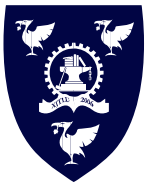

Xi'an Jiaotong-Liverpool University

西交利物浦大学

EEE-415 MULTIMEDIA COMMUNICATIONS

Lab 3: DASH Based on Deep Q-Learning

Student Name Xun Xu

Student ID 1926930

2020/06/20

Content

Introduction

The goal of Lab3's is to use the known information (past video quality, historical bandwidth, buffer level, and future bandwidth information, etc.) to adaptively predict the video rate that can be provided to users at the next moment to improve user's experience as much as possible. As shown in the figure 1.

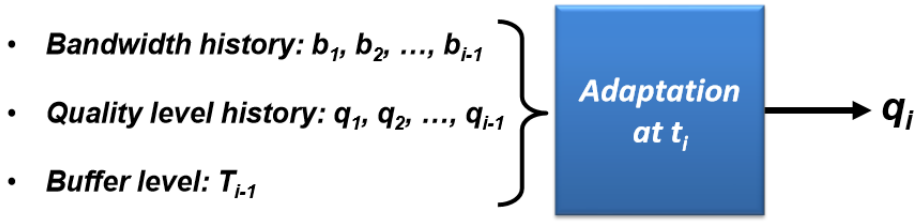

figure 1: DASH Adaptation

As can be seen from the figure 2, at the same moment, the video content provider will provide users with different quality video sequences, just like the 480p and 1080p we chose in YouTube, but due to the limitations of objective conditions such as channel bandwidth. The video rate that users can enjoy cannot exceed the limit of objective conditions, shown in the figure 3.At the same time, the switching of video quality or video buffering will affect the user's quality of service, then we need to use reasonable and efficient DASH Simulation means to maximize user experience.

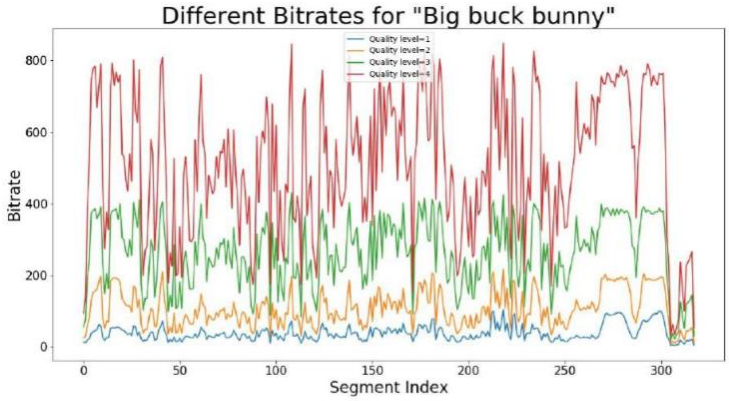

figure 2: Bitrate/segment for Different Qualit

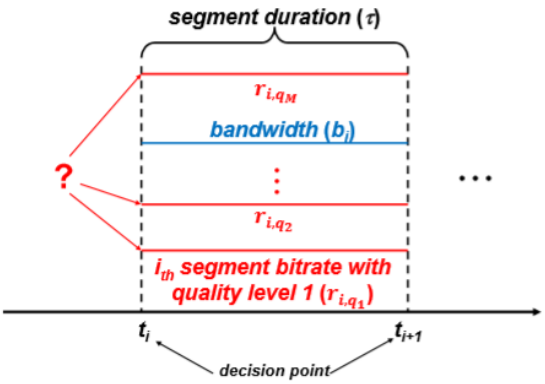

figure 3: DASH Adaptation simulation

That is to say, given the input data and the objective function (e.g., quality of experience (QoE)), a proper video quality for the current segment should be requested to maximize the objective function. And this adaptation process can be based on buffer level and future bandwidth information. The adaptation problem is formulated as an optimization process with the proposed internal QoE goal function.

Calculate the QoE

To calculate the QoE with future information, we use two algorithms. One is traditional arithmetic, and the other is based on Q-learning. Q-learning algorithm will introduce next part.

Traditional arithmetic

For traditional arithmetic, every decision point we consider each possibility quality level of current segment and future l segments comes with quality level of segments in the past to calculate QoE value. As is shown in figure 4.

