AML

1. Solution

In this task, we design five different deep reinforcement learning agents to play the SuperMarioBros autonomously based on a tutorial (https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html) which uses Double DQN to play SuperMarioBros. These five agents are: Nature DQN (Mnih, 2015), Nature DQN_RNN, Nature DQN_Transformer, Double DQN (Van Hasselt, 2016), Dueling DQN (Wang, 2016) . Next, we will introduce the design concepts, advantages and disadvantages of these different DQNs:

a) Nature DQN: The basic DQN (Mnih, 2013) comes from Q-learning, but it is different from Q-learning that its Q value is calculated not directly through the state and action, but through a Q network (CNN). At the same time, the experience replay strategy is used to train the learning model, which can remove the correlations between samples. However, the basic DQN uses the same network when calculating the target Q value and the current Q value, which is not conducive to the convergence of the algorithm. Therefore, Nature DQN uses two Q networks. One is current network, which is used to select actions and update model parameters, and another is target network, which is used to calculate the target Q value. The parameters of the current network are copied to the target network at regular intervals, which can reduce the correlation between the target Q value and the current Q value. Besides, the Q network (CNN) of Nature DQN used in this work is a very simple structure: Conv8x8 -> Conv4x4 -> Conv3x3 -> Flatten -> Linear+Relu -> Linear.

b) Nature DQN_RNN: Here, we replace CNN used in Nature DQN with RNN, which is a single layer of LSTM with input_size = 84, hidden_size = 16, num_layers = 1.

c) Nature DQN_Transformer: Here, we replace CNN used in Nature DQN with Transformer. The Transformer framework we adopted (Dosovitskiy, 2020) is shown in Fig. 4:

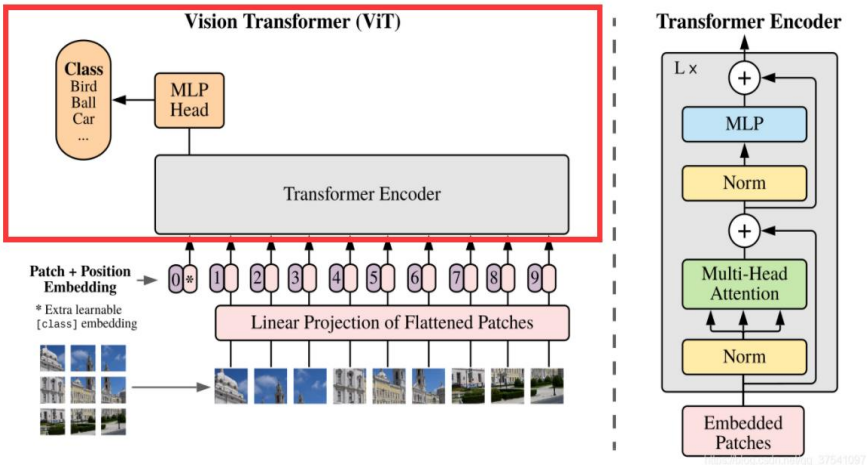


Fig. 4 Transformer Framework of the Nature DQN_Transformer agent (Dosovitskiy, 2020)

For this task, the parameters of Nature DQN_Transformer are set to image size = 84, patch size = 6, embedding dimension = 84, number heads = 3.

d) Double DQN (DDQN): In DQN and Nature DQN, the target Q value is directly obtained by E-greedy, resulting in over estimation. In order to solve this problem, DDQN no longer directly searches the maximum Q value of each action in the target network Q, but first finds the action corresponding

to the maximum Q value in the current network Q, and then uses the selected action to calculate the target Q value in the target network.

e) Dueling DQN: Dueling DQN improves the learning ability of agent by optimizing the network. It divides the network Q into two parts. The first part is only related to the state and has nothing to do with the specific action to be adopted. This part is called the state Value Function $V(s)$. The second part is related to both the state and action. This part is called Advantage Function $A(s,a)$, The final action value function is $Q_\pi(s,a)=V(s)+A(s,a)$. We can also see from Fig. 5 that the network architecture of Dueling DQN is different from that of single stream Q-Network:

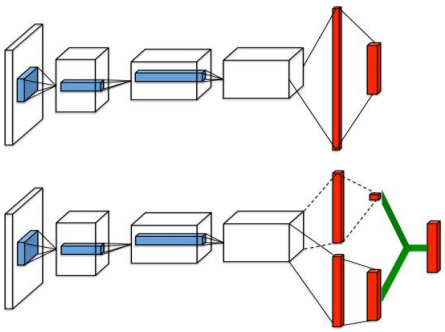


Fig. 5 The difference of network architecture between single stream Q-Network and Dueling DQN (Wang, 2016)

