CMU 10-703 Homework 2 Report

Zhengkun Dai, Xiaoyu Bai
Department of ECE
Carnegie Mellon University
Pittsburgh, PA 15213
zhengkud@andrew.cmu.edu, xiaoyub@andrew.cmu.edu

1. Problem 1

In this case, $\mathrm{Q}_w(s,a) = w^T \phi(s,a)_{s'a'}$ and $\phi(s,a)_{s'a'} = 1[s' = s, a' = a]$.

Therefore, $\nabla_w \mathrm{Q}_w(s,a) = \phi(s,a)_{s'a'} = 1[s' = s, a' = a]$.

When $s' = s, a' = a$, (1) and (2) are exactly the same;

When $s' \neq s, or\ a' \neq a$, both (1) and (2) do not update.

2. Software Structure

In the following implementations, our software structure is listed as below:

./main_Q2.py
./main_Q3.py
./main_Q4.py
./main_Q5.py
./main_Q6.py
./main_Q7.py
./deeprl_hw2/nn_lib.py
./deeprl_hw2/improcess.py
./deeprl_hw2/policy.py
./deeprl_hw2/memhelpers.py

The folder of "deeprl_hw2" contains all the helper functions that the main script needs to call. To run implementation of each problem, simply call the main function of each problem. Please keep all the file in the same directory when running the code. Below is a brief description of each script.

"nn_lib.py"

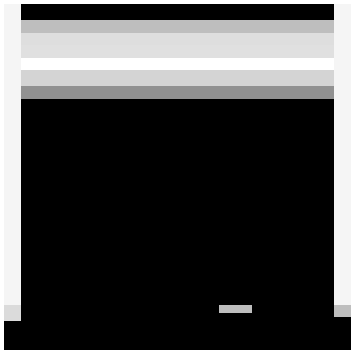
Library file which contains all the functions that deals with neural network. We implemented our neural network using Keras functional API since Keras is a higher level API of Tensorflow. Except for the helper functions that help neural network model to initialize, train, predict and evaluate, this file contains 3 classes:

NN_linear: linear neural network which is used in Q2, Q3, and Q4;
NN_cnn: deep Q-network which is used in Q5 and Q6;
NN_cnn_duel: deep Q-network with dueling structure which is used in Q7.

"improcess,py"

Preprocessor file. It contains 2 classes:

1. AtariProcessor: process feedback from environment. Key feature functions includes:
    - Downscale image from environment RGB (210×160×3) to grey scale (84×84). A sample down-scaled image is shown here:



    - Clip reward to [-1,1].

2. HistoryStore: store last k frames (i.e. down-scaled image from AtariProcessor) from environment. Key feature functions includes:
    - Add latest frame, and keep all frames in time order.
    - Return 4 frames as a state for neural network input. The return shape is (1, 84×84×4) for linear neural network, (84, 84, 4) for convolution neural network.

"memhelpers.py"

Memory replay file. It contains 2 classes:

1. NNMemStore: store the learning experience from environment. Key functions includes:
    - Initialize with random action selector to 1/20 of the memory size before training.
    - Append a memory element and keep all memory elements in time order.
    - Return a batch of random sampled memory elements.
2. **NNSample:** Memory element for NNMemStore.

"policy.py"

Different policy selectors including:

1. UniformRandomPolicy: Random select an action.
2. GreedyPolicy: Select best action according to Q values.
3. GreedyEpsilonPolicy: With ε probability use random UniformRandomPolicy; with 1-ε probability use GreedyPolicy.
4. GreedyEpsilonPolicy: with linear decayed ε. ε linear decays from 1 to 0.1 in 1/5 of total interactions with environment.