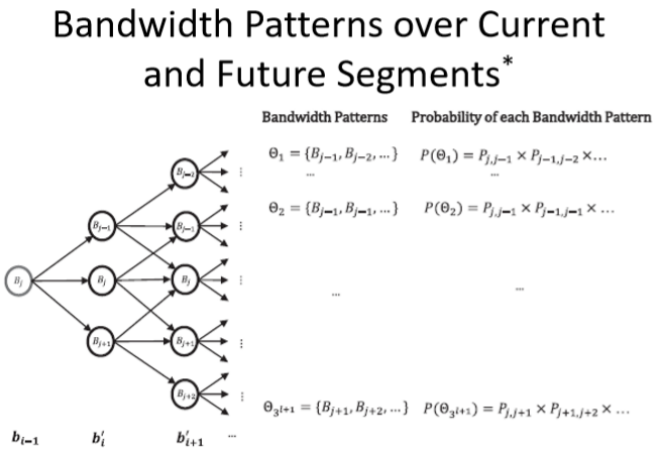

figure 4: traditional arithmetic of bandwidth prediction

For traditional arithmetic, it has disadvantages as below:

-It is assumed that the Markov channel model is known before the adaptation. This may not be the case in real systems.

• The number of bandwidth levels is limited.

• The greedy search approach for the sub optimization problem may be too complicated due to its large search space.

-It's over all possible combinations of predicted bandwidth and requested quality patterns

So we decided to use reinforcement q learning to calculate the QoE.

Reinforcement learning:

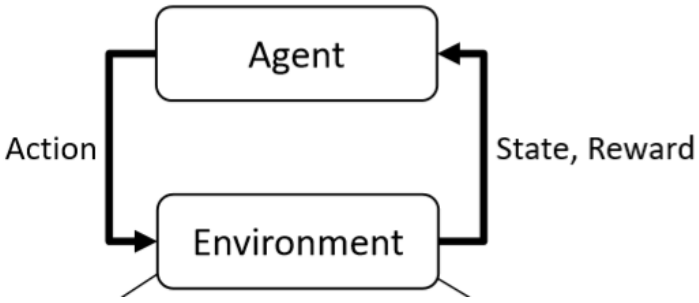

figure 5: Reinforcement Learning framework

Reinforcement learning (RL) is a machine learning technique where an agent is connected to its environment via perception and action. RL focuses on how agents can take a series of behaviors in the environment to achieve maximum cumulative returns,

which is called reward. By RL, an agent should know what state of action should be taken. RL is a study of the mapping from environmental state to action.

Reinforcement learning learns the optimal control strategy through early offline training, and then applies the strategy in real-time adaptive control, which can improve the flexibility and adaptability of the client's rate decision mechanism. The long-term utility can be formulated recursively using Bellman's equation as follows:

$$R(s_0;\Pi) = \sum_{s_1 \in S} P(s_1|s_0)\rho(s_0,s_1,\Pi(s_0)) + \lambda R(s_1;\Pi)$$

The optimal policy is obtained as follows:

$$\Pi^*(s) = \text{argmax}_{\Pi}\ R(s;\Pi)$$

Q learning

The optimal policy problem can be solved by dynamic programming, Q-learning is a model-free RL algorithm. Then the optimal policy can be obtained as:

$$\Pi^*(s) = \text{argmax}_a\ Q^*(s,a)$$

This is done by our ActionSelector function in the code. In standard Q-learning, the Q-value—i.e., the estimation of $Q*(s,a)$—is updated when the agent takes action at in state st as follows:

$$\hat{R}(s_t,a_t) = \rho(s_t,s_{t+1},a_t) + \lambda \underset{a}{max} Q(s_{t+1},a)$$

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha(\hat{R}(s_t,a_t) - Q(s_t,a_t))$$

In actual implementation, we maintain a table of Q values (Q-table) for state-action pairs initialized to arbitrary values (e.g., zeros) and update it based on the above equations. However, Q-learning has two main drawbacks:

- Continuous state space
- Curse of dimensionality

Deep Q-learning

For Q-learning cannot handle well a large state space (e.g., continuous state through quantization with a small step size). So the best compromise between representation accuracy and the number of states is often difficult to find. Then Deep Q-learning can address this issues by approximating the action-value function $Q(s,a)$

through neural networks shown in figure 6.

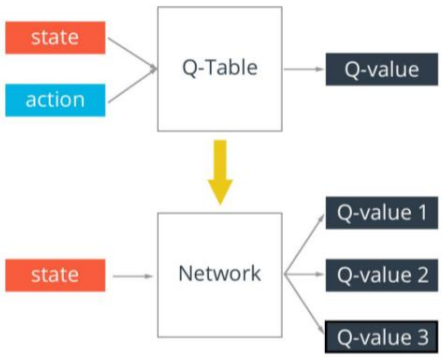

figure 6: The architecture of Q-learning and DQN

With deep Q-learning we cope with these issues by approximating the action-value function Q(s,q) through a neural network that, for any given state and action pair (s,q), returns the corresponding (estimated) Q-value Q(s,q). The network weights, once trained, will encode the mapping and replace the tables used by Q-learning. This allows the model to be fed with continuous variables, avoiding the quantization problem, and has the further desirable property that neural networks, if properly trained, are able to generalize, providing correct answers (excellent Q-value approximations) even for points (s,q) that were never processed in the training phase. In other words, neural networks act as universal approximators.

