Assignment 4

Peng Tang 517020910038

In this assignment , we are going to do experiment with deep Q network. We are going to compare the performance of DQN, Double DQN and Dueling DQN and test them in a classical RL control environment-MountainCar.

1 Analysis

1.1 Environment Explanation

In the MountainCar game, the goal is to drive up the mountain on the right. We are going to use the following information given by the environment to train our network and reach the goal by using the trained network in the end.

- State Space: there observation is a array containing two elements. The first one is the position of the car($[-1.2, 0.6]$), and the second one is the velocity of the car($[-0.07, 0.07]$).
- Starting State: the initial position of the car is assigned a uniform random value in $[-0.6, -0.4]$. The initial velocity of the car is 0.
- Action Space: there are three actions we can take in this environment, including $0, 1, 2.$ 0 means accelerating to the left, 1 means not accelerating, and 2 means accelerating to the right.
- Goal: making the car's position bigger than or equal to 0.5.
- Reward: reward is 0 if the car reach the goal, -1 otherwise.
- Terminal Conditions: the car position is more than 0.5 or the episode length is greater than 200.

1.2 DQN

DQN combines the deep neural network and q-learning. We implement this algorithm on this environment as follows.

1. Initialize replay memory D to capacity N.
2. Initialize evaluation neural network Q and target neural network Q' with same weights.
3. Initialize other parameters, like episodes times M, batch size S, exploration parameter ϵ, loss function F for Q, optimizer P for Q, update interval T for updating Q' with Q every T steps, and others.
4. Initialize the environment by using the `gym` interface and get the initial observation s, then we do the following steps until this episode is over.
    - select a action a by using the ϵ-greedy algorithm
    - get the next observation s', reward r and done d from the environment by take the action a
    - store (s, a, r, s', d) in D
    - set $s = s'$

- if D is full, sampling a bath B with size S from the D to train Q by using following steps.
    - compute evaluation value $q = Q(B(s), B(a))$
    - compute next states value $q_{next} = \max_a Q'(B(s'))$
    - compute target value $q' = B(r) + (1 - B(d)) * \lambda * q_{next}$
    - compute loss $L = F(q, q')$
    - perform a gradient descent by using L and update the weights of Q with P
- update the Q' with Q every T steps.

5. we do step 4 for M times.

By adjusting these parameters, we can get a good performance.

1.3 Double DQN

The Double DQN is used to solve the over-estimate problem of the DQN, and the difference is the way to compute the target value when training the network. Thus, we implement Double DQN as DQN and modify the step 4 in DQN as follows.

- compute evaluation value $q = Q(B(s), B(a))$
- compute the next states actions $a_{next} = \arg\max_a Q(B(s'))$
- compute next states value $q_{next} = Q'(B(s'), a_{next})$
- compute target value $q' = B(s) + (1 - B(d)) * \lambda * q_{next}$
- compute loss $L = F(q, q')$
- perform a gradient descent by using L and update the weights of Q with P

1.4 Dueling DQN

For dueling DQN, we need to change the structure of neural network. Specially, we are break the final layer in the neural network into two pieces, one is used to output $A(s, a)$ and the other is used to output $V(s)$. Then the final output of the dueling network is

$$Q(s, a) = V(s) + (A(s, a) - \frac{1}{|A|} \sum_a A(s, a))$$

2 Code

We implement the two methods in `DQN.py`.

2.1 Neural Network

```
class Network(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(Network, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Linear(input_dim, 32),
            nn.ReLU(),
            nn.Linear(32,64),
            nn.ReLU(),
            nn.Linear(64, output_dim),


        )

```

```
    def forward(self, input):
        output = self.layer1(input)
        return output

class DuelingNet(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(DuelingNet, self).__init__()

        self.layer1 = nn.Sequential(
            nn.Linear(input_dim, 32),
            nn.ReLU(),
            nn.Linear(32,64),
            nn.ReLU(),
        )
        self.layer_a = nn.Sequential(
            nn.Linear(64,output_dim)
        )
        self.layer_v = nn.Sequential(
            nn.Linear(64,1)
        )

    def forward(self, input):
        output1 = self.layer1(input)
        output_a = self.layer_a(output1)
        output_v = self.layer_v(output1)
        output = output_v + (output_a - torch.mean(output_a))
        return output
```

We use the `Network` class to build the Q and Q' in DQN and Double DQN and use the `DuelingNet` to build Q and Q' in Dueling DQN and Dueling&Double DQN. In the two networks, we use three liner layers and two activation function layers (`relu` function). And the `input_dim` should be 2 and the `output_dim` should be 3 in the problem.