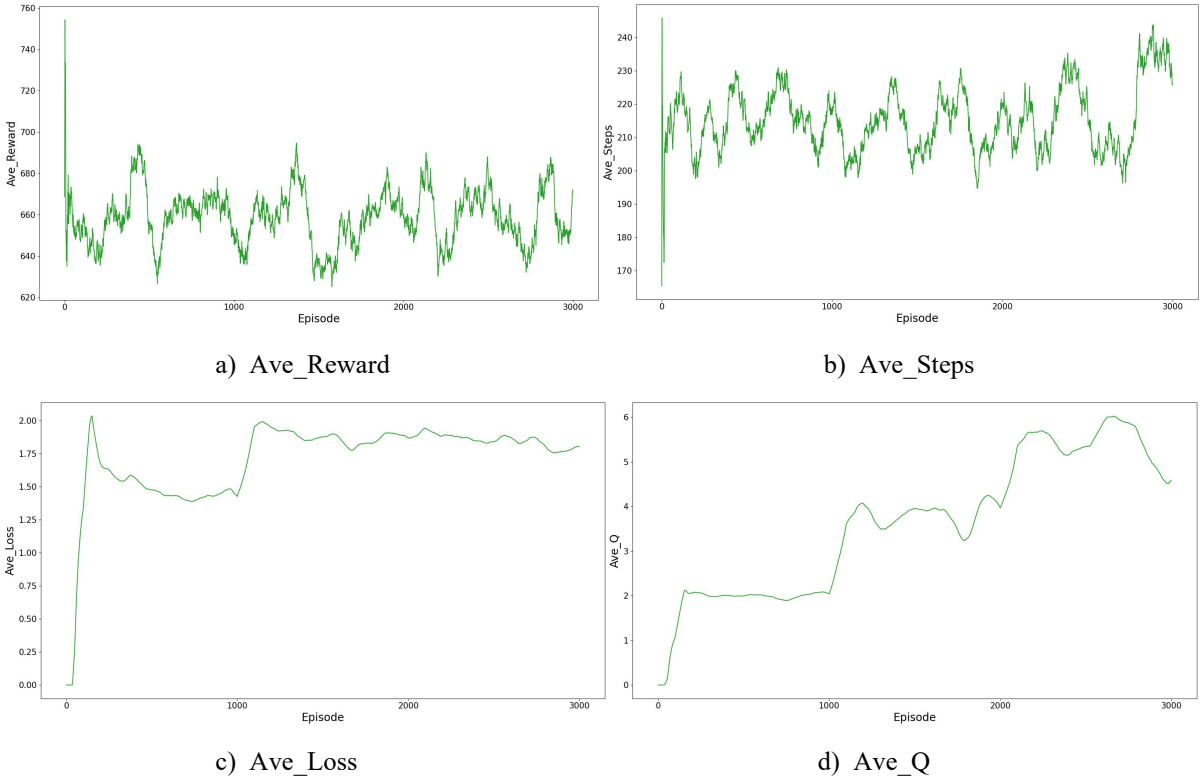
2. Experiments and Results

We conduct the training and test using above five different deep reinforcement learning agents. Due to limited training equipment, we set the maximum training episodes to 3000. The relevant parameters during training are set as Table 2:

Table 2. The parameters setting of training

Batch_size	64
Replay memory size	18000
Initial exploration rate	1
Minium exploration rate	0.1
Discount factor	0.9
Target network update frequency	1000 episodes
Optimizer	Adam
Learning rate	0.00001
Maximum episodes	3000

We recorded the average reward, average steps, average loss and average Q-value during the training of each agent. Then we averaged the training results of these five different agents. The overall change curves after averaging are shown in Fig. 6:



a) Ave_Reward b) Ave_Steps

c) Ave_Loss d) Ave_Q

Fig. 6 The overall change curves after averaging five different agents

We can see from Fig. 6, the Ave_ Loss is decreasing and Ave_ Q is rising. However, because the training episodes are not large enough, the Ave_ Reward and Ave_ Steps are still fluctuating.

At the same time, in order to compare the performance of different agents, we conducted 10 times replay test experiments using the each of trained model, and we averaged the results of 10 times replay test. The results are shown in Table 3:

Table 3. The average results of 10 times replay test

Agent	**Ave_Reward**
Nature_DQN (CNN)	421.0064
Nature DQN_RNN	268.6824
Nature DQN_Transformer	445.8128
Double DQN (CNN)	438.6004
Dueling DQN (CNN)	605.3765

We can see from Table 3, The Ave_Reward of Nature DQN_Transformer is higher than Nature_DQN (CNN) and Nature DQN_RNN. The Ave_Reward of Dueling DQN (CNN) is the highest, it achieves the best performance. So, in the future, we will consider to combine Dueling DQN with Transformer to design Dueling DQN_Transformer, which may get better performance.

3. Source code

The source code is shown at the end of the article (**see Appendix B**).

References

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. nature, 518(7540), pp.529-533.

Van Hasselt, H., Guez, A. and Silver, D., 2016, March. Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence (Vol. 30, No. 1).

Wang, Z., Schaul, T., Hessel, M., Hasselt, H., Lanctot, M. and Freitas, N., 2016, June. Dueling network architectures for deep reinforcement learning. In International conference on machine learning (pp. 1995-2003). PMLR.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S. and Uszkoreit, J., 2020. An image is worth 16x16 words: Transformers for image recognition at scale. arXiv preprint arXiv:2010.11929.

Appendix B

```
#
```

main.py

```
# The original frame is based on
# https://pytorch.org/tutorials/intermediate/mario_rl_tutorial.html
#
# This code are run on Python3 and Pytorch with GPU


import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'

import random, datetime
from pathlib import Path

import gym
import gym_super_mario_bros
from gym.wrappers import FrameStack, GrayScaleObservation, TransformObservation
from nes_py.wrappers import JoypadSpace
import argparse
from metrics import MetricLogger
from agent import Mario
from wrappers import ResizeObservation, SkipFrame
import time

# Training commands
# python main.py --agent Nature_DQN
# python main.py --agent Nature_DQN_RNN
# python main.py --agent Nature_DQN_Transformer
# python main.py --agent Double_DQN
# python main.py --agent Dueling_DQN

parser = argparse.ArgumentParser()
parser.add_argument('--agent', metavar='ARCH', default='Nature_DQN')
args = parser.parse_args()


# Initialize Super Mario environment
env = gym_super_mario_bros.make('SuperMarioBros-1-1-v0')

# Limit the action-space to
#   0. walk right
#   1. jump right
env = JoypadSpace(
    env,
    [['right'],
    ['right', 'A']]
)
```

```
# Apply Wrappers to environment
env = SkipFrame(env, skip=4)
env = GrayScaleObservation(env, keep_dim=False)
env = ResizeObservation(env, shape=84)
env = TransformObservation(env, f=lambda x: x / 255.)
env = FrameStack(env, num_stack=4)

env.reset()

save_dir = Path('checkpoints') / datetime.datetime.now().strftime('%Y-%m-%dT%H-%M-%S')
save_dir.mkdir(parents=True)

checkpoint = None #Path('checkpoints/2022-04-01T08-09-44/mario_net_10.chkpt')
mario = Mario(state_dim=(4, 84, 84), action_dim=env.action_space.n, save_dir=save_dir,
agent_type=args.agent, checkpoint=checkpoint)


# mario.exploration_rate = 0.2

logger = MetricLogger(save_dir)

episodes = 7001 #40000

### for Loop that train the model num_episodes times by playing the game
start_time = time.time()
for e in range(episodes):

    state = env.reset()

    # Play the game!
    while True:

        # 3. Show environment (the visual) [WIP]
        # env.render()

        # 4. Run agent on the state
        action = mario.act(state)

        # 5. Agent performs action
        next_state, reward, done, info = env.step(action)

        # 6. Remember
        mario.cache(state, next_state, action, reward, done)

        # 7. Learn
        q, loss = mario.learn()
        #print("q,loss:",q,loss)

        # 8. Logging
        logger.log_step(reward, loss, q)

        # 9. Update state
```

```
        state = next_state

        # 10. Check if end of game
        if done or info['flag_get']:
            break

    if e % mario.sync_every == 0:
        mario.sync_Q_target()

    if e % mario.save_every == 0:
        mario.save(e,args.agent)

    logger.log_episode()


    if e % 1 == 0:
        logger.record(
            episode=e,
            epsilon=mario.exploration_rate,
            step=mario.curr_step
        )
end_time = time.time()
training_time = end_time - start_time
print('\n')
print('training_time:',training_time)
```