In this lab, I advocate the use of FNN(with two hidden layers) and LSTM(RNN) network to learn excellent video adaptation strategies, while compactly and effectively capturing the experience acquired from the environment. As shown in the figure 7.

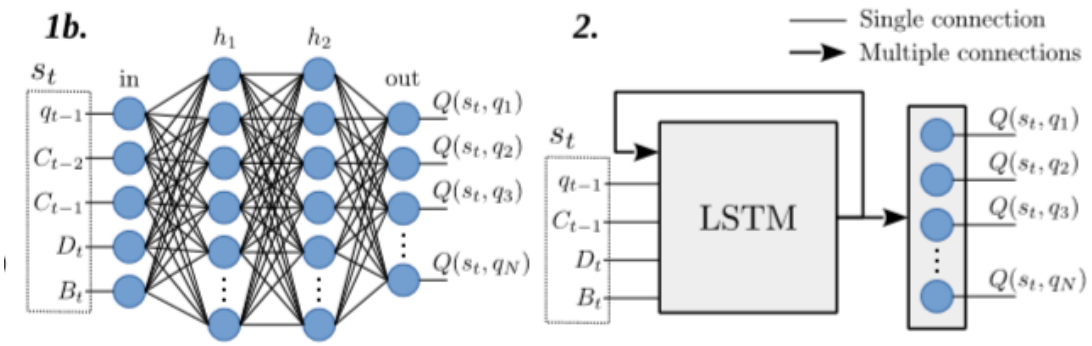

figure 7: Network architectures of FNN and LSTM

LSTM introduction

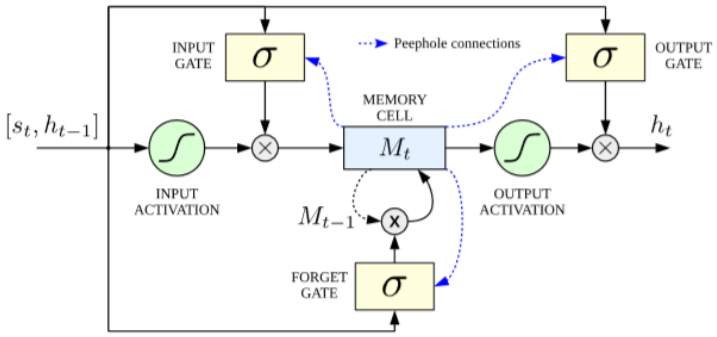

figure 8: diagram of an LSTM cell.

For LSTM, Recurrent Neural Networks (RNNs) implement some internal feedback mechanics that introduce memory, i.e., given an input vector, the network output depends on the network's weights and on the previous inputs. In this study, we implement an RNN network based on LSTM cells as sketched in figure 8. The memory is implemented through a Memory Cell that allows storing or forgetting information about past network states. This is made possible by structures called gates that handle access to the Memory Cell. Gates are composed of a cascade of a network with sigmoidal activation function (σ) and a pointwise multiplication block. There are three gates in an LSTM cell: 1. the input gate, that controls the new information that need to be stored in the Memory Cell, 2. the forget gate, that manages the information to keep in the memory and what to forget, and 3. the output gate, associated with the output of the cell (ht). In addition, all the data that pass through a gate is reshaped by an activation function. Backpropagation Through Time (BPTT) is usually used in conjunction with optimization methods to train RNNs.

The whole experiment process is shown in the figure 9, the contents of Target Network and Replay Memory will be discussed in more detail at the remaining of this report.

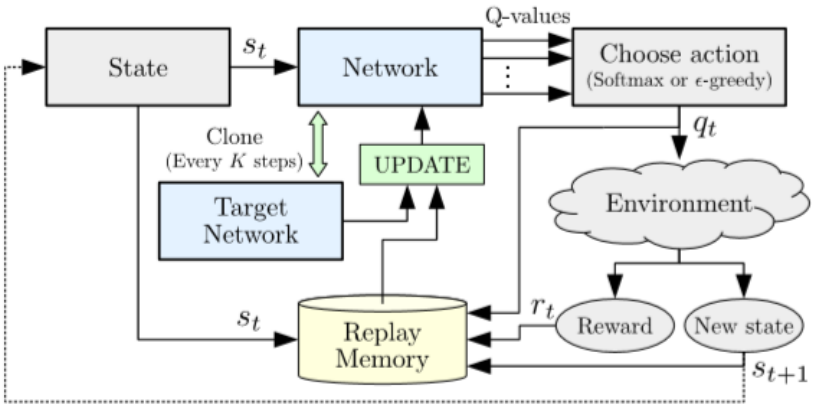

figure 9: A Deep Q-Learning Framework for DASH Video Streaming

Methodology

GPU using

At the beginning, I found that the code could not be run with the GPU. After the corresponding code was modified, cuda can be used.

