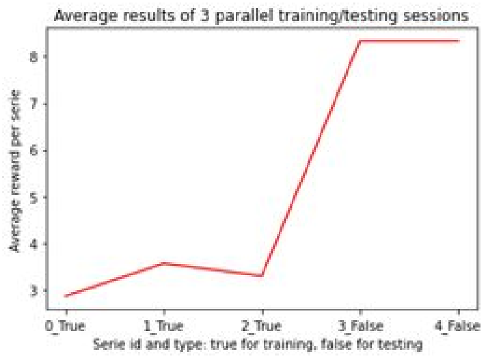
- Number of items: 50
- Number of recommended items: 1
- State memory: 2
- Hidden Layer Size: 10

3) Deep Q-learning: Full Neural Network model with no restriction on the number of layers while the list of the layers is specified in the input parameters. The class is implemented using the 'torch.nn' module.



Input parameters:

- Number of items: 4
- Number of recommended items: 1
- State memory: 1

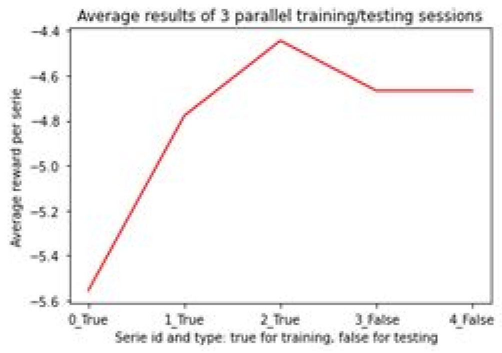


Input parameters:

- Number of items: 100
- Number of recommended items: 1
- State memory: 1

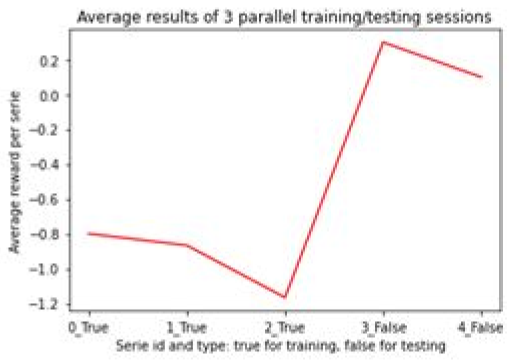
4) Fast Deep Q-learning: The main issue with our previous Qlearning classes is that we are making some very large loops, with all of the possible actions. This way, we can get the best possible action in each situation. However, this is not scalable at all to huge catalogs. Searching through all of the youtube videos for the best one would be very expensive to compute. Furthermore, what we really want is not that much to have the best possible action at each time step, but rather to get a "good enough" action. This is why we began implementing the DeepQlearningFaster class. It's structure is similar to the Deep Q-Learning class.

However, to speed up the process, instead of looking for the best action from a million items, we will look for the best action out of a predefined subset of items .We had no time to develop and test this idea better. One major drawback was that the results might end up being too approximated in the end (convergence issues).



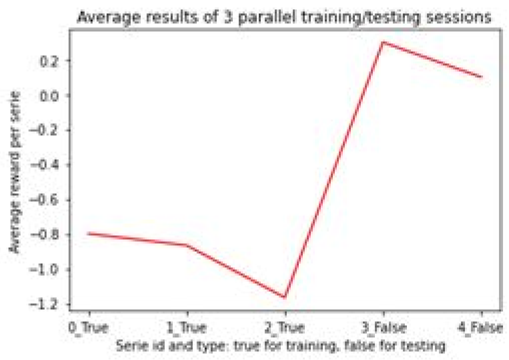
Input parameters:

- Number of items: 4
- Number of recommended items: 2



Input parameters:

- Number of items: 100
- Number of recommended items: 1
- State memory: 1



Input parameters:

- Number of items: 1000
- Number of recommended items: 1
- State memory: 1

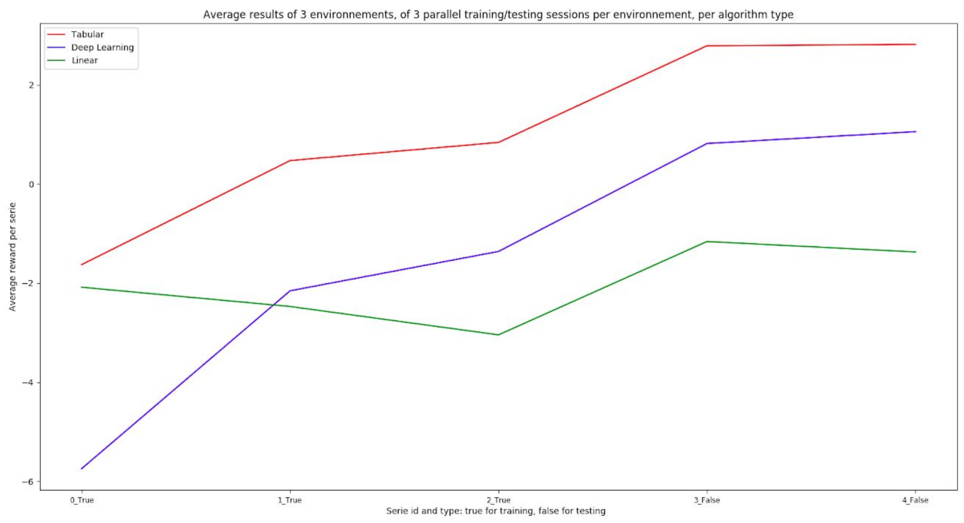
2.3 Comparison graphs

We did several tests with state memory equal to 1 and number of items to recommend also equal to 1.
The idea was to compare the performances of our most relevant algorithms on the exact same environnement -in terms of number of training epochs versus average reward-.
We used the similarWithSubset user to plot those graphs.
To get some more statistically scalable results, what we did was the average of the results of 3 instances of the following process :

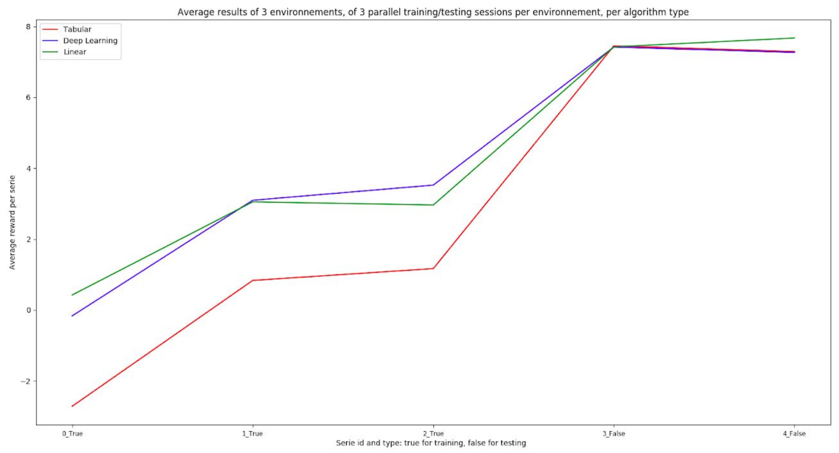
- *Create an environnement*
- *For each algorithm type:*
    - *Create 3 parallel agent, train them after an hyperparameter grid search , average the results among them*

We therefore have an "average" among different environnements - to get more scalable results.
Also, all of the environnements had the same probability for an item to be cached, and the user behavior was the same.