agent.py

```
import torch
import random, numpy as np
from pathlib import Path
from torch.autograd import Variable
from neural import MarioNet,MarioNet_RNN,MarioNet_Dueling
from transformer import vit_base_patch16_224_in21k as transformer_model
from collections import deque


class Mario:
    def __init__(self, state_dim, action_dim, save_dir, agent_type='', checkpoint=None):
        self.state_dim = state_dim
        self.action_dim = action_dim
        self.agent_type = agent_type
        self.memory = deque(maxlen = 18000) # 100000
        self.batch_size = 64

        self.exploration_rate = 1
        self.exploration_rate_decay = 0.99999975
        self.exploration_rate_min = 0.1 #0
        self.gamma = 0.9

        self.curr_step = 0
        self.burnin = 1e4 # 1e5 # min. experiences before training
```

```
        self.learn_every = 3   # no. of experiences between updates to Q_online
        self.sync_every = 1000 #4, 4 eposide update target_net  # no. of experiences between Q_target & Q_online
sync

        self.save_every = 1000#1000 # 500 #500 eposide save once   # no. of experiences between saving Mario
Net
        self.save_dir = save_dir

        self.use_cuda = torch.cuda.is_available()

        if self.use_cuda:
            print("train on cuda")
        else:
            print("train on cpu")

        # Mario's DNN to predict the most optimal action - we implement this in the Learn section
        if self.agent_type == 'Nature_DQN': #Nature_DQN using CNN
            self.policy_net = MarioNet(self.state_dim, self.action_dim).float()
            self.target_net = MarioNet(self.state_dim, self.action_dim).float()
        elif self.agent_type == 'Nature_DQN_RNN': #Nature_DQN using RNN
            self.policy_net = MarioNet_RNN(self.state_dim, self.action_dim).float()
            self.target_net = MarioNet_RNN(self.state_dim, self.action_dim).float()
        elif self.agent_type == 'Nature_DQN_Transformer': #Nature_DQN using Transformer
            self.policy_net = transformer_model(self.state_dim[-1],self.action_dim).float()
            self.target_net = transformer_model(self.state_dim[-1], self.action_dim).float()
        elif self.agent_type == 'Double_DQN': #Double_DQN using CNN
            self.policy_net = MarioNet(self.state_dim, self.action_dim).float()
            self.target_net = MarioNet(self.state_dim, self.action_dim).float()
        elif self.agent_type == 'Dueling_DQN': #Dueling_DQN using CNN
            self.policy_net = MarioNet_Dueling(self.state_dim, self.action_dim).float()
            self.target_net = MarioNet_Dueling(self.state_dim, self.action_dim).float()

        if self.use_cuda:
            self.policy_net = self.policy_net.to(device='cuda')
            self.target_net = self.target_net.to(device='cuda')
        if checkpoint:
            self.load(checkpoint)

        self.optimizer = torch.optim.Adam(self.policy_net.parameters(), lr=0.00001) # 0.000025
        self.loss_fn = torch.nn.SmoothL1Loss()


    def act(self, state):
        """
        Given a state, choose an epsilon-greedy action and update value of step.

        Inputs:
        state(LazyFrame): A single observation of the current state, dimension is (state_dim)
        Outputs:
        action_idx (int): An integer representing which action Mario will perform
        """
        # EXPLORE
```

```
        if np.random.rand() < self.exploration_rate:
        #if np.random.rand() < 0:

            action_idx = np.random.randint(self.action_dim)

        # EXPLOIT
        else:

            state = torch.FloatTensor(state).cuda() if self.use_cuda else torch.FloatTensor(state)
            state = state.unsqueeze(0)

            action_values = self.policy_net(state)
            action_idx = torch.argmax(action_values, axis=1).item()

        # decrease exploration_rate
        self.exploration_rate *= self.exploration_rate_decay
        self.exploration_rate = max(self.exploration_rate_min, self.exploration_rate)

        # increment step
        self.curr_step += 1
        return action_idx

    def cache(self, state, next_state, action, reward, done):
        """
        Store the experience to self.memory (replay buffer)
        Inputs:
        state (LazyFrame),
        next_state (LazyFrame),
        action (int),
        reward (float),
        done(bool))
        """

        state = torch.FloatTensor(state).cuda() if self.use_cuda else torch.FloatTensor(state)
        next_state = torch.FloatTensor(next_state).cuda() if self.use_cuda else torch.FloatTensor(next_state)
        action = torch.LongTensor([action]).cuda() if self.use_cuda else torch.LongTensor([action])
        reward = torch.DoubleTensor([reward]).cuda() if self.use_cuda else torch.DoubleTensor([reward])
        done = torch.BoolTensor([done]).cuda() if self.use_cuda else torch.BoolTensor([done])
        self.memory.append( (state, next_state, action, reward, done,) )


    def recall(self):
        """
        Retrieve a batch of experiences from memory
        """
        batch = random.sample(self.memory, self.batch_size)
        state, next_state, action, reward, done = map(torch.stack, zip(*batch))
        return state, next_state, action.squeeze(), reward.squeeze(), done.squeeze()

    def td_estimate(self, state, action):

        #print('state:', state)
```

```
        #print('state.shape:', state.shape)
        #print('[np.arange(0, self.batch_size), action]:',[np.arange(0, self.batch_size), action])
        #print(' self.policy_net(state)',  self.policy_net(state))
        #print(' self.policy_net(state)[0]', self.policy_net(state[0]))
        #print('state,',state)

        # do computing for each item from one batch,then merge the result
        if self.agent_type == 'Nature_DQN_RNN' or self.agent_type == 'Nature_DQN_Transformer':
            policy_out = []
            for i in range(state.shape[0]):
                #print('state[i]:',state[i])
                RNN_out = self.policy_net(state[i])
                #print('RNN_out:',RNN_out)
                RNN_out = RNN_out.numpy().tolist()
                #print('list_RNN_out[0]:', RNN_out[0])
                policy_out.append(RNN_out[0])
            #print('policy_out',policy_out)
            current_Q = torch.Tensor(policy_out)
            current_Q = Variable(current_Q, requires_grad=True)
            current_Q = current_Q.cuda()
            current_Q = current_Q[np.arange(0, self.batch_size), action]  # Q_online(s,a)
            #print('current_Q', current_Q)

        else:
            #print('self.policy_net(state):',self.policy_net(state))
            current_Q = self.policy_net(state)[np.arange(0, self.batch_size), action] # Q_online(s,a)
            #print('current_Q',current_Q)

        return current_Q

    @torch.no_grad()
    def td_target(self, reward, next_state, done):

        # refer to https://github.com/sachinruk/Mario/blob/master/dqn_agent.py
        if self.agent_type=='Nature_DQN' or self.agent_type == 'Nature_DQN_RNN' or self.agent_type ==
'Nature_DQN_Transformer': #Nature_DQN
            #print('next_state.shape,', next_state.shape)

            # do computing for each item from one batch,then merge the result
            if self.agent_type == 'Nature_DQN_RNN' or self.agent_type == 'Nature_DQN_Transformer':
                Q_out = []
                for i in range(next_state.shape[0]):
                    out = self.target_net(next_state[i]).max(-1, keepdim=True)[0]
                    #print('out:',out)
                    out = out.numpy().tolist()
                    #print('list_out[0]:', out[0])
                    Q_out.append(out[0])
                #print('Q_out',Q_out)
                next_Q = torch.Tensor(Q_out)
                next_Q = next_Q.cuda()
                #print('next_Q', next_Q)
                return (reward + (1 - done.float()) * self.gamma * next_Q).float()
```

```
            else:
                next_Q = self.target_net(next_state).max(-1, keepdim=True)[0]
                #print('next_Q',next_Q)
                return (reward + (1 - done.float()) * self.gamma * next_Q).float()

        else:

            next_state_Q = self.policy_net(next_state)
            best_action = torch.argmax(next_state_Q, axis=1)
            next_Q = self.target_net(next_state)[np.arange(0, self.batch_size), best_action]
            return (reward + (1 - done.float()) * self.gamma * next_Q).float()

    def update_Q_online(self, td_estimate, td_target) :
        loss = self.loss_fn(td_estimate, td_target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        return loss.item()


    def sync_Q_target(self):
        self.target_net.load_state_dict(self.policy_net.state_dict())


    def learn(self):
        # if self.curr_step % self.sync_every == 0:
        #     self.sync_Q_target()

        # if self.curr_step % self.save_every == 0:
        #     self.save()

        if self.curr_step < self.burnin:
            return None, None

        if self.curr_step % self.learn_every != 0:
            return None, None

        # Sample from memory
        state, next_state, action, reward, done = self.recall()

        # Get TD Estimate
        td_est = self.td_estimate(state, action)

        # Get TD Target
        td_tgt = self.td_target(reward, next_state, done)

        # Backpropagate loss through Q_online
        #print('td_est:',td_est)
        #print('td_tgt', td_tgt)
        #print('td_est.shape:', td_est.shape)
        #print('td_tgt.shape', td_tgt.shape)
```

```
        loss = self.update_Q_online(td_est, td_tgt)

        return (td_est.mean().item(), loss)


    def save(self,e,agent):
        if agent == 'Nature_DQN':
            save_path = self.save_dir /  f"Nature_DQN_mario_net_{int(e // self.save_every)}.chkpt"
        elif agent == 'Nature_DQN_RNN':
            save_path = self.save_dir /  f"Nature_DQN_RNN_mario_net_{int(e // self.save_every)}.chkpt"
        elif agent == 'Nature_DQN_Transformer':
            save_path = self.save_dir / f"Nature_DQN_Transformer_mario_net_{int(e // self.save_every)}.chkpt"
        elif agent == 'Double_DQN':
            save_path = self.save_dir /  f"Double_DQN_mario_net_{int(e // self.save_every)}.chkpt"
        elif agent == 'Dueling_DQN':
            save_path = self.save_dir /  f"Dueling_DQN_mario_net_{int(e // self.save_every)}.chkpt"

        #save_path = self.save_dir / f"mario_net_{int(e // self.save_every)}.chkpt"
        torch.save(
            dict(
                model=self.policy_net.state_dict(),
                exploration_rate=self.exploration_rate
            ),
            save_path
        )
        print(f"MarioNet saved to {save_path} at step {self.curr_step}")


    def load(self, load_path):
        if not load_path.exists():
            raise ValueError(f"{load_path} does not exist")

        ckp = torch.load(load_path, map_location=('cuda' if self.use_cuda else 'cpu'))
        exploration_rate = ckp.get('exploration_rate')
        state_dict = ckp.get('model')

        print(f"Loading model at {load_path} with exploration rate {exploration_rate}")
        self.policy_net.load_state_dict(state_dict)
        self.sync_Q_target()
        self.exploration_rate = exploration_rate
```