Code that uses the GPU run：（add .to(device) and double at the end of some code)

```
state_batch = torch.stack([experiences[i].state.tensor().to(device)
                                        for i in range(BATCH_SIZE)])
next_state_batch = torch.stack([experiences[i].next_state.tensor().to(device)
                                            for i in range(BATCH_SIZE)])
action_batch = torch.tensor([experiences[i].action
                                        for i in range(BATCH_SIZE)],
dtype=torch.long).to(device)
reward_batch = torch.tensor([experiences[i].reward
                                        for i in range(BATCH_SIZE)],
dtype=torch.float32).to(device)
```

```
state_action_values = policy_net(state_batch).gather(1,
action_batch.view(BATCH_SIZE, -1)).double()
```

```
next_state_values = policy_net(next_state_batch).max(1)[0].detach().double()
```

State

We define the state class in the code as follows, and st = (qt-1,Ct,φt, Bt) in DASH Simulation pdf can be obtained through this four variables.

State define corresponding code:

```
class State:
    """
    $s_t = (q_{t-1}, F_{t-1}(q_{t-1}), B_t, \bm{C}_t)$, which is a modified
    version of the state defined in [1].
    """
    sg_quality: int
    sg_size: float
    buffer: float
```

```
    ch_history: np.ndarray
```

State class instantiation(initializing) corresponding code:

```
state = State(
    sg_quality=sg_quality,
    sg_size=sss[CH_HISTORY-1, sg_quality],
    buffer=T,
    ch_history=bws[0:CH_HISTORY]
)
```

State renew corresponding code:

```
sg_quality = selector.action(state)
sqs[t-CH_HISTORY] = sg_quality
# update the state
tau = sss[t, sg_quality] / bws[t]
buffer_next = T - max(0, state.buffer-tau)
next_state = State(
    sg_quality=sg_quality,
    sg_size=sss[t, sg_quality],
    buffer=buffer_next,
    ch_history=bws[t-CH_HISTORY+1:t+1]
)
```

Reward

Calculate reward according to the calculation formula of reward, the reward function is based on QoE taking into account video quality, quality variations, rebuffering events, and buffer management penalty, which is similar to the one we already studied but only for the current and previous segments.

$$\hat{R}(s_t, a_t) = \rho(s_t, s_{t+1}, a_t) + \lambda \max_a Q(s_{t+1}, a)$$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha(\hat{R}(s_t, a_t) - Q(s_t, a_t))$$

Reward calculation corresponding code:

```
downloading_time = next_state.sg_size/next_state.ch_history[-1]
rebuffering = max(0, downloading_time-state.buffer)
rewards[t-CH_HISTORY] = next_state.sg_quality \
    - BETA*abs(next_state.sg_quality-state.sg_quality) \
    - GAMMA*rebuffering - DELTA*max(0, B_THR-next_state.buffer)**2
```

Loss function

The loss function Lt at time t is evaluated using et = (St,qt, rt, St+1), which is called the agent's experience at time t, and is given by:

$$\tilde{L}_t(s_t, q_t, r_t, s_{t+1}|\boldsymbol{w}_t) = (r_t + \lambda \max_q \hat{Q}(s_{t+1}, q|\overline{\boldsymbol{w}_t}) - Q(s_t, q_t|\boldsymbol{w}_t))^2$$

where rt is the reward for segment t and is given by:

$$r_t(q_{t-1}, q_t, \phi_t, B_{t+1}) = q_t - \beta \parallel q_t - q_{t-1} \parallel - \gamma\phi_t - \delta[max(0, B_{thr} - B_{t+1})]^2$$

Loss function calculation corresponding code:

```
state_action_values =
policy_net(state_batch).gather(1,action_batch.view(BATCH_SIZE, -1))
# $\max_{q}\hat{Q}(s_{t+1},q|\bar{\bm{w}}_t$ in (13) in [1]
# TASK 2: Replace policy_net with target_net.
next_state_values = policy_net(next_state_batch).max(1)[0].detach()
# expected Q values
expected_state_action_values = reward_batch + (LAMBDA * next_state_values)
# loss fuction, i.e., (14) in [1]
mse_loss = torch.nn.MSELoss(reduction='mean')
loss = mse_loss(state_action_values,expected_state_action_values.unsqueeze(1))
```

Target network

A second neural network, referred to as the target network, is accounted to increase the stability of the learning system, and its weight vector is updated every K steps (segments), by setting it equal to that of the first network and keeping it fixed for the next K-1 steps, The target network is used to retrieve the mapping of Q values.