2.2 DQN Class

This class contains two networks and one replay memory.

```
class DQN(object):
    def __init__(self, input_dim,  output_dim, NetworkType):

        self.input_dim = input_dim
        self.output_dim = output_dim
        self.NetworkType = NetworkType

        self.BufferSize = 10000
        self.replayBuffer = np.zeros(self.BufferSize, dtype=object)
        self.currentpointer = 0
        self.batchSize = 1024

        self.device = torch.device("cuda:0" if torch.cuda.is_available()
else "cpu")

        self.evalNet = None
        self.targetNet = None
        self.buildNetwork()

```

```
        self.targetNet.load_state_dict(self.evalNet.state_dict())
        self.loss_fun = torch.nn.MSELoss().to(self.device)
        self.optimizer = torch.optim.Adam(self.evalNet.parameters(),
lr=0.0001)

        self.updateinterval = 20
        self.epsilon = 1
        self.gamma = 0.99
        self.learntimes = 0

    def buildNetwork(self):

        #DQN or double DQN
        if self.NetworkType == 0 or self.NetworkType == 1:
            self.evalNet = Network(self.input_dim,
self.output_dim).to(self.device)
            self.targetNet = Network(self.input_dim,
self.output_dim).to(self.device)

        # Dueling DQN or Dueling double DQN
        elif self.NetworkType == 2 or self.NetworkType == 3:
            self.evalNet = DuelingNet(self.input_dim,
self.output_dim).to(self.device)
            self.targetNet = DuelingNet(self.input_dim,
self.output_dim).to(self.device)


    def get_action(self, obs, mode):

        obs = torch.from_numpy(np.array(obs)).float().to(self.device)
        q_val = self.targetNet(obs).detach().cpu().numpy()
        if mode:
            if random.random() < self.epsilon:
                return np.random.randint(self.output_dim)
            else:
                return np.argmax(q_val)
        else:
            return np.argmax(q_val)


    def push_transition(self, trans):
        self.currentpointer = self.currentpointer % self.BufferSize
        self.replayBuffer[self.currentpointer] = trans
        self.currentpointer += 1

    def get_traindata(self):
        Batch = np.random.choice(self.replayBuffer, self.batchSize,
replace=False)
        obsBatch = []
        actionBatch = []
        rewardBatch = []
        nextObsBatch = []
        doneBatch = []

        for i in range(self.batchSize):
            obsBatch.append(Batch[i][0])
            actionBatch.append(Batch[i][1])
            rewardBatch.append(Batch[i][2])
```

```
            nextObsBatch.append(Batch[i][3])
            doneBatch.append(Batch[i][4])

        obsBatch = np.array(obsBatch)
        actionBatch = np.array(actionBatch)
        rewardBatch = np.array(rewardBatch)
        nextObsBatch = np.array(nextObsBatch)
        doneBatch = np.array(doneBatch)

        return obsBatch, actionBatch, rewardBatch, nextObsBatch, doneBatch


    def train(self):


        obsBatch, actionBatch, rewardBatch, nextObsBatch, doneBatch =
self.get_traindata()
        obsBatch = torch.from_numpy(obsBatch).float().view(-1,
2).to(self.device)
        actionBatch = torch.LongTensor(actionBatch).view(-1,
1).to(self.device)
        rewardBatch = torch.from_numpy(rewardBatch).view(-1,
1).to(self.device)
        nextObsBatch = torch.from_numpy(nextObsBatch).float().view(-1,
2).to(self.device)
        doneBatch = torch.from_numpy(doneBatch).view(-1, 1).to(self.device)

        q_eval = self.evalNet(obsBatch).gather(1, actionBatch)
        q_next = self.targetNet(nextObsBatch).detach()

        # double DQN or dueling double DQN
        if self.NetworkType == 1 or self.NetworkType == 3:
            q_next_eval = self.evalNet(nextObsBatch).detach()
            max_action_indx = q_next_eval.max(1)[1].view(-1,1)
            q_target = (rewardBatch +
(torch.logical_not(doneBatch))*self.gamma*q_next.gather(1,
max_action_indx).view(-1, 1)).float()

        # Dueling DQN or DQN
        else:
            q_target = (rewardBatch +
(torch.logical_not(doneBatch))*self.gamma*(q_next.max(1)[0]).view(-1,
1)).float()

        loss = self.loss_fun(q_eval, q_target)
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        if self.learntimes % self.updateinterval == 0:
            #self.targetNet.load_state_dict(self.evalNet.state_dict())
            for target_param, param in zip(self.targetNet.parameters(),
self.evalNet.parameters()):
                target_param.data.copy_(target_param.data * 0.98 +
param.data * 0.02)

        self.learntimes += 1
        self.epsilon = self.epsilon*0.99  # 下降
```

```
118
```