neural.py

```
from torch import nn
import copy
import torch
from torch.autograd import Variable

# Nature_DQN or Double_DQN using CNN
# refer to https://github.com/yfeng997/MadMario
class MarioNet(nn.Module):
    '''mini cnn structure
```

```
    input -> (conv2d + relu) x 3 -> flatten -> (dense + relu) x 2 -> output
    '''
    def __init__(self, input_dim, output_dim):
        super().__init__()
        c, h, w = input_dim

        if h != 84:
            raise ValueError(f"Expecting input height: 84, got: {h}")
        if w != 84:
            raise ValueError(f"Expecting input width: 84, got: {w}")

        self.net = nn.Sequential(
            nn.Conv2d(in_channels=c, out_channels=32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten(),
            nn.Linear(3136, 512),
            nn.ReLU(),
            nn.Linear(512, output_dim)
        )

    def forward(self, input):
        #print('self.net(input):',self.net(input))
        return self.net(input)

# Nature_DQN or Double DQN using RNN
# refer to https://www.dandelioncloud.cn/article/details/1469891088759758850
class MarioNet_RNN(nn.Module):

    def __init__(self, input_dim, output_dim):
        super().__init__()
        c, h, w = input_dim

        if h != 84:
            raise ValueError(f"Expecting input height: 84, got: {h}")
        if w != 84:
            raise ValueError(f"Expecting input width: 84, got: {w}")

        self.rnn1 = nn.LSTM(
            input_size=84,
            hidden_size=16,
            num_layers=1, #2
            batch_first=True,  # （time_step,batch,input）时是 Ture

            #input_size=84,
            #hidden_size=128,
            #num_layers=2,
            #batch_first=True,  # （time_step,batch,input）时是 Ture
        )
```

```
        self.out = nn.Linear(16, output_dim)
        #self.out = nn.Linear(128, output_dim)

    def forward(self, x):
        x = x.view(-1, 84, 84)
        r_out, (h_n, h_c) = self.rnn1(x)
        out = self.out(r_out[:, -1, :])
        out = out[0, :]
        out = out.cpu().detach().numpy()
        out = [list(out)]
        out = torch.Tensor(out)
        #print('out:',out)

        return out

# Dueling_DQN using CNN
# refer to https://github.com/likemango/DQN-mario-xiaoyao
class MarioNet_Dueling(nn.Module):
    def __init__(self, input_dim, output_dim):
        super().__init__()
        c, h, w = input_dim

        if h != 84:
            raise ValueError(f"Expecting input height: 84, got: {h}")
        if w != 84:
            raise ValueError(f"Expecting input width: 84, got: {w}")

        self.commonLayer = nn.Sequential(
            nn.Conv2d(in_channels=c, out_channels=32, kernel_size=8, stride=4),
            nn.ReLU(),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=4, stride=2),
            nn.ReLU(),
            nn.Conv2d(in_channels=64, out_channels=64, kernel_size=3, stride=1),
            nn.ReLU(),
            nn.Flatten()
        )

        self.advantage = nn.Sequential(
            nn.Linear(3136, 512),
            nn.ReLU(),
            nn.Linear(512, output_dim)
            )

        self.value = nn.Sequential(
            nn.Linear(3136, 512),
            nn.ReLU(),
            nn.Linear(512, 1)
            )

    def forward(self, x):
```

```
        x = self.commonLayer(x)
        advantage = self.advantage(x)
        value = self.value(x)
        #print('value:',value)
        return advantage + value - advantage.mean()
```