Definition of target network:

```
target_net = copy.deepcopy(policy_net)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()
```

Use target network to compute:

```
# TASK 2: Replace policy_net with target_net.
next_state_values = policy_net(next_state_batch).max(1)[0].detach().double()
```

Renew target network:

```
if t % TARGET_UPDATE == 0:
    target_net.load_state_dict(policy_net.state_dict())
```

Replay memory

Another improvement is given by the implementation of a technique called experience replay. The agent's experience et=(st,qt,rt,st+1) is stored into a replay memory R={ e1,...,et} after each iteration. In this way, a new loss function Lt that also accounts for the past experience can be considered. Specifically, a subset Rm={e1,...,eM} of M samples, with ej ∈ R, j=1,2,...,M, is extracted uniformly at random from the replay memory R, and Lt is finally evaluated as an empirical mean over the samples in set Rm:

$$L_t(w_t) = \frac{1}{M} \sum_{e_j \in R_M} \tilde{L}_t(e_j|w_t)$$

This leads to three main advantages:

- greater data efficiency
- uncorrelated subsequent training samples
- independence between current policy and samples

LSTM implementation

When using LSTM for training, I mainly made the following code modifications:

Change CH_HISTORY from 2 to 1, whereas recurrent networks only need the last channel capacity sample Ct−1 as input, as the feedback loop inside the LSTM cell keeps track of the

channel memory:

```
CH_HISTORY = 1          # number of channel capacity history samples
```

Set λ to 0.5 according to the reference paper as shown in figure 10:

```
LAMBDA = 0.5
```

LSTM $LSTM_{ph}$	number of units N_c	128
	learning rate	10^{-3}
	batchsize M	100
	K	200
	policy	Softmax
	λ	0.5

figure 10: adaptation algorithm parameters in paper

Modify the relevant code in the action function according to the results obtained by the LSTM network, accordingly, the output of LSTM is processed by another linear layer to provide the final Q-values:

```
if sample > eps_threshold:
    with torch.no_grad():
        out_action, (hn, cn) = policy_net(state.tensor().to(device).reshape(1, 1,
num_features))
        out1 = forwardCalculation(out_action)
        return int(torch.argmax(out1))
else:
    return random.randrange(self.num_actions)
```

```
if self.greedy_policy:
    with torch.no_grad():
        out_action, (hn, cn) = policy_net(state.tensor().to(device).reshape(1, 1,
num_features))
        out1 = forwardCalculation(out_action)
        return int(torch.argmax(out1))
```

LSTM Network parameter settings:

The parameters of the LSTM network mainly include the input dimension, the number of hidden layer units, and the number of network layers. The input dimension is consistent with the original input in the code. The hidden number is set to 16, which can be set according to our needs. I set the number of layers to two Layers to increase the depth and nonlinearity of the network:

```
hidden_dim = 128
policy_net = torch.nn.LSTM(N_I , hidden_dim, num_layers=2).to(device)
forwardCalculation = torch.nn.Linear(hidden_dim, 4).to(device)
optimizer = torch.optim.Adam(policy_net.parameters(), lr=LEARNING_RATE)
```

The setting of the number of iterations, according to my experience of running the code many times, the value of reward has already begun to converge when the number of iterations reaches more than 200, so I set the maximum iteration to 300:

```
num_episodes = 300
```

The modification of the training part code is to be compatible with the original program. The two-dimensional data is changed into the three-dimensional input of the LSTM network according to the batch size set by itself, and then enter the linear layer after inputting it to the LSTM network to obtain output q values.

```
# reshape to input the LSTM
```

```
new_state_batch = state_batch.reshape(NUM_OF_BATCH, ONE_BATCH,
num_features)
new_next_state_batch = next_state_batch.reshape(NUM_OF_BATCH, ONE_BATCH,
num_features)

# $Q(s_t, q_t|\bm{w}_t)$ in (13) in [1]
# 1. policy_net generates a batch of Q(...) for all q values.
# 2. columns of actions taken are selected using 'action_batch'.
output1, (hn, cn) = policy_net(new_state_batch)
out_of_hidden1 = forwardCalculation(output1).reshape(BATCH_SIZE,
num_features)
state_action_values = out_of_hidden1.gather(1, action_batch.view(BATCH_SIZE, -
1)).double()

# $\max_{q}\hat{Q}(s_{t+1},q|\bar{\bm{w}}_t$ in (13) in [1]
# TASK 2: Replace policy_net with target_net.
output2,(hn,cn) = policy_net(new_next_state_batch)
out_of_hidden2 = forwardCalculation(output2).reshape(BATCH_SIZE,
num_features)
next_state_values = out_of_hidden2.max(1)[0].detach().double()
```

