Harjot Mangat

EECS 258 – Multimedia Systems

Project Report – *Tabular Dyna-Q for Adaptive Chunk Size*

Abstract

For this project, I implemented tabular Dyna-Q within a small TF Agents environment. The goal of this project was to have a simple implementation that represents adaptive chunk size for video streaming. Through the evaluations we can see that this tabular reinforcement learning algorithm can develop a policy to maximize chunk sizes for the given throughput.

Tabular Dyna-Q

Tabular Dyna-Q is a simple reinforcement learning algorithm designed to combine both learning from experience and planning from a model. *Figure 1* shows the Tabular Dyna-Q algorithm. This algorithm is an evolution over the standard Q-learning algorithm, while still retaining a similar policy building behavior. This algorithm functions similarly to Q-learning in

Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
- (a) $S \leftarrow$ current (nonterminal) state
- (b) $A \leftarrow \varepsilon$-greedy(S,Q)
- (c) Take action A; observe resultant reward, R, and state, S'
- (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$
- (e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)
- (f) Loop repeat n times:
  - $S \leftarrow$ random previously observed state
  - $A \leftarrow$ random action previously taken in S
  - $R, S' \leftarrow Model(S,A)$
  - $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Figure 1 Tabular Dyna-Q

that it begins with an empty state-action values (*Q(s,a)*) table and begins exploring from the initial state. The algorithm uses the *Q(s,a)* values to determine the action that it should take for any particular state (S). The *Q(s,a)* values are updated once the agent receives the next state (S') and any reward (R) from its actions. Up to this point, the algorithm is identical to the standard Q-learning algorithm, the evolution comes from the planning phase that begins at *step e* in *figure 1*. At this point the algorithm can learn from model planning on top of the learning it did from experience of interacting with the environment. This model planning update is similar to

the update from *step d* of the experience update. There are 3 hyperparameters for this algorithm, alpha (α), epsilon (ε), and discount (*γ)*. The alpha can control the learning rate, determining how much impact the update step has on the existing *Q(s,a)* value. The epsilon controls the ratio of exploration that the algorithm performs, this determines how often the algorithm picks a random action instead of the action recommended by *Q(s,a)*. This exploration allows the algorithm to continually try new actions even later into its learning stage. The discount controls how much importance future updates have to the current *Q(s,a)* value update. This allows the algorithm to focus on either current rewards with a low discount value, or future rewards with a high discount value.

TF Agents Environment

The environment for this project is created in TF Agents. This allows for discrete steps for the tabular RL algorithm to interact with this created environment. The environment handles the states (*S*) and rewards (*R*) based on the actions of the agent. The state consists of a buffer (*b*), a current throughput (t_c), and a future throughput (t_f). At each step, the goal of the Dyna-Q algorithm is to choose an action that will prevent the buffer (*b*) from running empty. The TF Agents environment allows the program to be run in discrete steps that aggregate into an episode. At each step, the agent is given a state denoted $s(b,t_c,t_f)$ and can choose an action $a(p_{chunk})$. The action that the agent takes is a decision on the size of the chunk to be downloaded based on the throughput of the environment. The environment will calculate the reward received from the action and calculate the next state. The reward is mainly calculated based on two events. The first event is if the buffer runs empty, in which case the reward would be -1 and the episode will end. The second event is if the p_{chunk} selected by the agent is the maximal available within t_c without going over the amount t_c can support in that step, in which case the reward is 1. The next state is calculated based on the p_{chunk} that the agent selected, if the p_{chunk} was less than or equal to t_c, the buffer is increased by p_{chunk}. If p_{chunk} is greater than t_c, the agent must wait until the p_{chunk} can be downloaded. If the agent can fill the buffer, the episode ends without any penalty. A quick summary of the state, action, reward can be seen in *Equation 1*.

- State: $s(b,t_c,t_f)$
- Action: $a(p_{chunk})$
- Reward:
  - -1 if b=0
  - 1 if $p_{chunk} = t_c$
  - Else 0

b = [0-10]
t_c = [1-4]
t_f = [1-4]
P_{chunk} = [0-4]

Equation 1 State, Action, Rewards for our Environment

Experiment

The experiment was run in Jupyter Notebook using python 3.8. The hyper parameters for the Dyna-Q algorithm were: alpha (α) = 0.5, epsilon (ε) = 0.05, discount (*γ)* = 0.9. The experiment was run for 100 episodes.