transformer.py

```
# The frame of transformer
# input [batch,4,84,84]
# refer to:
#https://github.com/rwightman/pytorch-image-models/blob/master/timm/models/vision_transformer.py
#https://blog.csdn.net/lwf1881/article/details/123673666

from functools import partial
from collections import OrderedDict
import torch,math,warnings
import torch.nn as nn

def _no_grad_trunc_normal_(tensor, mean, std, a, b):
    # Cut & paste from PyTorch official master until it's in a few official releases - RW
    def norm_cdf(x):
        # Computes standard normal cumulative distribution function
        return (1. + math.erf(x / math.sqrt(2.))) / 2.

    if (mean < a - 2 * std) or (mean > b + 2 * std):
        warnings.warn("mean is more than 2 std from [a, b] in nn.init.trunc_normal_. "
                      "The distribution of values may be incorrect.",
                      stacklevel=2)

    with torch.no_grad():
        l = norm_cdf((a - mean) / std)
        u = norm_cdf((b - mean) / std)

        tensor.uniform_(2 * l - 1, 2 * u - 1)

        tensor.erfinv_()

        # Transform to proper mean, std
        tensor.mul_(std * math.sqrt(2.))
        tensor.add_(mean)

        # Clamp to ensure it's in the proper range
        tensor.clamp_(min=a, max=b)
        return tensor

def trunc_normal_(tensor, mean=0., std=1., a=-2., b=2.):
    return _no_grad_trunc_normal_(tensor, mean, std, a, b)

def drop_path(x, drop_prob: float = 0., training: bool = False):
```

```
    if drop_prob == 0. or not training:
        return x
    keep_prob = 1 - drop_prob
    shape = (x.shape[0],) + (1,) * (x.ndim - 1)  # work with diff dim tensors, not just 2D ConvNets
    random_tensor = keep_prob + torch.rand(shape, dtype=x.dtype, device=x.device)
    random_tensor.floor_()  # binarize
    output = x.div(keep_prob) * random_tensor
    return output

class DropPath(nn.Module):
    """
    Drop paths (Stochastic Depth) per sample  (when applied in main path of residual blocks).
    """
    def __init__(self, drop_prob=None):
        super(DropPath, self).__init__()
        self.drop_prob = drop_prob

    def forward(self, x):
        return drop_path(x, self.drop_prob, self.training)

class PatchEmbed(nn.Module):
    """
    From image to Embeadding
    """

    def __init__(self, img_size=224, patch_size=16, in_c=4, embed_dim=768, norm_layer=None):

        super().__init__()
        img_size = (img_size, img_size)
        patch_size = (patch_size, patch_size)
        self.img_size = img_size
        self.patch_size = patch_size
        self.grid_size = (int(img_size[0])/patch_size[0], int(img_size[1])/patch_size[1])
        self.num_patches = self.grid_size[0] * self.grid_size[1]

        self.proj = nn.Conv2d(in_c, embed_dim, kernel_size=patch_size, stride=patch_size)
        self.norm = norm_layer(embed_dim) if norm_layer else nn.Identity()

    def forward(self, x):


        if x.size()[0]==4: #[4,84,84]->[1,4,84,84]
            x = x.unsqueeze(0)

        B, C, H, W = x.shape
        assert H == self.img_size[0] and W == self.img_size[1], \
            f"Input image size ({H}*{W}) doesn't match model ({self.img_size[0]}*{self.img_size[1]})."

        x = self.proj(x).flatten(2).transpose(1, 2)
        x = self.norm(x)

        return x
```

```
class Attention(nn.Module):
    def __init__(self,
                 dim,
                 num_heads=8,
                 qkv_bias=False,
                 qk_scale=None,
                 attn_drop_ratio=0.,
                 proj_drop_ratio=0.):
        super(Attention, self).__init__()
        self.num_heads = num_heads
        head_dim = dim // num_heads
        self.scale = qk_scale or head_dim ** -0.5
        self.qkv = nn.Linear(dim, dim * 3, bias=qkv_bias)
        self.attn_drop = nn.Dropout(attn_drop_ratio)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(proj_drop_ratio)

    def forward(self, x):
        # [batch_size, num_patches + 1, total_embed_dim]
        B, N, C = x.shape

        qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, C // self.num_heads).permute(2, 0, 3, 1, 4)
        # [batch_size, num_heads, num_patches + 1, embed_dim_per_head]
        q, k, v = qkv[0], qkv[1], qkv[2]  # make torchscript happy (cannot use tensor as tuple)

        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)

        x = (attn @ v).transpose(1, 2).reshape(B, N, C)
        x = self.proj(x)
        x = self.proj_drop(x)
        return x


class Mlp(nn.Module):
    """
    MLP as used in Vision Transformer, MLP-Mixer and related networks
    """

    def __init__(self, in_features, hidden_features=None, out_features=None, act_layer=nn.GELU, drop=0.):

        super().__init__()
        out_features = out_features or in_features
        hidden_features = hidden_features or in_features
        self.fc1 = nn.Linear(in_features, hidden_features)
        self.act = act_layer()
        self.fc2 = nn.Linear(hidden_features, out_features)
        self.drop = nn.Dropout(drop)
```

```
    def forward(self, x):
        x = self.fc1(x)
        x = self.act(x)
        x = self.drop(x)
        x = self.fc2(x)
        x = self.drop(x)
        return x


class Block(nn.Module):
    def __init__(self,
                 dim,
                 num_heads,
                 mlp_ratio=4.,
                 qkv_bias=False,
                 qk_scale=None,
                 drop_ratio=0.,
                 attn_drop_ratio=0.,
                 drop_path_ratio=0.,
                 act_layer=nn.GELU,
                 norm_layer=nn.LayerNorm):
        super(Block, self).__init__()
        self.norm1 = norm_layer(dim)
        self.attn = Attention(dim, num_heads=num_heads, qkv_bias=qkv_bias, qk_scale=qk_scale,
                              attn_drop_ratio=attn_drop_ratio, proj_drop_ratio=drop_ratio)
        # NOTE: drop path for stochastic depth, we shall see if this is better than dropout here
        self.drop_path = DropPath(drop_path_ratio) if drop_path_ratio > 0. else nn.Identity()
        self.norm2 = norm_layer(dim)
        mlp_hidden_dim = int(dim * mlp_ratio)
        self.mlp = Mlp(in_features=dim, hidden_features=mlp_hidden_dim, act_layer=act_layer, drop=drop_ratio)

    def forward(self, x):
        x = x + self.drop_path(self.attn(self.norm1(x)))
        x = x + self.drop_path(self.mlp(self.norm2(x)))
        return x

class VisionTransformer(nn.Module):
    def __init__(self, img_size=224, patch_size=16, in_c=4, num_classes=1000,
                 embed_dim=768, depth=12, num_heads=12, mlp_ratio=4.0, qkv_bias=True,
                 qk_scale=None, representation_size=None, distilled=False, drop_ratio=0.,
                 attn_drop_ratio=0., drop_path_ratio=0., embed_layer=PatchEmbed, norm_layer=None,
                 act_layer=None):

        super(VisionTransformer, self).__init__()
        self.num_classes = num_classes
        self.num_features = self.embed_dim = embed_dim  # num_features for consistency with other models
        self.num_tokens = 2 if distilled else 1
        norm_layer = norm_layer or partial(nn.LayerNorm, eps=1e-6)
        act_layer = act_layer or nn.GELU

        self.patch_embed = embed_layer(img_size=img_size, patch_size=patch_size, in_c=in_c,
embed_dim=embed_dim)
```

```
        num_patches = int(self.patch_embed.num_patches)

        self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
        self.dist_token = nn.Parameter(torch.zeros(1, 1, embed_dim)) if distilled else None
        self.pos_embed = nn.Parameter(torch.zeros(1, num_patches + self.num_tokens, embed_dim))
        self.pos_drop = nn.Dropout(p=drop_ratio)

        dpr = [x.item() for x in torch.linspace(0, drop_path_ratio, depth)]  # stochastic depth decay rule
        self.blocks = nn.Sequential(*[
            Block(dim=embed_dim, num_heads=num_heads, mlp_ratio=mlp_ratio, qkv_bias=qkv_bias,
qk_scale=qk_scale,
                  drop_ratio=drop_ratio, attn_drop_ratio=attn_drop_ratio, drop_path_ratio=dpr[i],
                  norm_layer=norm_layer, act_layer=act_layer)
            for i in range(depth)
        ])
        self.norm = norm_layer(embed_dim)

        # Representation layer
        if representation_size and not distilled:
            self.has_logits = True
            self.num_features = representation_size
            self.pre_logits = nn.Sequential(OrderedDict([
                ("fc", nn.Linear(embed_dim, representation_size)),
                ("act", nn.Tanh())
            ]))
        else:
            self.has_logits = False
            self.pre_logits = nn.Identity()

        # Classifier head(s)
        self.head = nn.Linear(self.num_features, num_classes) if num_classes > 0 else nn.Identity()
        self.head_dist = None
        if distilled:
            self.head_dist = nn.Linear(self.embed_dim, self.num_classes) if num_classes > 0 else nn.Identity()

        # Weight init
        trunc_normal_(self.pos_embed, std=.02)
        if self.dist_token is not None:
            trunc_normal_(self.dist_token, std=0.02)

        trunc_normal_(self.cls_token, std=0.02)
        self.apply(_init_vit_weights)

    def forward_features(self, x):
        x = self.patch_embed(x)
        cls_token = self.cls_token.expand(x.shape[0], -1, -1)
        if self.dist_token is None:
            x = torch.cat((cls_token, x), dim=1)  # [B, 197, 768]
        else:
            x = torch.cat((cls_token, self.dist_token.expand(x.shape[0], -1, -1), x), dim=1)

        x = self.pos_drop(x + self.pos_embed)
```

```
        x = self.blocks(x)
        x = self.norm(x)
        if self.dist_token is None:
            return self.pre_logits(x[:, 0])
        else:
            return x[:, 0], x[:, 1]

    def forward(self, x):
        x = self.forward_features(x)
        if self.head_dist is not None:
            x, x_dist = self.head(x[0]), self.head_dist(x[1])
            if self.training and not torch.jit.is_scripting():
                # during inference, return the average of both classifier predictions
                return x, x_dist
            else:
                return (x + x_dist) / 2
        else:
            x = self.head(x)

        x = x[0, :]
        x = x.cpu().detach().numpy()
        x = [list(x)]
        x = torch.Tensor(x)

        return x

def _init_vit_weights(m):
    """
    ViT weight initialization
    :param m: module
    """
    if isinstance(m, nn.Linear):
        trunc_normal_(m.weight, std=.01)
        if m.bias is not None:
            nn.init.zeros_(m.bias)
    elif isinstance(m, nn.Conv2d):
        nn.init.kaiming_normal_(m.weight, mode="fan_out")
        if m.bias is not None:
            nn.init.zeros_(m.bias)
    elif isinstance(m, nn.LayerNorm):
        nn.init.zeros_(m.bias)
        nn.init.ones_(m.weight)

def vit_base_patch16_224_in21k(img_size: int = 84, num_classes: int = 2, has_logits: bool = True):
    model = VisionTransformer(img_size=img_size,
                              patch_size=6,
                              embed_dim=84,
                              depth=4,
                              num_heads=3,
                              representation_size=84 if has_logits else None,
                              num_classes=num_classes)
    return model
```

