Reinforcement Learning

Network Friendly Recommendations

Phase II

Konstantinos Mylonas
TUC
9 September 2023

Introduction

The goal of this assignment is to use reinforcement learning in order to recommend tuples of N videos to a user who is watches one out of K videos at each time. We want to recommend videos that are cached and relevant at the same time, so that we minimise the network load and keep the user satisfied. In a previous assignment we solved this problem with Policy Iteration and Q-learning. But these methods are not solving the problem efficiently for large K and N, so in this assignment we will try to tackle the problem with Policy Gradient method.

User Model

The user is free to choose any next video (transit to any next state) from the catalog. He can either choose one of the recommended videos or navigate through the search bar. If at least one of the recommended videos is irrelevant to the current video, then the user ignores the recommendations and navigates through the search bar with probability 1. If all videos in the recommendation tuple are relevant to s, then the user chooses one of the recommended videos with probability a or navigates through the search bar with probability 1-a. Relevance is given from a relevance matrix U where element $u_{ij} \in [0,1]$ shows how relevant is content j to content i. We define a threshold u_{min} such that content j is relevant to content i iff $u_{ij} > u_{min}$. Otherwise its considered to be irrelevant. Of course all of this is true under the condition that the user doesn't end the session, which happens with probability q. Transitioning to a cached video yields a reward equal to 1. In any other case the reward that the algorithm observes is 0.

Policy Gradient - General Idea

The reason I chose to tackle this problem with policy gradient is because its action set is huge. The action set (the set of possible N-tuples) grows very fast with K and N. The main advantage of policy gradient is that it goes straight for the policy, in contrast to value based methods that first aim to calculate the value of each state-action pair and then come up with a policy. Value based methods have to maximise over all Q values for each state, in order to come up with the best action and construct the policy,. Policy gradient methods skip this step. Instead, the output of policy gradient is a probability distribution over all possible actions. The actual action is sampled from that distribution. This strategy can save us a lot of times in cases where the action set is large.

In order to implement the method I used PyTorch. The main idea of policy gradient is to estimate the gradient of the utility function with the use of sample gradients $\nabla_\theta log\pi(\alpha;\theta)$ weighted by some advantage A. Policy $\pi(\alpha;\theta)$ is in essence the neural network that takes as input a feature vector that represents the state and outputs a probability distribution over actions.

There are many choices for the advantage A, which weights each gradient. The naive approach is to weight each $\nabla_\theta log\pi(\alpha;\theta)$ by the total reward of the episode. The estimator that is used is unbiased but it is also sample based which means that we require a big amount of samples in order for the estimate to be precise. But there are some tricks that one can use in order to reduce the variance, like taking into account only the return that was observed after a specific action was (don't let the past distract you) and subtract a baseline. That's what I did on my implementation.

The baseline that I chose to use is the value of each state V(s). The value function is approximated by another neural network.

So each $\nabla_\theta log\pi(\alpha;\theta)$ is multiplied by $A = R_{t...H} - V(S_t)$ which intuitively means that if the return that was observed after taking action α at state S_t was more than the expected return, then this action performs very well and we should favour it. The quantity $R_{t...H} - V(S_t)$ is positive and because gradient $\nabla_\theta log\pi(\alpha;\theta)$ is the steepest ascent, the algorithm ends up favouring the actions that perform *better than average*.

Policy Network Architecture

The idea behind approximation methods is that the neural network (or other approximator that might be used) can be trained on some sample states and generalise its knowledge for states that it has never experienced. In order for this to happen we need to use appropriate feature vectors that capture the similarities between different states. For this problem I used as feature vector the row of U table for each state (I call it relevance row). The idea behind this choice is that if the network can find a good action (recommendations) for a video, then this action will most likely be good for any other video that has similar relevant content. So the input of the neural network is a K dimensional vector that corresponds to the relevance row of the state that the user is currently in. The NN has one hidden layer with $2 \cdot K$ neurons and the output layer has as many neurons as the possible actions (K choose N). There's nothing sophisticated with the hidden layer here. I just chose to use a small hidden layer because the network is already big and having many weights to train in the hidden layer would make the training process slower. The hidden layer has ReLU activation while the output layer softmax, because the output has to be a probability distribution.

Experiments

In practice, policy gradient worked as expected. I will first try to show that policy gradient indeed finds a right policy and then I will compare it to Q-learning.