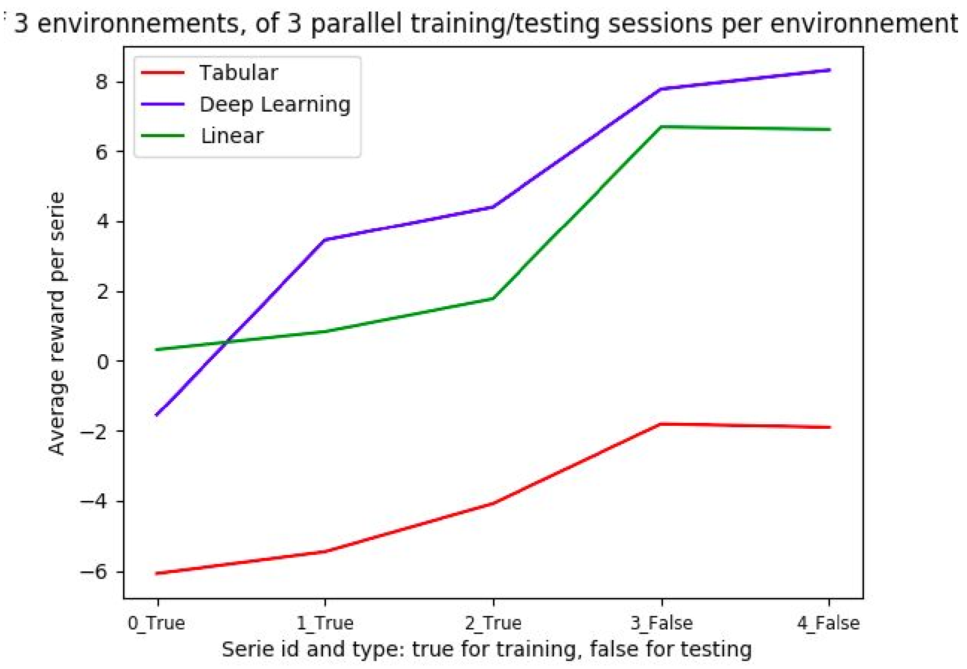
a) Catalog size: 10 items
With a small catalog of items, the tabular version grasps more rapidly the best policy, whereas the other algorithms seem not as efficient



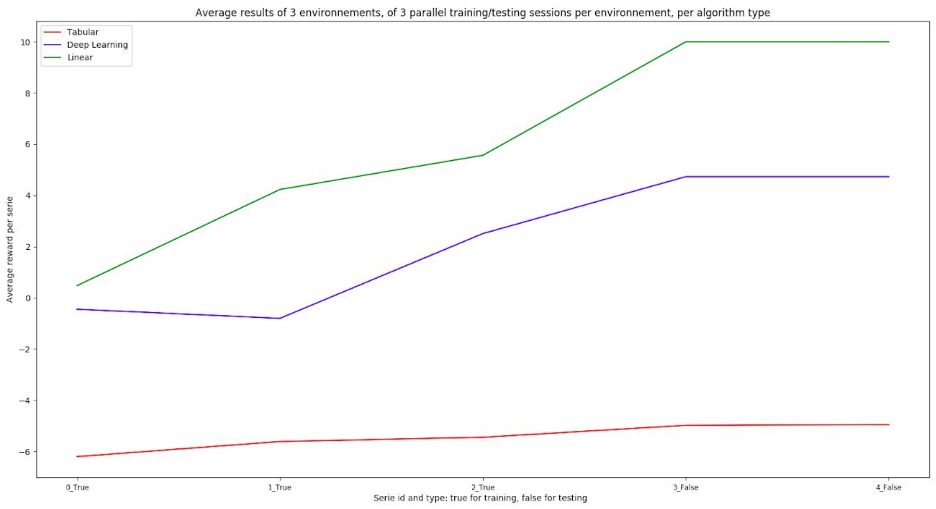
b) Catalog size: 20 items
With a small but bigger catalog of items, the tabular version does not outperform as easily as before the function approximations. Indeed, it needs more time to explore enough the space, whereas the function approximation versions are based on transformed input features, and not the item id, therefore the learning process needs less time to “explore”, with bigger catalogs.