metrics.py

```
import numpy as np
import time, datetime
import matplotlib.pyplot as plt

class MetricLogger():
    def __init__(self, save_dir):
        self.save_log = save_dir / "log.csv"
        with open(self.save_log, "w") as f:
            f.write(
                f"{'Episode':>8}{'Step':>8}{'Epsilon':>10}{'MeanReward':>15}"
                f"{'MeanLength':>15}{'MeanLoss':>15}{'MeanQValue':>15}"
                f"{'TimeDelta':>15}{'Time':>20}\n"
            )
        self.ep_rewards_plot = save_dir / "reward_plot.jpg"
        self.ep_lengths_plot = save_dir / "length_plot.jpg"
        self.ep_avg_losses_plot = save_dir / "loss_plot.jpg"
        self.ep_avg_qs_plot = save_dir / "q_plot.jpg"

        # History metrics
        self.ep_rewards = []
        self.ep_lengths = []
        self.ep_avg_losses = []
        self.ep_avg_qs = []

        # Moving averages, added for every call to record()
        self.moving_avg_ep_rewards = []
        self.moving_avg_ep_lengths = []
        self.moving_avg_ep_avg_losses = []
        self.moving_avg_ep_avg_qs = []

        # Current episode metric
        self.init_episode()

        # Timing
        self.record_time = time.time()


    def log_step(self, reward, loss, q):
        self.curr_ep_reward += reward
        self.curr_ep_length += 1
        if loss:
            self.curr_ep_loss += loss
            self.curr_ep_q += q
            self.curr_ep_loss_length += 1

    def log_episode(self):
        "Mark end of episode"
        self.ep_rewards.append(self.curr_ep_reward)
```

```
        self.ep_lengths.append(self.curr_ep_length)

        if self.curr_ep_loss_length == 0:
            ep_avg_loss = 0
            ep_avg_q = 0
        else:
            ep_avg_loss = np.round(self.curr_ep_loss / self.curr_ep_loss_length, 5)
            ep_avg_q = np.round(self.curr_ep_q / self.curr_ep_loss_length, 5)

        self.ep_avg_losses.append(ep_avg_loss)
        self.ep_avg_qs.append(ep_avg_q)

        self.init_episode()

    def init_episode(self):
        self.curr_ep_reward = 0.0
        self.curr_ep_length = 0
        self.curr_ep_loss = 0.0
        self.curr_ep_q = 0.0
        self.curr_ep_loss_length = 0

    def record(self, episode, epsilon, step):
        mean_ep_reward = np.round(np.mean(self.ep_rewards[-100:]), 3)
        mean_ep_length = np.round(np.mean(self.ep_lengths[-100:]), 3)
        mean_ep_loss = np.round(np.mean(self.ep_avg_losses[-100:]), 3)
        mean_ep_q = np.round(np.mean(self.ep_avg_qs[-100:]), 3)
        self.moving_avg_ep_rewards.append(mean_ep_reward)
        self.moving_avg_ep_lengths.append(mean_ep_length)
        self.moving_avg_ep_avg_losses.append(mean_ep_loss)
        self.moving_avg_ep_avg_qs.append(mean_ep_q)

        last_record_time = self.record_time
        self.record_time = time.time()
        time_since_last_record = np.round(self.record_time - last_record_time, 3)

        print(
            f"Episode {episode} - "
            f"Step {step} - "
            f"Epsilon {epsilon} - "
            f"Mean Reward {mean_ep_reward} - "
            f"Mean Length {mean_ep_length} - "
            f"Mean Loss {mean_ep_loss} - "
            f"Mean Q Value {mean_ep_q} - "
            f"Time Delta {time_since_last_record} - "
            f"Time {datetime.datetime.now().strftime('%Y-%m-%dT%H:%M:%S')}"
        )

        with open(self.save_log, "a") as f:
            f.write(
                f"{episode:8d}{step:8d}{epsilon:10.3f}"
                f"{mean_ep_reward:15.3f}{mean_ep_length:15.3f}{mean_ep_loss:15.3f}{mean_ep_q:15.3f}"
                f"{time_since_last_record:15.3f}"
```

```
            f"{datetime.datetime.now().strftime('%Y-%m-%dT%H:%M:%S'):>20}\n"
        )

    '''
    for metric in ["ep_rewards", "ep_lengths", "ep_avg_losses", "ep_avg_qs"]:
        plt.plot(getattr(self, f"moving_avg_{metric}"))
        plt.savefig(getattr(self, f"{metric}_plot"))
        plt.clf()
    '''
```