PG in practice

In order to show that PG finds a correct solution I constructed an artificial example:

Configuration:

$$a = 1 \text{ - } u_{min} = 1e-5 \text{ - q} = 0.1 \text{ - } K = 20 \text{ - } N = 2 \text{ - } epochs = 100 \text{ - } batch\ size = 1000$$

So in this artificial example I made sure that there is always a tuple with cached and relevant items to recommend. Since the user always follows our recommendation (given that what we recommend is relevant - which I've taken care of), he should achieve an average return per episode equal to 9, which he does.

But to be more precise, I am also showing the sampled actions for each state, so its more clear that the recommendations are indeed correct.

```
C
[1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]

State 0 Action: (3, 5)
State 1 Action: (3, 5)
State 2 Action: (3, 5)
State 3 Action: (1, 2)
State 4 Action: (1, 2)
State 5 Action: (0, 4)
State 6 Action: (1, 2)
State 7 Action: (3, 5)
State 8 Action: (3, 5)
State 9 Action: (3, 5)
State 10 Action: (1, 2)
State 11 Action: (1, 2)
State 12 Action: (3, 5)
State 13 Action: (3, 5)
State 14 Action: (3, 5)
State 15 Action: (3, 5)
State 16 Action: (3, 5)
State 17 Action: (1, 2)
State 18 Action: (3, 5)
State 19 Action: (3, 5)
```

The diagonal of the relevance matrix is filled with 0 in order to avoid recommending the same video that is being watched. As we can see our algorithm always recommends cached items. Moreover there is no state in

which the algorithm will recommend the same video. This means that the algorithm learns to recommend both relevant and cached videos.

Policy Gradient vs. Q learning

The training of the NN is done in epochs. One epoch ends when a number of episodes (batch_size) is played. Then the parameters of the network are updated. The Q learning algorithm that I implemented in the previous phase of the project was trained for a number of training episodes (there was no such thing as epochs). But in order to compare the two algorithms we need to see their progress on a similar setting. So the Q learning was slightly modified and is now trained in terms of epochs and batch_size.

Configuration:

$$a = 0.9 \text{ - } u_{min} = 0.1 \text{ - q} = 0.1 \text{ - } K = 20 \text{ - } N = 2 \text{ - } epochs = 230 \text{ - } batch\ size = 1000$$

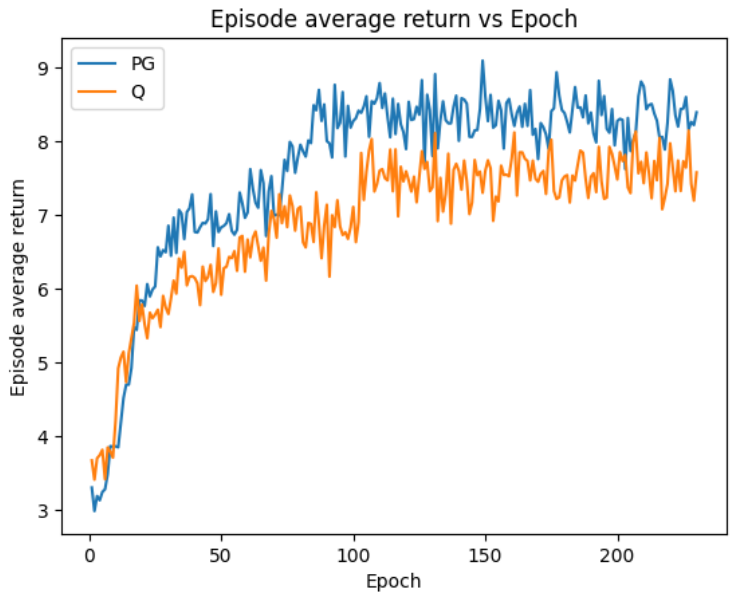

Figure 1

One thing to keep in mind about Figure 1 is that it shows the *training* returns. During training, Q learning has a fixed exploration probability (ε=0.2) so the training graph doesn't really show how good Q learning is, because 20% of the times it acts randomly. After training is completed we can get a deterministic policy that will result to a higher average return than what the Figure 1 suggests. Policy gradient has some stochasticity as well because its actions are sampled from a distribution. But the network is continuously updated so it doesn't have this "fixed" exploration that Q learning has. After many epochs its quite possible that most of the probability mass is given to the best action.