3. Experiment Set-up

As suggested by TA, we implement Huber loss when compiling the model as well as MSE. The clip value in Huber loss is set to be 1. The Huber loss function is from an online open source in GitHub (https://github.com/matthiasplappert/keras-rl/blob/master/rl/util.py). We tried two

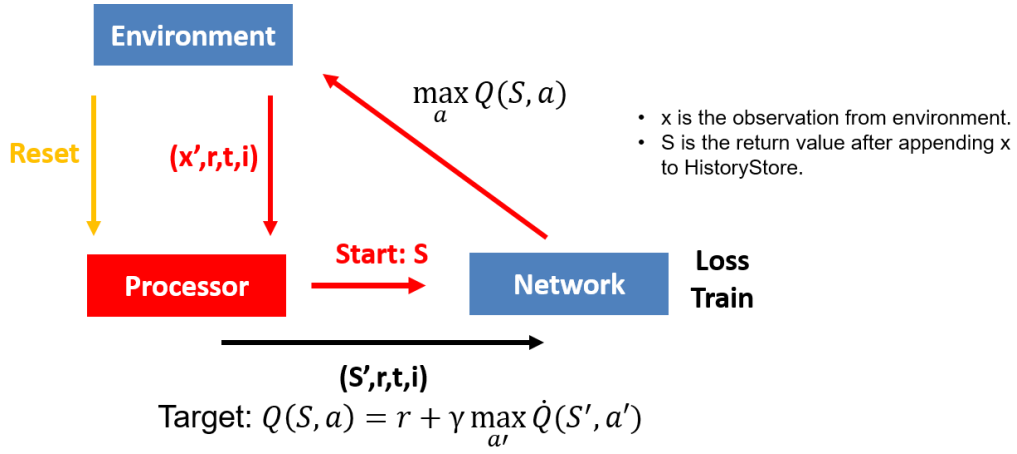
optimizers including rmsprop and adam. We tried two different learning rate schedules: fix learning rate at 0.00025 and learning rate decay from 0.01 to 0.0001 linearly.

With memory replay, large memory consumption is needed. The memory size is set to be 800k. In particular, at the moment we train our model, bridge PSC has severe delay issue. Therefore we bought 6 AWS machines to run our calculation. This time, we set losing one life as terminal state instead of losing the game (losing all 3 lives) during training. But the environment is reset only after losing the game (losing all 3 lives) to ensure the network can see the new states after losing one life.

As mentioned before, we use ε linear decay policy when storing states into the memory with ε linearly decay from 1 to 0.1. After interaction reaches 1/5 of its maximum value, ε is fixed as 0.1. This setup is the same as it is in the reference paper.

For the environment, we start with Breakout-v0 since it is considered a simple one among the Atari games.

4. Problem 2

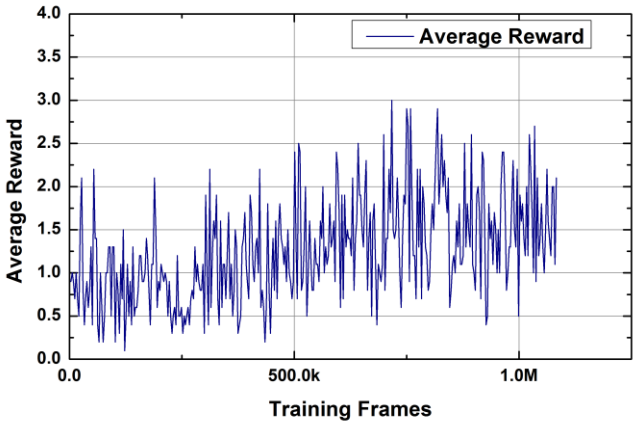


The logic flow of Q2 is as following:

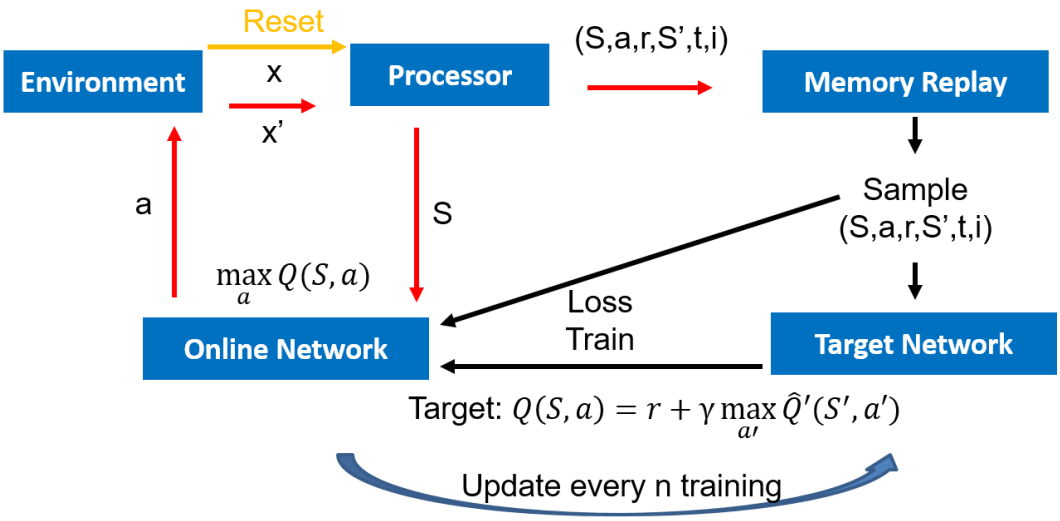
- Reset the environment: ENV->Processor
- Follow the red route to perform an interaction with environment:
- Processor → S → NN → max Q(S,a) → a → ENV → (x',r,t,i) → Processor
- Follow the black route to perform a training on network
- Processor → (S',r,t,i) → NN → max Q(S',a') → Target, Q(S,a) = r + γ,

  max Q(S',a') → NN(Train)
- Repeat(2) and (3), this time the return state from processor is S'

The setting in Q2 has several disadvantages. Without memory replay, the network may not be able to see enough states to guide its learning. Without target fixing, the network is updated every 4 interactions with the environment which is too frequent leading to unstable results. Linear network is naive to generalize the optimal behavior from the input states. In Q2, we simply use SGD instead of batch-GD which may also lead to slow learning. The results of our experiment from Q2 does not yield too much difference from random sampling.



5. Problem 3



Environment Interaction with online network:

- Store experience in memory replay(can interact n times before performing one training);
- a is chosen by online network according to S.

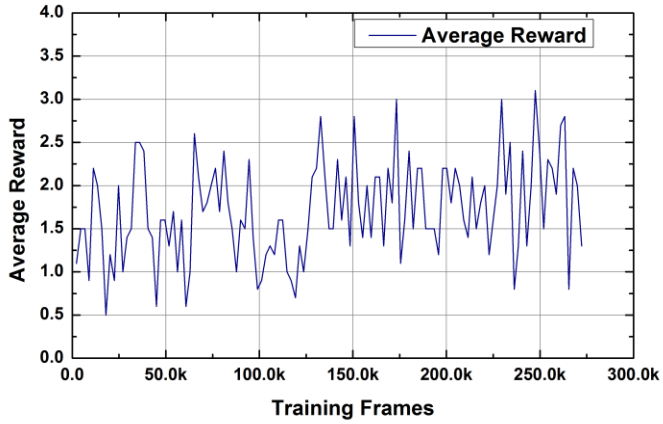
Train online network, sample data from memory replay:

- Q'(S',a') is calculated by target network according to S';
- a' is chosen by max Q'(S',a').

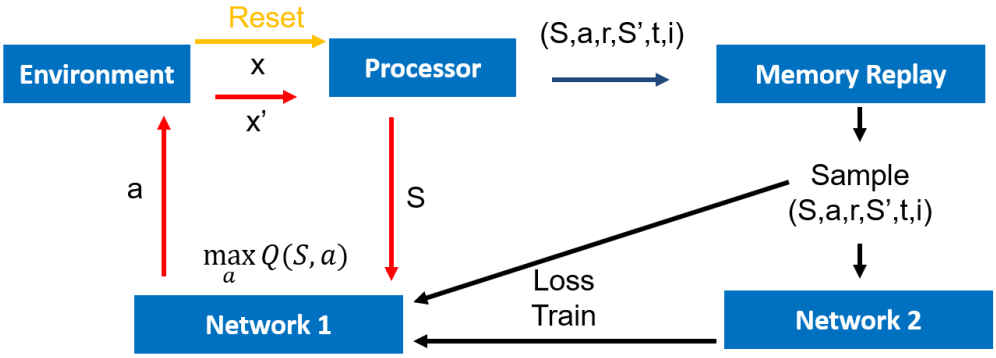
Although Q3 still uses linear network, it applies memory replay and target fixing.