replay.py

```
import random, datetime
from pathlib import Path

import gym,os,time
import gym_super_mario_bros
from gym.wrappers import FrameStack, GrayScaleObservation, TransformObservation
from nes_py.wrappers import JoypadSpace
import argparse
from metrics import MetricLogger
from agent import Mario
from wrappers import ResizeObservation, SkipFrame

#  reply commands
# python replay.py --agent Nature_DQN --model xxx.chkpt
# python replay.py --agent Nature_DQN_RNN --model xxx.chkpt
# python replay.py --agent Nature_DQN_Transformer --model xxx.chkpt
# python replay.py --agent Double_DQN --model xxx.chkpt
# python replay.py --agent Dueling_DQN --model xxx.chkpt

parser = argparse.ArgumentParser()
parser.add_argument('--agent', metavar='ARCH', default='Nature_DQN')
parser.add_argument('--model', metavar='M', default=' ')
args = parser.parse_args()

env = gym_super_mario_bros.make('SuperMarioBros-1-1-v0')

env = JoypadSpace(
    env,
    [['right'],
    ['right', 'A']]
)

env = SkipFrame(env, skip=4)
env = GrayScaleObservation(env, keep_dim=False)
env = ResizeObservation(env, shape=84)
env = TransformObservation(env, f=lambda x: x / 255.)
env = FrameStack(env, num_stack=4)

env.reset()
```

```
save_dir = Path('checkpoints') / datetime.datetime.now().strftime('%Y-%m-%dT%H-%M-%S')
save_dir.mkdir(parents=True)


trained_model = args.model
checkpoint = Path(trained_model)
mario = Mario(state_dim=(4, 84, 84), action_dim=env.action_space.n, save_dir=save_dir,
agent_type=args.agent, checkpoint=checkpoint)

mario.exploration_rate = mario.exploration_rate_min

logger = MetricLogger(save_dir)

episodes = 10

start_time = time.time()
for e in range(episodes):

    state = env.reset()

    while True:

        env.render()

        action = mario.act(state)

        next_state, reward, done, info = env.step(action)

        mario.cache(state, next_state, action, reward, done)

        logger.log_step(reward, None, None)

        state = next_state

        if done or info['flag_get']:
            break

    logger.log_episode()

    if e % 1 == 0:
        logger.record(
            episode=e,
            epsilon=mario.exploration_rate,
            step=mario.curr_step
        )


end_time = time.time()
test_time = end_time - start_time
print('\n')
print('Ave_test_time:',test_time / float(episodes))
env.close()
```

