Reinforcement Learning based Recommendation System

(Practical Project)

Rohit Thakur

26 Aug 2022

Github Repository: https://github.com/thakur-ro/udacity-datascientist

1 Introduction

In this era, a massive volume of information is available to the users through web which leads to information overload. The Recommender systems are used to facilitate the search through this vast space of items by giving user personalised services and items. The vast majority of traditional recommendation systems consider the recommendation procedure as a static process and make recommendations following a fixed strategy. A user interacts with recommendation engine in a sequence of exchanges of recommendations and provides feedback on them. Hence, we should also try to incorporate the feedback of the user at each time step while recommending items at the next time step. The idea is that previous interaction influence the later ones and the importance of the sequence of interactions can be modeled using Markov decision processes and solved by reinforcement learning.
Various methods are quite popular in Recommendation systems domain like Collaborative filtering, matrix factorization, factorization machines etc.
All these methods require daily retraining of large user-item space and provides static recommendations. We aim to develop a RL policy that will change the recommended lists dynamically based on users' recent interaction with the item space.

2 Problem Statement

To formulate the recommendation system as Markov Decision Process, we consider the Recommendation System as an Agent. The user interacts with the agent via environment, and obtains a reward and the new state. Our objective is to learn an Optimal Policy function using Deep Deterministic Policy Gradient Method which can maximize cumulative rewards.

3 Methodology

3.1 Data Ingestion and Preprocessing:

To create the RL based recommendation system, we decided to use the Amazon Electronics Reviews Dataset, as it had all the required features like user-item rating dataset, timestamps of reviews as well as user and item metadata.
Our initial dataset had around 2 million unique items and 20 million user-item interactions(ratings) and around 5 million unique users. In order to accomodate our RL model with limited time and compute resources, we sampled according to the following specifications:
First, we reduced the item space by selecting top unique 15000 items to get maximum user-item interactions. Cleaning the dataset included removing null, duplicate and erroneous ratings with duplicate timestamps. Once we obtained clean dataset, we sorted the user-item interactions by timestamp, to create sequential user history for environment creation.
Since we had very large discrete item set, we decided to generate continuous vector representation of each item. For this, we used item name and item description as well as performed some initial text cleaning. We used pre-trained weights of Language representation model to generate fixed size-50 vector representation of each unique item.
Now, every action is a vector of shape (50, 1), and every state is a concatenated array of vectors representing last N items in user history with shape (50, N).

3.2 Generating custom Gym Environment:

As we did not have any previously implemented environment for recommendation systems, we decided to generate our custom gym environment for this task. Formally, the definitions of the MDP environment are:

- State Space: A list of N last interacted items before time t.

$$s_t = \{s_t^1, s_t^2, \ldots s_t^N\}$$

- Action Space: Action in the context of recommendation is the item to recommend at time t. Hence, the action space is equal to the total number of unique products available.

- Reward R: The recommendation agent executes action a_t and state s_t, and obtains the reward r_t in terms of product rating. Hence, the product rating can be binary or can be available rating options [1 to 5].

- Transition function: The transitions are deterministic. At state s_t, we take action a_t. Our next state is:

$$s_{t+1} = s_t + a_t - s_t^0$$

We remove the first item in the state s_t to keep the state size fixed for further implementations.

- Discount Factor: $\gamma : \gamma \in [0, 1]$ defines the discount factor when we measure the present value of future reward.

3.2.1 Reward Simulation[3]:

The key component of environment was creating reward simulator. In this problem setting, the rewards are stochastic i.e we cannot specify a fixed reward value of rating in the step function as each user can give different rating to recommended item. However, when given a state and action pair, the environment is supposed to give a fixed reward. To avoid this problem, we can take inspiration of collaborative filtering method and we can generate most probable reward(rating) for given (state, action) pair using the existing data.
To get the reward value, First, we build the simulator memory M, which contains all the (state, action, reward) triplets generated from training data. Then, we group these triplets based on the reward value (total 5 groups). For each group, we calculate average state and action vectors $\bar{s}$ and $\bar{a}$. Then for every environment step i.e for any given (state, action) pair, the reward can be assigned using the minimum cosine distance of current (state, action) pair with all 5 average $(\bar{s}, \bar{a})$ pair. This also takes inspiration from classic recommendation systems where we follow intuition that users with similar interests will rate the items in similar fashion. Simulator memory is updated by the agent at every 10% of training steps.