Memory replay ensures that the network could see some unseen scenes from Q2 from the initial random policy. The training cases becomes more diverse and efficient in this way. Each experience might be used for many weight updates as we randomly sampled from memory. In addition, samples from network can sometimes have strong correlations which leads the batch training to be not fair. For example, sometimes only one action is updated for many updates.

The way we do target fixing is to make two identical networks at the beginning. One works as the target network and one works as the online network. Online network is trained every 4 interactions. After every 10k interactions, the target network is updated by the online network. Hopefully this method could make the learning process more stable. However the result we achieve from Q3 is still not as expected. We consider this might attribute to the naive activation in linear network.



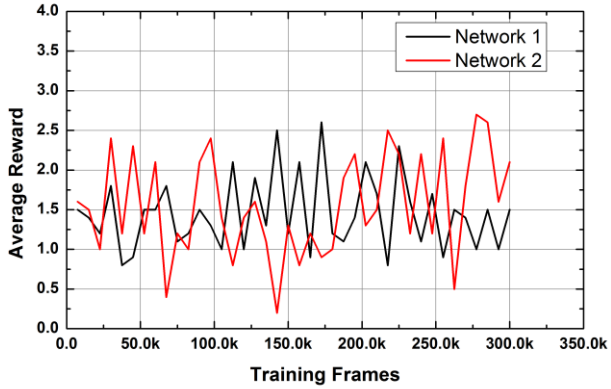
6. Problem 4



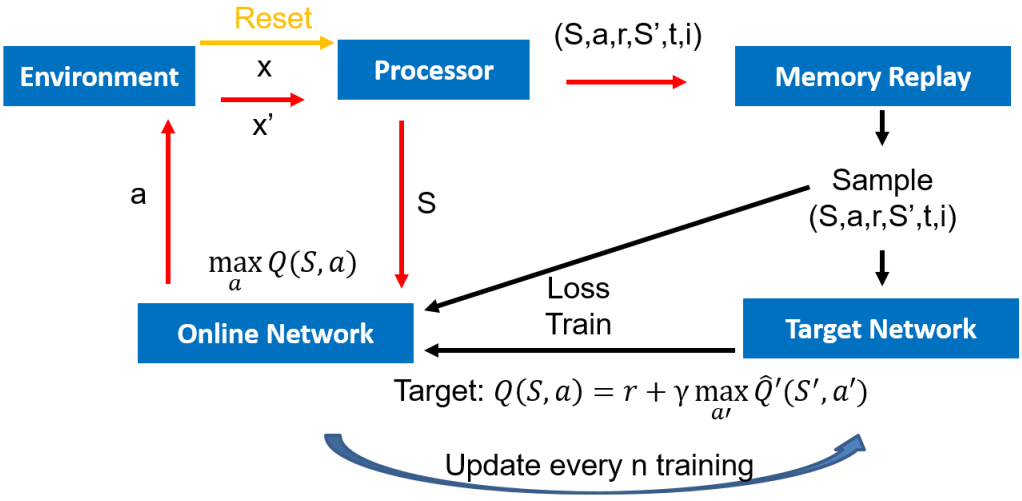
Target: $Q(S,a) = r + \gamma \max_{a'} \dot{Q}(S',a')$

Network 1 and network 2 may exchange role before each interaction. So no need to update the network.

The only difference between Q3 and Q4 is the network update. Previously we use one online network and one target network. Online network is trained every 4 interactions and target network is updated every 10k interactions. This time we use two equivalent networks and before each interaction we do a coin-flipping. During this interaction, each network has 50% chance to be online network or target network. Each 4 interactions the online network is trained. Hopefully the two network will converge to the same global minima at the end of training.