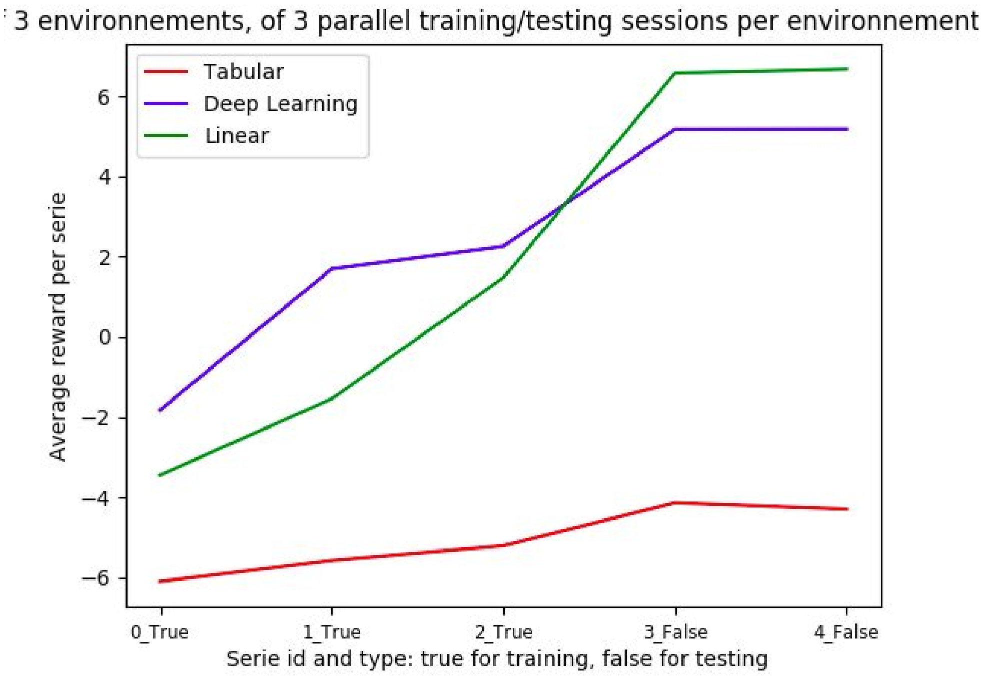
c) Catalog size: 100 items
With a bigger catalog, the Tabular version is inefficient.
The Deep and Linear versions seem to be roughly equivalent at this level, considering the simple input features. We tried to make some statistically scalable enough results, but we do not think that it is enough to draw some conclusions to differentiate Linear and Deep Learning algorithm.



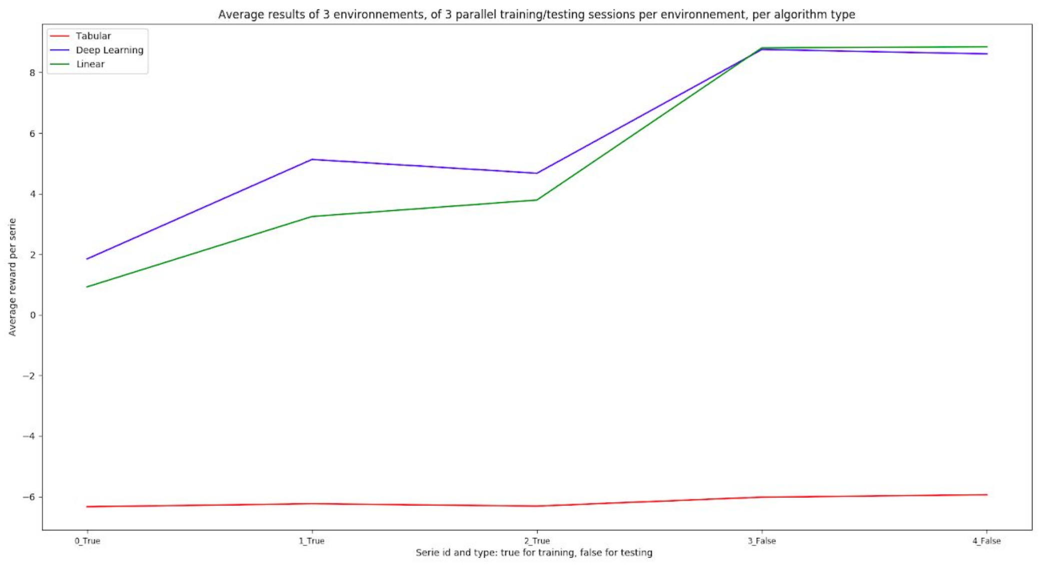
d) Catalog size: 200 items - smaller number of training epochs
The architecture of the deep learning version might not be the most adapted, as the linear version is achieving better performances. It would have been interesting to plot similar graphs but on the performances of different deep learning architectures