3.3 DDPG Agent:

Deep Deterministic Policy Gradient is an algorithm which concurrently learns a q function and a policy. The reason behind choosing DDPG algorithm for agent training is that in recommendation environment setting , we have very high number of unique states and actions. In simple Q-learning methods, we find the argmax over every (state, action) pair at each state to learn the optimal policy. However, taking maximum over such large discrete action space can quickly increase the time complexity of algorithm and requirement of compute resources for convergence.
Hence, to tackle this issue, we decided to convert discrete action space into continuous using vector representations. This allows us to setup an efficient, gradient based learning rule for a policy i.e. DDPG Algorithm.
DDPG is an model-free, off-policy actor-critic algorithm which combines Deterministic Policy Gradient with Deep Q-learning. Here, the actor network updates the policy distribution in the direction suggested by the critic network. Our Actor-Critic models total 4 neural networks as Actor, Critic, Actor-target and Critic-target.

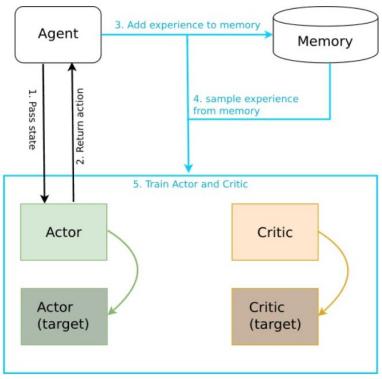

Figure 1: Deep Deterministic Policy Gradient Architecture

Soft Update: The target networks are employed for better explorations. DDPG does soft updates on the parameters of both target networks with $\tau << 1$.

$$\theta' \leftarrow \tau\theta + (1-\tau)\theta'$$

Where θ and θ' are the weights of original network and target networks respectively. This allows target network to update slowly compared to DQN design.

Actor Update: The policy $\mu_\theta(s)$ can be defined as

$$J(\Theta) = E_s[Q(s, \mu_\theta(s))]$$

Using the gradient ascent on above loss function w.r.t θ, we can find the optimal θ.

Critic Update: For loss function, we use **Mean Squared Bellman Error (MSBE)**. The target value is evaluated using target network

$$y = r + \gamma Q(s', \mu_\theta(s'|\theta^{\mu'})|\theta^{Q'})$$

Experience Replay: We continually update a replay memory table of transitions as steps are performed. The actor and critic networks are trained on random mini-batches of transitions from replay memory to reduce the correlation between the samples and more efficient use of previous experience.

4 Experiments and Results:

We ran multiple experiments over various hyperparameters such as state size, learning rate of actor and critic networks, sample size per user, soft update parameter τ, batch size, number of steps and discount factor.
Out of those experiments, below hyperparameters gave us the most significant results.