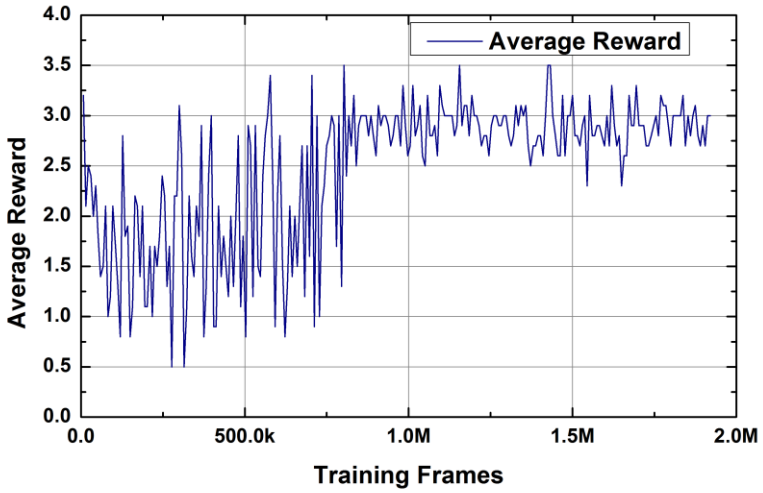


7. Problem 5



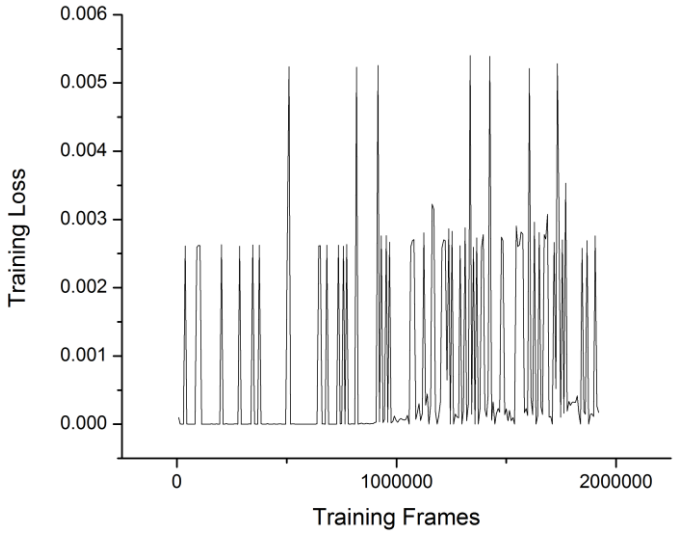
The logic flow of Q5 is similar to Q3 and the only difference is the network structure. Instead of a linear network, we are using a convolutional neural network which can better capture image feature generally. The network structure is exactly the same as in the nature paper. The model summary is listed below:

```
Layer (type)                 Output Shape              Param #
=================================================================
input (InputLayer)           (None, 84, 84, 4)         0
_________________________________________________________________
batch_normalization_1 (Batch (None, 84, 84, 4)         16
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 20, 20, 32)        8224
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 9, 9, 64)          32832
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 7, 7, 64)          36928
_________________________________________________________________
flatten_1 (Flatten)          (None, 3136)              0
_________________________________________________________________
dense_1 (Dense)              (None, 512)               1606144
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 3078
=================================================================
Total params: 1,687,222.0
Trainable params: 1,687,214.0
Non-trainable params: 8.0
```