e) Catalog size: 200 items - bigger number of training epochs



f) Catalog size: 300 items
With an acceptable number of training epochs, on large catalogs, function approximation outperforms the tabular version.
Deep learning and Linear approximation appear to be roughly the same, it is hard to draw conclusions from all of those graphs.



3. Conclusion

In this project, we studied the problem of cache-friendly recommendation systems in the context of Reinforcement Learning. We introduced the two types of user behavior in the content services and implemented several Q-learning algorithms in order to see the performance. Our project implementation and performed research showed that the function approximation methods such as Deep Q-Learning are undeniably faster than the tabular case, in big enough catalogs, and that we managed to learn something valuable by having meaningful input features.

The main drawback of our project is the computation time. We are using many for loops, we thought about changing those for loops in order to get matrix multiplications and more python-efficient computation such as max. We would have to improve the whole structure of our agent. But we had no time to make those changes (we just made a little boost recently).
In fact, the main cause for those for loops is that we went for algorithms using state-action value networks, instead of networks that directly output the action or action probability (instead of the value).

The main reason for this choice is that it is convenient when the output of a state-action network has no need for post processing, that it is directly the action.For example, in a very basic video game where the action can be LEFT button or RIGHT button, it is quite simple to implement the state-action network. For example, the output could be an array of size 2, with the probabilities of choosing LEFT and RIGHT. However, when the action space is very large, such as in our recommender setting, this becomes more complicated, less convenient, and too space consuming, to output all probabilities for all actions.

And, as the indexes of items have not really a meaning in regards of the nature of the item, a model outputting directly the ids of the items in the action array seems not very plausible.
Also, we thought maybe of a network directly outputting the needed representative features, with features being for example similar to the input features we had chosen for our function approximation models. But it would be useful only if there was a bijective function to directly map the output feature array to the item ids of the action tuple, without having to do again loops to find the corresponding items. And also, if there was a guarantee that there indeed is always an action tuple corresponding to the required output features array.We had no time to explore this track any further, and also no time to improve the main drawback of our DeepQlearningFast class, which was supposed to solve this issue, but was too random.

Our project (with our current input features) has for the moment not yet gotten better than the level of the greedy algorithm (looking only one step beyond), even though we implemented our code so that the state memory could be a scalable parameter. The input features would have to be improved in order to get a strategy that looks several steps ahead (markovian user).

Furthermore, there are many aspects that could be developed to improve our code. Double Deep Q learning (with also target network), DynaQ/ DynaQ+ (for planning, and also interesting to use in a changing environment), replay buffer…Also, we used PyTorch but did not use already existing RL libraries, such as the already implemented Keras RL agents.
As a side note, very recently, Deepmind just launched a new framework to make distributed RL agents for research more easily, Acme https://deepmind.com/research/publications/Acme It could be interesting to use this new framework and adapt the best ideas to it, rather than staying with scratch python and torch, for efficiency.