Plot

After saving the data through the npz file in the dash simulation code, then read it in the plot code and make a picture to compare the results.

Code for saving:

```
np.savez('dash_LSTM_without_target_300', data_sqs=mean_sqs,
data_rewards=mean_rewards)
```

Code for reading and ploting

```
fnn_with_target = np.load('./data/dash_FNN_with_target_300.npz')
fnn_no_target= np.load('./data/dash_FNN_without_target_300OK.npz')
lstm_no_target = np.load('./data/dash_LSTM_without_target_300ok.npz')
lstm_with_target = np.load('./data/dash_LSTM_with_target_300ok.npz')

fig, axs = plt.subplots(nrows=2, sharex=True)
axs[0].plot(fnn_with_target['data_rewards'])
axs[0].plot(fnn_no_target['data_rewards'])
axs[0].set_ylabel("Reward")
axs[0].vlines(len(fnn_with_target['data_sqs']), *axs[0].get_ylim(), colors='red',
linestyles='dotted')
axs[1].plot(fnn_with_target['data_sqs'])
axs[1].plot(fnn_no_target['data_sqs'])
axs[1].set_ylabel("Video Quality")
```

```
axs[1].set_xlabel("Video Episode")
axs[1].vlines(len(fnn_with_target['data_sqs']), *axs[1].get_ylim(), colors='red',
linestyles='dotted')
plt.legend(['fnn_with_target','fnn_no_target'], loc='upper right')
plt.show()
plt.close('all')
```

Whole coding ideas

The whole training process is described as follows:

A schematic diagram of an update iteration is shown in figure 9. First, the current environment state st is fed to the deep neural network, which outputs an estimate of the Q-value for each possible action $q \in A$, i.e., the various representations in the adaptation set A. Then, an action qt is chosen according to either an ε-greedy. Upon taking action qt, the system moves to the new state st+1 and a new reward rt is evaluated. The newly acquired experience et = (st,qt,rt,st+1) is stored into the replay memory R. Hence, a batch of M samples is extracted, uniformly at random, from the replay memory and is used to update the network's weights through the Adam optimization method. The loss function is minimized, using the mapping from the target network, whose weights are updated every K=20 steps.

Results analysis

Results is as follows:

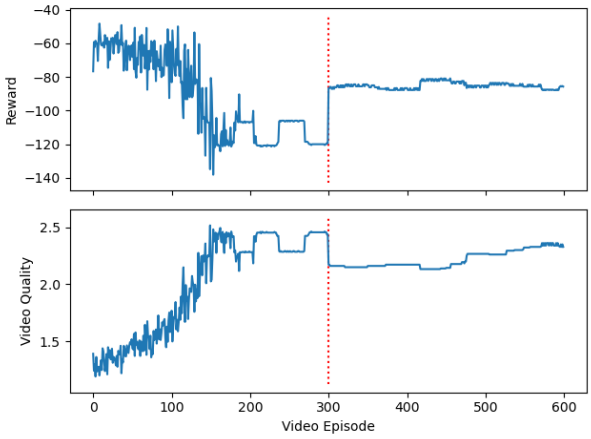

figure 11: FNN without target network

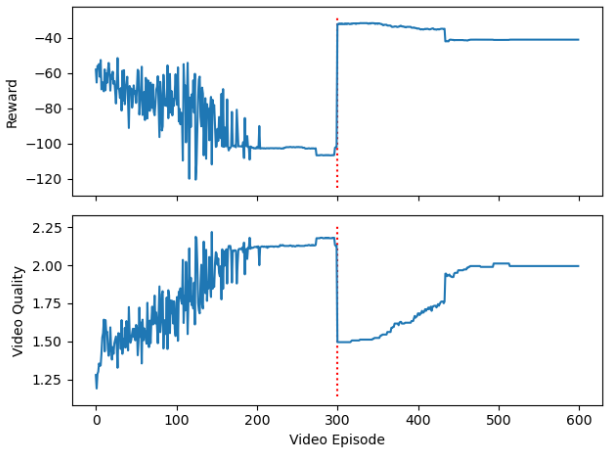

figure 12: figure 13: FNN with target network

#1 Implement the policy network based on the RNN using LSTM cells, replacing the FNN in the baseline implementation, run DASH simulation, and obtain the QoE results.