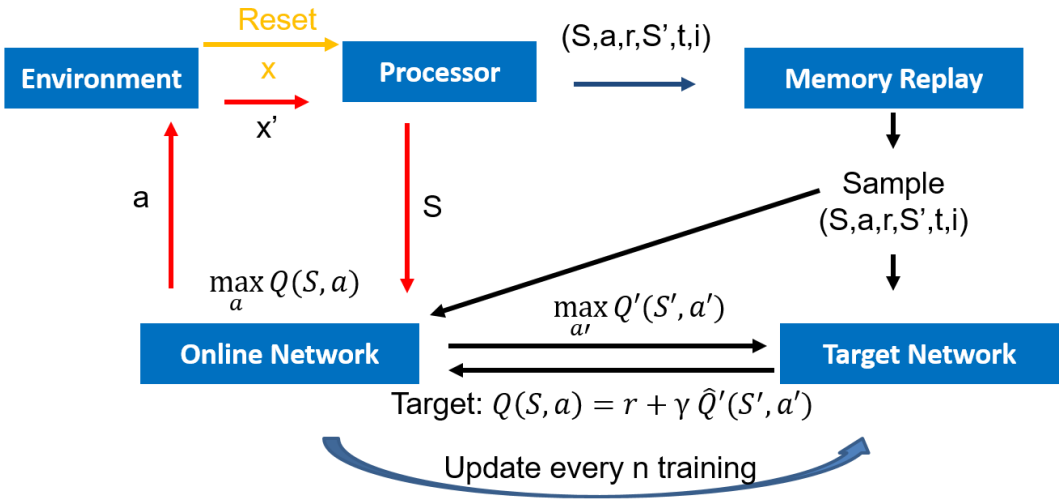
We expect this network to yield very nice performance but the results are not as we expected. The video shows that the agent moves towards one corner and just waits there. We suspect it has been stuck in a local minima so we change the learning rate schedule to a linear decay schedule from 0.01 to 0.0001 but the results still turns not as good. After 1 million frames of training, the agent seems still keep doing the same behavior. The reward per episode just maintain at 3 and does not improve after 2 million frames of training. Then we changed the optimizer from adam to rmsprop, the loss function from Huber loss to MSE, but nothing changes this situation… Each training time takes more than 10 hours and we are not able to run a grid search on the hyper parameters. Then we plot the loss vs interaction number.

This plot comes from Huber loss with adam optimizer. The loss seems to oscillate over the



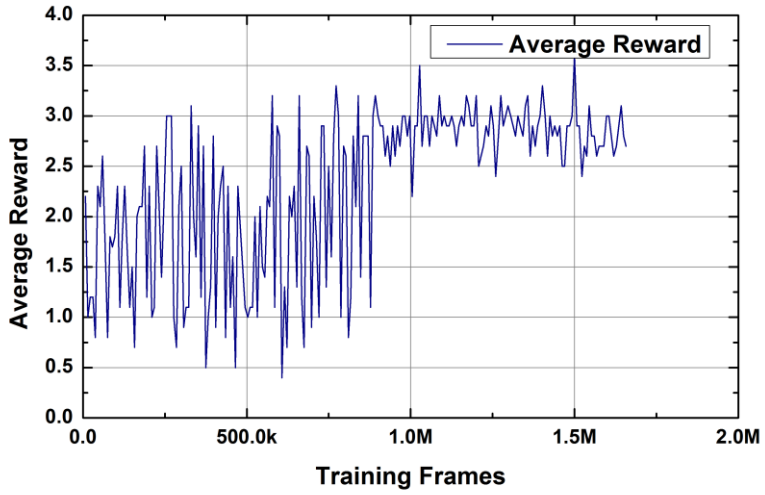
entire training process with large scale especially after 1 million of frames. We doubt that it might be due to somewhere the learning rate is too large that leads to this oscillation. But the linear decay learning rate (0.01 to 0.0001) still does not change the situation. Due to time limit we do not have time for further experiments.

8. Problem 6



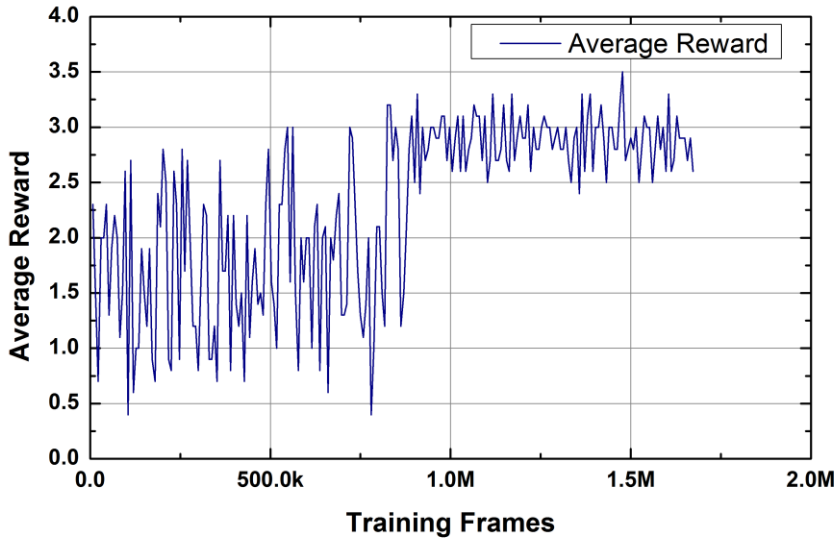
- a is chosen by online network from S
- a' is also chosen by online network from S'
- Target network only calculate Q'(S',a')