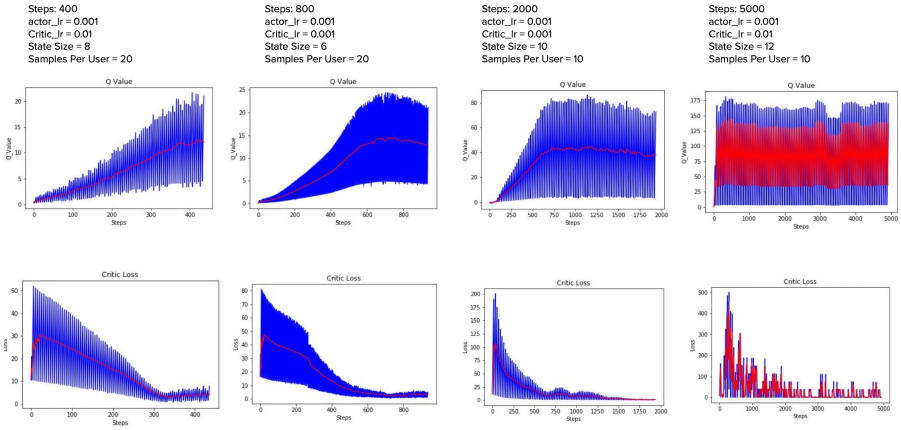

Figure 2: Deep Deterministic Policy Gradient Architecture

In above figure, 1st row indicates the q-values and their rolling average with x axis showing number of steps and q-values on Y-axis. The 2nd row shows critic loss on Y-axis and number of steps on X-axis. The details about the hyperparameters are shown above each column From the above plots, we can see that our model quickly learns the optimal q value and then stabilises after around 800 steps. The figure on the extreme right shows that model quickly learns and then overfits making the loss fluctuate at very high variance. This could be because of high number of steps and higher learning rates. Hence, we decided to reduce number of steps and learning rates.

After reducing the number of steps, the q-value starts showing definitive patterns, which can be seen from the first three columns. We can also verify steep decrease in critic loss for those experiments. We found out from multiple such experiments that 800 steps, with 20 samples per user, 0.0001 as actor critic loss and state size of 6 provides us the best possible Q-values. We kept gamma and tau constant at 0.99 and 0.001 respectively.

We believe that better performance at smaller state size is due to the fact that smaller state size allows us to obtain more number of unique (state, action) pairs from the training dataset. Interestingly, this will also allow us to address cold start issues in recommendation engine where the new user does not have substantial interaction history.