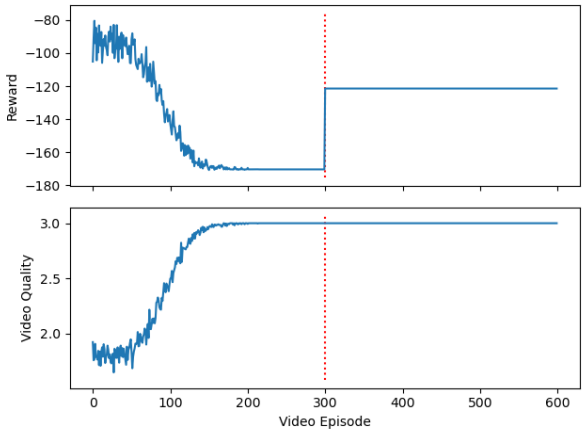

figure 14: LSTM without target network

And the code is d-dash-LSTM-without-target.py

#2 Implement the target network on top of the the results from #1, run DASH simulation, and obtain the QoE results.

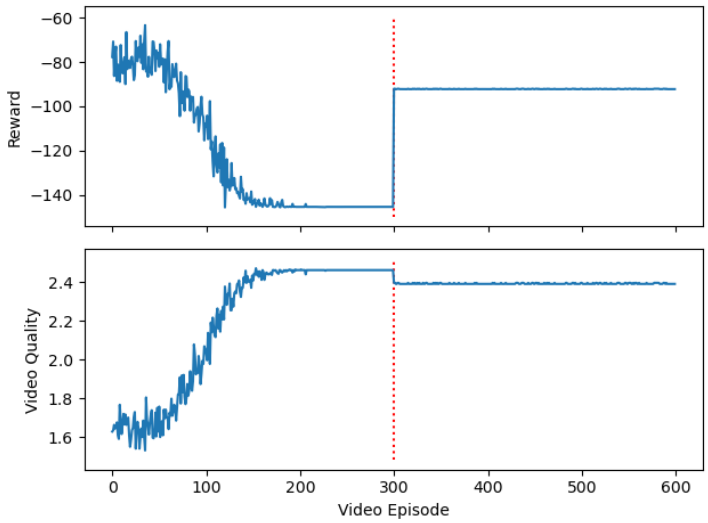

figure 15: LSTM with target network

And the code is d-dash-LSTM-with-target.py

#3 Generate a plot comparing all the results from the baseline and #1–#3. #4

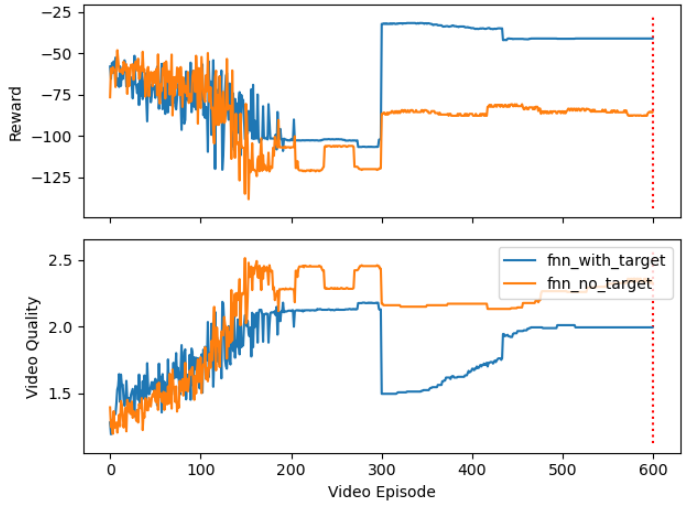

figure 16: Comparison of FNN with and without target network

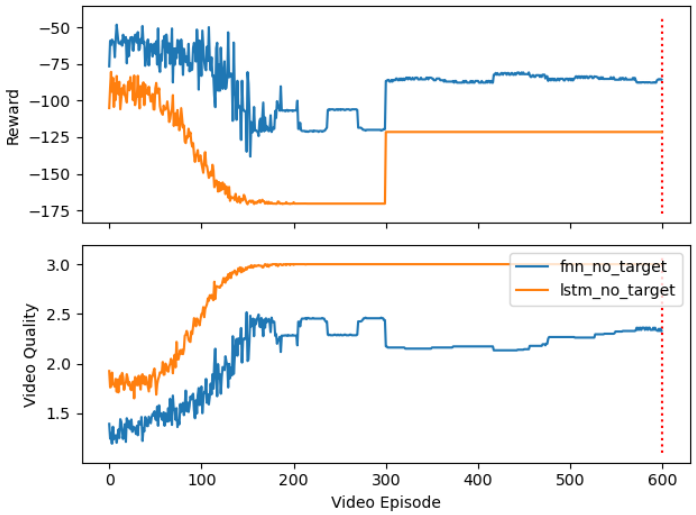

figure 17: Comparison of FNN and LSTM

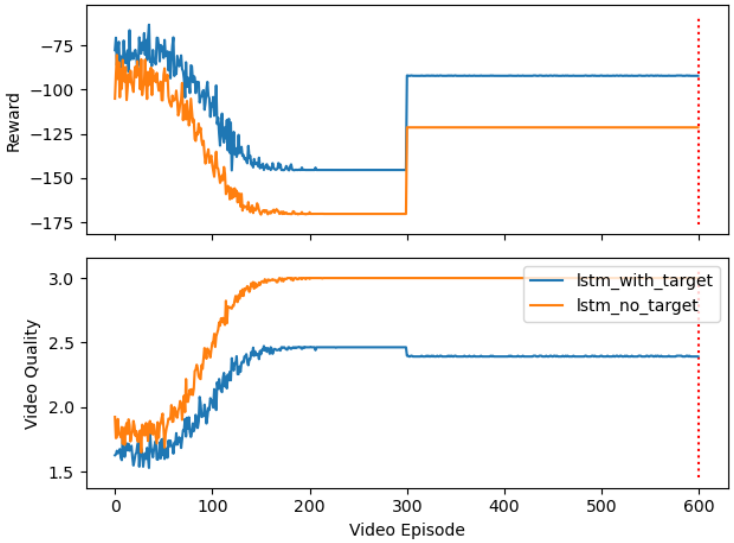

figure 18: Comparison of LSTM with and without target network

And the code is plt_of_dash.py

Discuss the following points based on the results from your analysis above:

Question 1:

Are there any advantages of using RNN with LSTM cells over FNN in terms of performance (i.e., QoE) and implementation? If so, explain them.

Answer:

Yes, LSTM cells has obvious advantages over FNN, as can be seen from Figure 17 above. In the terms of Stability, Convergence speed and Quality, LSTM is much better. LSTM gets a lower reward at convergence, taking about 140 episodes to converge as FNN needs about 180 episodes to converge. The quality of LSTM is much higher than

the FNN(3 to 2.5). And no matter in the training or testing phase, the oscillation and jitter of the FNN curve are much more obvious, that is to say, LSTM has better stability.

Reason:

LSTM algorithm proved to be very valuable on channel traces with long-term correlation. First of all, LSTM belongs to a type of RNN. Due to the particularity of its structure, RNN causes the training results in the front to affect the results of the subsequent training. This is why RNN is used to deal with a series of correlations effectively. But when dealing with long-term related sequences, RNN has the problem of gradient disappearance, and LSTM has solved this problem on the basis of using controlled with additive interactions. The ordinary FNN network does not have this advantage, so LSTM will be much better than FNN.