For this example, I also used policy iteration (PI) and came up with three policies, one for PI, one for PG and one for Q learning. Then I evaluated each policy for 10.000 episodes and calculated the average episode return. The result was the following:

Method	**Average reward on 10000 episodes**
PI	8.2621
PG	8.1264
Q learning	7.9132

According to the above results, all methods find the optimal policy (Q learning maybe needed some more training). We are sure that PI finds the optimal policy (we've shown that in the previous assignment) and since PG has similar performance to PI we can assume that it finds the right solution. Another informal way of thinking about it is that q=0.1 means that each episode has 10 time steps where the last one gives no reward because its a terminal state, so there are 9 time steps out of which the agent can receive a reward. If the user always follows our recommendations, then the maximum average reward per episode would be 9. But a = 0.9 means that the user

follows our recommendations 9/10 times on average. So achieving something close to 8.2 is reasonable for an optimal policy under this configuration (given that the there are at least two cached videos relevant to each state, so the optimal agent always recommends relevant and cached videos).

The evaluation of Q learning on 10.000 episodes gives an average episode return of 7.9. If one looks carefully on the training graph, the reward of Q learning is centered around 7. But when we get the resulting deterministic policy and try it for 10.000 episodes, we get something closer to 8, which confirms my earlier claim: that the training graph doesn't really show how "good" Q learning is.

Configuration:

$$a = 0.9 \text{ - } u_{min} = 0.1 \text{ - q} = 0.1 \text{ - } K = 100 \text{ - } N = 2 \text{ - } epochs = 230 \text{ - } batch\ size = 1000$$

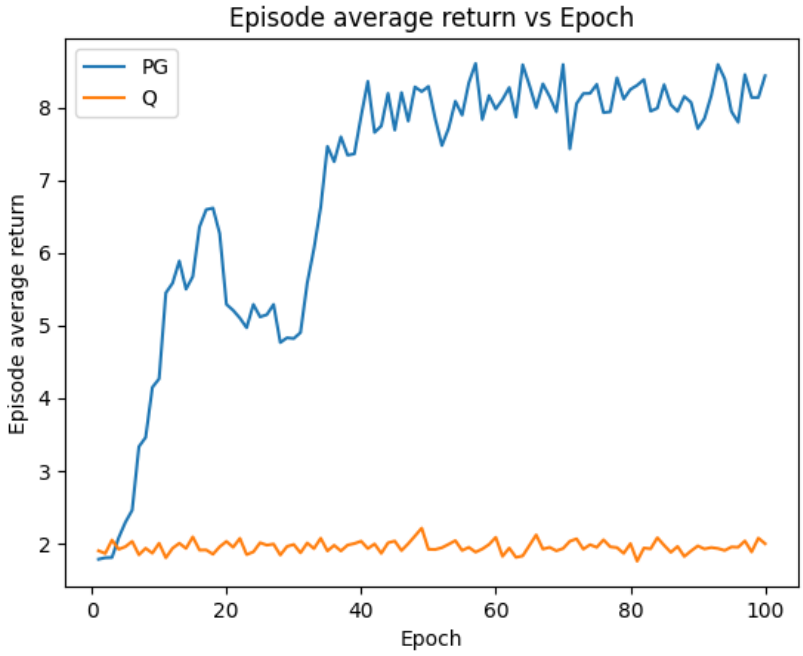

Figure 2

In this more demanding experiment, PG outperforms Q learning. Q learning completely fails to learn anything useful in 100 epochs. In contrast, Policy gradient seems to learn very fast and performs great after epoch 40 (which translates to 40 x 1000 = 40.000 episodes). In the previous phase we showed that Q learning required more than 5.000.000 episodes in order to learn an optimal policy for K=100 and N=2. This translates to 5.000 epochs of 1.000 episodes each. So in terms of required experience, PG outperforms Q learning. The 5.000.000 episodes took more than 1h to run, while the 100 epochs of PG only took 20 minutes.

Configuration:

$$a = 0.9 \text{ - } u_{min} = 0.1 \text{ - q} = 0.1 \text{ - } K = 50 \text{ - } N = 3 \text{ - } epochs = 50 \text{ - } batch\ size = 1000$$

This is an even more demanding example because the action set in this case is way larger than for K=100 - N=2.