training_plot_Average.py

```
# plot the Average training curves of different agents

# coding=utf-8
import csv
import numpy as np
from numpy import genfromtxt
import matplotlib.pyplot as plt
from matplotlib.pyplot import MultipleLocator
import logging
import matplotlib.ticker as mticker


def plot1(logged_data):
    logged_data = np.array(logged_data)
    fig, axs = plt.subplots(1)

    plt.xticks(fontname="Times New Roman", fontsize=15)
    plt.yticks(fontname="Times New Roman", fontsize=15)
    plt.xlabel('Episode', fontdict={"family": "Times New Roman", "size": 20})
    plt.ylabel('Ave_Reward', fontdict={"family": "Times New Roman", "size": 20})

    #axs.set(xlabel='Episode', ylabel='Ave_Reward')
    #axs.set_yscale('log')
    axs.plot(logged_data[:, 0], logged_data[:, 3], 'tab:green')

    x_major_locator = MultipleLocator(1000)  #
    axs.xaxis.set_major_locator(x_major_locator)  #

    plt.show()

def plot2(logged_data):
    logged_data = np.array(logged_data)
    fig, axs = plt.subplots(1)

    plt.xticks(fontname="Times New Roman", fontsize=15)
    plt.yticks(fontname="Times New Roman", fontsize=15)
    plt.xlabel('Episode', fontdict={"family": "Times New Roman", "size": 20})
    plt.ylabel('Ave_Steps', fontdict={"family": "Times New Roman", "size": 20})

    #axs.set(xlabel='Episode', ylabel='Ave_Steps')
    #axs.set_yscale('log')
    axs.plot(logged_data[:, 0], logged_data[:, 4], 'tab:green')
    x_major_locator = MultipleLocator(1000)  #
    axs.xaxis.set_major_locator(x_major_locator)  #

    plt.show()

def plot3(logged_data):
    logged_data = np.array(logged_data)
```

```
    fig, axs = plt.subplots(1)

    plt.xticks(fontname="Times New Roman", fontsize=15)
    plt.yticks(fontname="Times New Roman", fontsize=15)
    plt.xlabel('Episode', fontdict={"family": "Times New Roman", "size": 20})
    plt.ylabel('Ave_Loss', fontdict={"family": "Times New Roman", "size": 20})

    #axs.set(xlabel='Episode', ylabel='Ave_Loss')
    #axs.set_yscale('log')
    axs.plot(logged_data[:, 0], logged_data[:, 5], 'tab:green')
    x_major_locator = MultipleLocator(1000)  #
    axs.xaxis.set_major_locator(x_major_locator)  #

    plt.show()

def plot4(logged_data):
    logged_data = np.array(logged_data)
    fig, axs = plt.subplots(1)

    plt.xticks(fontname="Times New Roman", fontsize=15)
    plt.yticks(fontname="Times New Roman", fontsize=15)
    plt.xlabel('Episode', fontdict={"family": "Times New Roman", "size": 20})
    plt.ylabel('Ave_Q', fontdict={"family": "Times New Roman", "size": 20})

    #axs.set(xlabel='Episode', ylabel='Ave_Q')
    #axs.set_yscale('log')
    axs.plot(logged_data[:, 0], logged_data[:, 6], 'tab:green')
    x_major_locator = MultipleLocator(1000)  #
    axs.xaxis.set_major_locator(x_major_locator)  #

    plt.show()


# when generates log after training, needs to modify the name of log file to the corresponding name manually
# which is shown as below
#
#Read dataset to numpy.array and remove the first row of the dataset (variable name)
data1 = genfromtxt("log_nature_dqn.csv") #read dataset by genfromtxt
#print('type(data):',type(data))
data1 = data1[1:3000][:] # Remove the first row of the array (variable name)

data2 = genfromtxt("log_double_dqn.csv") #read dataset by genfromtxt
data2 = data2[1:3000][:] # Remove the first row of the array (variable name)

data3 = genfromtxt("log_dueling_dqn.csv") #read dataset by genfromtxt
data3 = data3[1:3000][:] # Remove the first row of the array (variable name)

data4 = genfromtxt("log_nature_dqn_rnn.csv") #read dataset by genfromtxt
data4 = data4[1:3000][:] # Remove the first row of the array (variable name)

data5 = genfromtxt("log_nature_dqn_transformer.csv") #read dataset by genfromtxt
data5 = data5[1:3000][:] # Remove the first row of the array (variable name)
```

```
Ave_data = (data1+data2+data3+data4+data5)/float(5)


plot1(Ave_data)
plot2(Ave_data)
plot3(Ave_data)
plot4(Ave_data)
```

###test_plot.py ###

```
# calculate the replay test results

# coding=utf-8
import csv
import numpy as np
from numpy import genfromtxt
import matplotlib.pyplot as plt
from matplotlib.pyplot import MultipleLocator
import logging
import matplotlib.ticker as mticker

def plot1(logged_data):
    logged_data = np.array(logged_data)
    fig, axs = plt.subplots(1)
    axs.set(xlabel='Episode', ylabel='Ave_Reward')
    axs.set_yscale('log')
    axs.plot(logged_data[:, 0], logged_data[:, 3], 'tab:green')

    x_major_locator = MultipleLocator(1000)  #
    axs.xaxis.set_major_locator(x_major_locator)  #

    plt.show()

def plot2(logged_data):
    logged_data = np.array(logged_data)
    fig, axs = plt.subplots(1)
    axs.set(xlabel='Episode', ylabel='Ave_Steps')
    axs.set_yscale('log')
    axs.plot(logged_data[:, 0], logged_data[:, 4], 'tab:green')
    x_major_locator = MultipleLocator(1000)  #
    axs.xaxis.set_major_locator(x_major_locator)  #

    plt.show()

def plot3(logged_data):
    logged_data = np.array(logged_data)
    fig, axs = plt.subplots(1)
    axs.set(xlabel='Episode', ylabel='Ave_Loss')
    axs.set_yscale('log')
    axs.plot(logged_data[:, 0], logged_data[:, 5], 'tab:green')
    x_major_locator = MultipleLocator(1000)  #
    axs.xaxis.set_major_locator(x_major_locator)  #
```

```
    plt.show()

def plot4(logged_data):
    logged_data = np.array(logged_data)
    fig, axs = plt.subplots(1)
    axs.set(xlabel='Episode', ylabel='Ave_Q')
    axs.set_yscale('log')
    axs.plot(logged_data[:, 0], logged_data[:, 6], 'tab:green')
    x_major_locator = MultipleLocator(1000)  #
    axs.xaxis.set_major_locator(x_major_locator)  #

    plt.show()


#Read dataset to numpy.array and remove the first row of the dataset (variable name)
data = genfromtxt("log.csv") #read dataset by genfromtxt
data = data[1:][:] # Remove the first row of the array (variable name)

#plot1(data)
#plot2(data)
#plot3(data)
#plot4(data)

print("Ave_Reward,Ave_Steps,Ave_Loss,Ave_Q:",sum(data[:, 3])/ float(10),sum(data[:, 4])/
float(10),sum(data[:, 5])/ float(10),sum(data[:, 6])/ float(10))
```

###wrappers.py ###

```
import gym
import torch
import random, datetime, numpy as np
from skimage import transform

from gym.spaces import Box

class ResizeObservation(gym.ObservationWrapper):
    def __init__(self, env, shape):
        super().__init__(env)
        if isinstance(shape, int):
            self.shape = (shape, shape)
        else:
            self.shape = tuple(shape)

        obs_shape = self.shape + self.observation_space.shape[2:]
        self.observation_space = Box(low=0, high=255, shape=obs_shape, dtype=np.uint8)

    def observation(self, observation):
        resize_obs = transform.resize(observation, self.shape)
        # cast float back to uint8
        resize_obs *= 255
        resize_obs = resize_obs.astype(np.uint8)
```

```
        return resize_obs


class SkipFrame(gym.Wrapper):
    def __init__(self, env, skip):
        """Return only every `skip`-th frame"""
        super().__init__(env)
        self._skip = skip

    def step(self, action):
        """Repeat action, and sum reward"""
        total_reward = 0.0
        done = False
        for i in range(self._skip):
            # Accumulate reward and repeat the same action
            obs, reward, done, info = self.env.step(action)
            total_reward += reward
            if done:
                break
        return obs, total_reward, done, info
```