The difference between Q5 and Q6 is all the action is chosen by online network. The target network only provides Q value for training. Hopefully this decomposition of action selection and evaluation could reduce overestimation. Similar to Q5, our Q6 network yields similar results, so we doubt it is due to the same reason.



9. Problem 7

The implementation structure of Q7 is exactly the same with Q6 but with different network module. Dueling network insert another layer which decompose state value and action value. In this way, the network learns to deal with some states which critically needs certain actions. For example, in Breakout-v0, when the ball is away from the agent the action selection seems not very critical. But at the process the ball falling to the agent, action selection becomes very important.



10. Results Comparison

Problem	Average Reward per Episode
Q2	1.6075
Q3	1.8958
Q4	1.4475
Q5	2.9435
Q6	2.8152
Q7	2.8844

11. Conclusion and Discussion

In this homework, neural network is used to deal with the reinforcement learning in an environment with large state numbers. The basic idea is neural network can give an accurate(to some extent) estimate of Q(S,A) for all state. Then the agent can choose the best action according to max Q(s,a).

The learning procedure is assume the network is correct, then according to Bellman equation, the Q value of current state can be calculated from the next state and the immediate reward, which can be express as $Q(S,a)=r+\gamma*\max Q'(S',a')$.

By interacting with the environment and perform gradient descent to optimize the neural network, we can get the neural network to predict Q(S,A) more accurately, i.e. converge to the global minimum.

During this learning procedure, the loss function is estimated from the neural network(or an old copy of itself), which makes the learning very slow.

Memory replay can help us correct the neural network by checking the learning information again.

From the attached video, we can see all the networks(linear or CNN) start to learn to react to the environment at the beginning(i.e. try to move around to reflect the ball back in Breakout_v0).

The linear neural network is too simple to learn complex environment to give an accurate estimation of Q(S, A).

The CNN is more complex and good at dealing with image input, but it can get stuck into local minimum. From the videos analysis, it seems our neural network just learned how to react to the ball coming from a fixed point and moving in a fixed direction. Then it gets stuck there, it doesn't learn how to react to the following scenario.

For example, Q5 learns to reflect the ball coming to the left corner in 1/3 training, then it gets stuck there for 2/3 and 3/3.

Q6 first learns to reflect the ball coming to the right corner in 1/3 training, then for 2/3 training it thinks that reflect the ball coming to the left corner back is better, so it moves to the right corner immediately, but it gets stuck at the right corner until 3/3 training.

Q7 also learns to reflect the ball coming to the right corner in 1/3 training, then decides to catch the ball at left corner for the rest of the training

To make it better, we think the following changes might work:

(1) Provide more random actions as experience. During learning, we are updating the memory with the actions from neural network, the experience is gradually filled with the choice of the neural network. If it gets stuck in local minimum (probably the wrong choice), then it will never try to learn the other actions.

(2)Change the learning rate to get out of the local minimum.

(3)Change the updating frequency of target network. Though updating the target network too often can cause unstable problem. Proper updating frequency can provide learned information to target network timely

(4)Start learning with the terminal states (i.e. increase the ratio of terminal state in memory replay). We think start to learn from the terminal state can make the neural network to avoid getting into terminal state. This idea looks like a back propagation of the MDP procedure.

But due to time limitation, we don't have time to try them. We will try it in our project.

This is really an interesting and challenging homework, we would like to continue working on it after the deadline to see whether we could achieve better results.