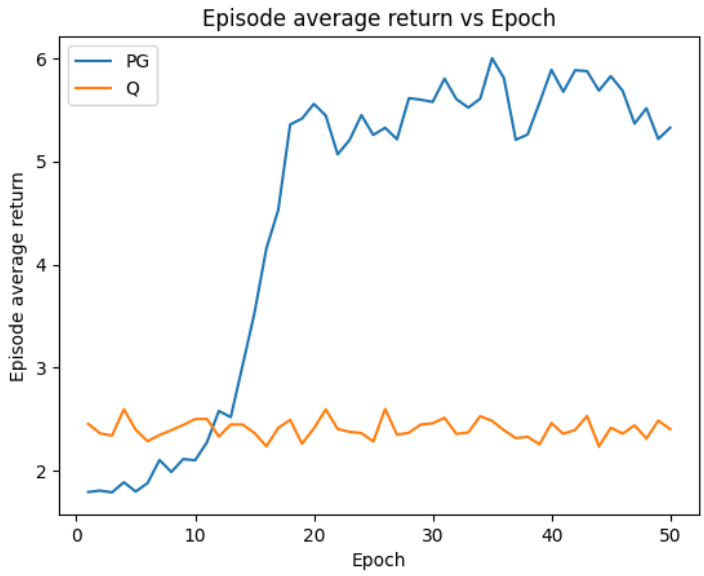

Figure 3

PG again outperforms Q learning, which fails to learn anything useful. But policy gradient seems to converge to an average return of 5.5.

I tried many experiments of K=150 - N=2 for different architectures. For example, I tried to increase the hidden layer neurons to $10 \cdot K$ neurons instead of $2 \cdot K$. I also tried to change the feature vector. Instead of just using the relevance row, I concatenated it with the cached vector. In this case the input is of size $2 \cdot K$ and includes the relevance row along with the cached vector. The algorithm always appears to converge to an average return between 5 and 6. The algorithm is still useful but I'm not quite sure if the result is the optimal (with this configuration I would expect the optimal to be close to 8.2, if there are always relevant and cached videos (but for K=150, N=2 and due to the fact that u_{min} is very small, it makes sense to assume that there should always be an optimal action to take).

Configuration:

$$a = 0.9 \text{ - } u_{min} = 0.1 \text{ - q} = 0.1 \text{ - } K = 150 \text{ - } N = 2 \text{ - } epochs = 100 \text{ - } batch\ size = 1000$$

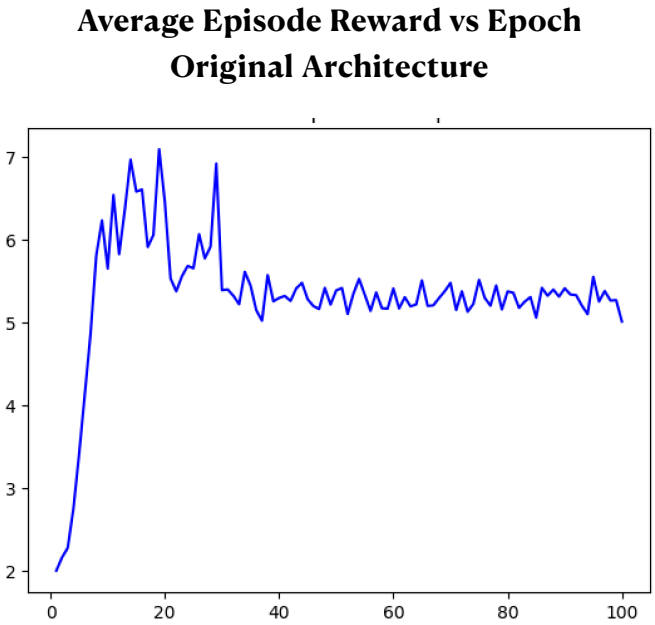

Figure 4

Figure 4 refers to PG for K=150, N=2. The input is just the relevance row. Hidden layer has $2 \cdot K$ neurons. It is the original architecture.

Conclusion

In conclusion, policy gradient performs better than Q learning. It is more effective for large K and N because it is able to generalise its knowledge over states that it has not experienced. However, PG is limited as well because as the action set becomes larger, the output layer of the policy network grows very fast, which in turn means that there are many weights that need to be trained. Forward and backward pass turn out to be pretty slow for large K and N.