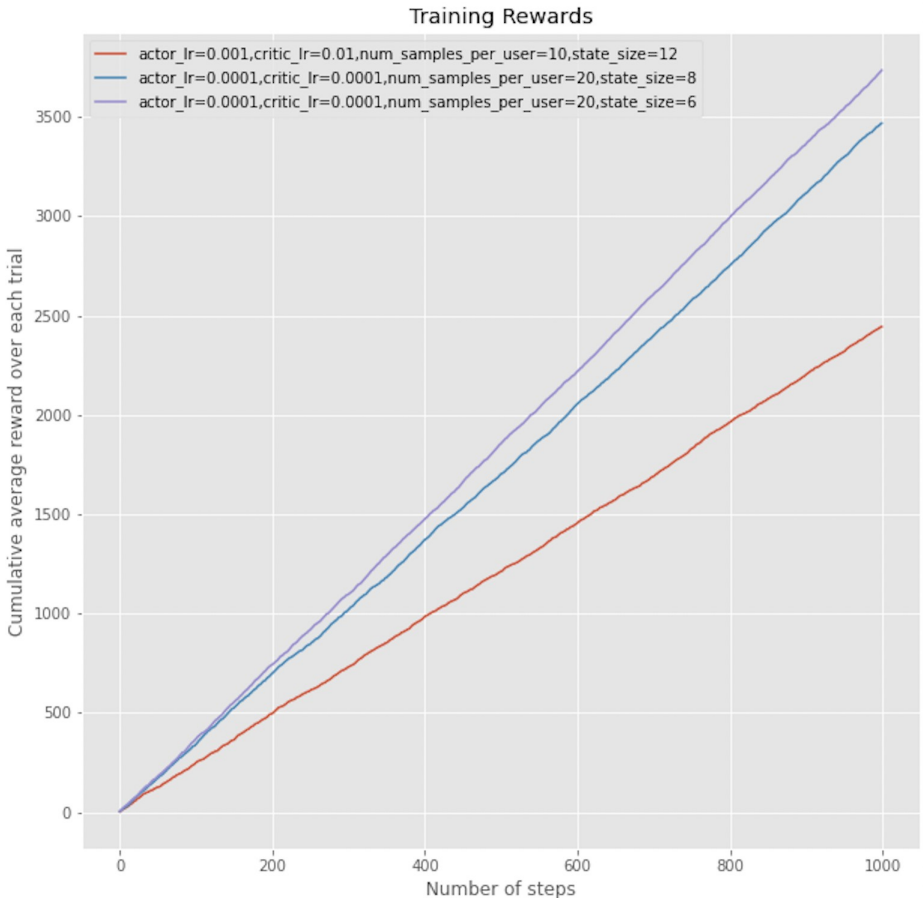

Figure 3: Deep Deterministic Policy Gradient Architecture

The above plot shows us the reward learning curves over three different experiments. The reward values are simulated ratings at each step, within set [1, 2, 3, 4, 5]. As there is no negative reward at any step, the above learning curves are only increasing with very low variance, and does not accumulate at certain optimal reward stage due to continuous nature of the environment (no terminal state).
Hence, we do not get much useful information about the model performance from the reward plot, thus, we decided to use q-value plots to monitor the performance of the agent. However, the best set of hyperparameters in above plot indicates the maximum cumulative reward in our experiments.

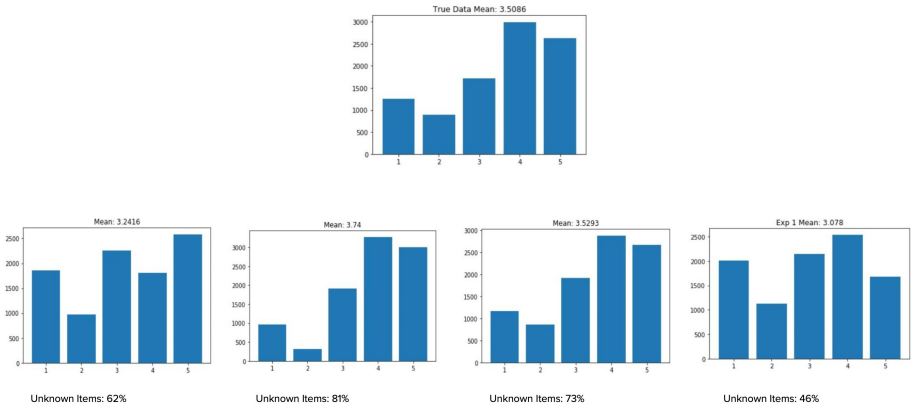

Figure 4: Deep Deterministic Policy Gradient Architecture

To verify if our learned optimal policy has learned the user's aggregated behaviour, we took the predictions/recommendations from actor network using test states. The ideal policy should give us the mean ratings over recommended products at least as good as original test ratings. In above plots we are showing ratings distributions over four experiments to compare with actual distribution. The mean simulated ratings from the second experiment were 5.71% higher than the mean actual ratings. This indicates our policy was able to provide better recommendations based on user's context.
We also calculated the diversity of recommended items in terms whether the item was already present in user's history. The diversity of recommendation is important to help user explore more products, which in turn could potentially generate higher revenue. Our best experiment had around 81% of newly recommended items which shows higher diversity in recommendations.
One thing to note here is that the true model evaluation can only be performed accurately using online A/B testing in production environment, however our results can also be interpreted as baseline offline results.

5 Conclusion and Future steps

In this project, we have studied the application of reinforcement learning algorithms to the task of recommendations. The main highlights of the project were casting the recommendation engine in MDP framework for reinforcement learning, creating a scalable custom recommendation gym environment and experiment with policy gradient method. This allows us to extend complex Reinforcement Learning algorithms to meaningful business problems.
Even though these results are promising there is still a vast opportunity for improvements. First, we can change state space representation from fixed to dynamic to tackle the cold start issue more effectively. We can also experiment with multiple reward functions rather than just positive reward values to map

user behavior, in turn yielding a better policy approximation. Lastly, learning better vector representations for state and actions can further help in increasing performance of the agent.

References

[1] M. Mehdi Afsar, Trafford Crump, and Behrouz Far
Link: Reinforcement Learning based Recommender Systems: A Survey

[2] Anton Dorozhko, Evgeniy Pavlovskiy
Link: Reinforcement learning for long-term reward optimization in recommender systems

[3]Xiangyu Zhao, Liang Zhang, Long Xia, Zhuoye Ding, Dawei Yin, Jiliang Tang
Link: Deep Reinforcement Learning for List-wise Recommendations

[4] Link: Deep Deterministic Policy Gradient Documentation

[5] Link: Amazon Reviews Dataset