Question 2：

Is there any improvement in terms of performance by using the target network? If so, explain why.

Answer:

As can be seen from the above figure 16 and figure 18, in FNN, the target network has a more obvious advantage, effectively alleviating the curve oscillation and jitter of FNN without target.In LSTM, this is not obvious. We can also see that the final video quality of the simulation with target network is lower than the simulation without the target network.

Reason:

In RL, when a nonlinear function is used to approximate the Q value function, the update of the Q value is prone to oscillate, showing unstable learning behavior. The role of target network is actually a mechanism to disrupt correlation. Using target network will cause two networks with identical structures but different parameters to appear in DQN. The policy network that predicts Q estimates uses the latest Parameters, and the Target network parameter of the neural network that predicts Q reality is used before 20 segments time before（in our lab）. After the introduction of Target Net, the target Q value remains unchanged for a period of time, which reduces the correlation between

the current Q value and the target Q value to a certain extent, improves the stability of the algorithm, and makes the learning process more stable.

Conclusion

In this work, we designed several Reinforcement Learning based DASH adaptation algorithms, exploiting Deep Q-networks to speed up the convergence and adaptability of the system, and improve its efficiency. Our DASH simulation uses different deep learning structures to approximate the Q-values, taking the raw system state as input and requiring no arbitrary design choices that might influence its performance. RNN with LSTM cells shows significantly better performance than FNN schemes in all the three metrics.