Results

The results of running this Dyna-Q agent in the environment show that Dyna-Q can learn the behavior of keeping the buffer full within 100 episodes. The rewards per episode can be seen in *figure 2*. We can see from *figure 2* that as the agent gets to interact with the environment more, the rewards it can accumulate per episode increase, showing that it is able to learn how to keep the buffer from running empty and that the agent can choose the correct

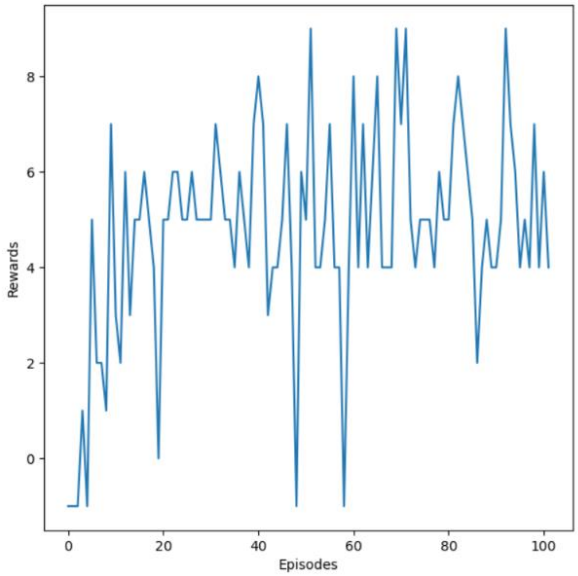

Figure 2 Rewards per Episode

p_{chunk} for the t_c. We can also look at the *Q(s,a)* tables to see the learning habits of the algorithm.

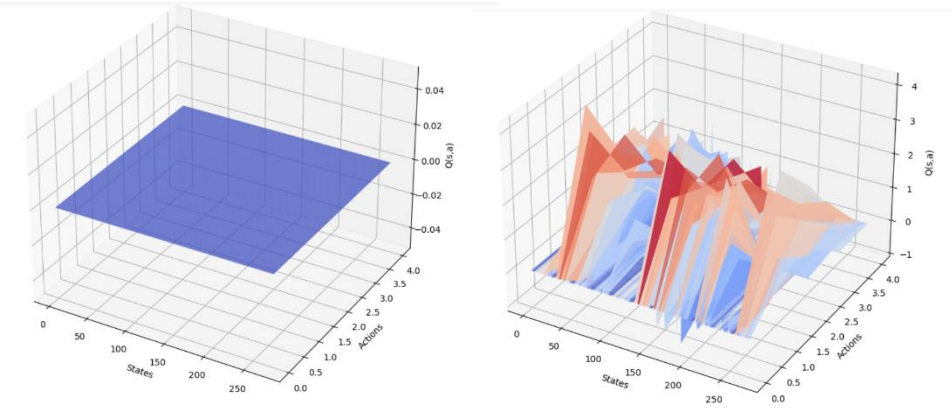

Figure 3 Q(s,a) values for our Dyna-Q agent. (Left is Episode 0) (Right is Episode 100)

Figure 3 shows the *Q(s,a)* at both Episode 0 and Episode 100. We can see that the initial *Q(s,a)* values are correctly initialized to 0 across all values. After 100 episodes we can see that the agent is learning which *Q(s,a)* have high values, represented by the high red peaks, and which *Q(s,a)* have low values, represented by low blue valleys.

Conclusion

In this project I implemented the tabular Dyna-Q learning algorithm to handle the adaptive video chunk size problem represented by my TF Agents environment. This project shows insight into the ability of the tabular Dyna-Q algorithm, but the tabular nature of this algorithm limits it from scaling to larger real-world examples because of the increasing size of the state-action space. The tabular RL algorithm was able to learn and maximize the video chunk size to match the throughput given by the environment. The results show that the Dyna-Q agent can learn a policy with less than 100 episodes of experience with the environment.