- The class implements DQN, Double DQN, Dueling DQN and Dueling&Double DQN.
- In this class, the parameters **input_dim** and **output_dim** are used to initialize the two networks; **NetworkType** is used to choose different DQNs (0 means DQN, 1 means Double DQN, 2 means Dueling DQN, 3 means Dueling&Double DQN); **replayBuffer** is used to store transitions; **BufferSize** is the size of **replayBuffer**; **currentpointer** is current position of **replayBuffer** to store transition; **batchSize** is the size of batch sampled; **device** is used to choose the gpu/cpu; **evalNet** is evaluation network and **targetNet** is target network; **loss_fun** is the loss function; **optimizer** is the optimizer to update the **evalNet**; **updateinterval** is the interval steps of updating **targetNet**; **epsilon** is the ϵ of ϵ-greedy and we will decrease it as the process; **gamma** is the λ in the equation of computing q'; **learntimes** is memory the times of training.
- The **buildNetwork** is used to build different neural network according to the parameter **NetworkType**. If **NetworkType** is 0 or 1, we choose **Network** class; otherwise, we choose **DuelingNet** class.
- The **get_action** function is used to choose an action by giving a state. The parameter **mode** is to choose the ϵ-greedy algorithm or greedy algorithm.
- The **push_transition** function is used to store a transition in the replay memory.
- The **get_traindate** function is used to randomly sample a batch from the replay memory.
- The **train** function is used to train the evaluation network and update the target network every **updateinterval** steps. Because the only difference between the DQN and Double DQN lies in the **train** method, we can implement two different updating methods in this function to implement DQN and Double DQN. If **NetworkType** is 0 or 2, we choose DQN updating method; otherwise, we choose Double DQN updating method.

2.3 Test Function

We use this function to test the performance of our trained network by loading the parameters of our trained network. We test the score of 100 episodes to see the network's performance.

```
1  def testNetwork(parameterpath, type):
2      environment = gym.make('MountainCar-v0')
3
4      testModel = DQN(input_dim=2, output_dim=3, NetworkType=type)
5      testModel.targetNet.load_state_dict(torch.load(parameterpath))
6
7      rewardList = []
8      for i in range(100):
9          obs = environment.reset()
10         totalReward = 0
11         while True:
12             #environment.render()
13             action = testModel.get_action(obs, False)
14
15             nextobs, reward, done, info = environment.step(action)
16             totalReward += reward
17             obs = nextobs
18             if done:
19                 rewardList.append(totalReward)
20
21                 break
22     environment.close()
23
```

```
    return rewardList

```

2.4 Training Function

We use this function to perform the environment and get the training data to train needed network according to given parameter.

```
def training(type):

    environment = gym.make('MountainCar-v0')
    maxEpisodes = 600
    DQNModel = DQN(input_dim=2, output_dim=3, NetworkType=type)
    step = 0
    rewardlist = []
    maxReward = -300
    for episode in range(maxEpisodes):
        observation = environment.reset()
        totalreward = 0
        while True:
            step += 1
            action = DQNModel.get_action(observation, True)
            nextobservation, reward, done, info = environment.step(action)
            totalreward += reward
            reward = abs(nextobservation[0]+0.5)

            transtion = (observation, action, reward, nextobservation, done)
            DQNModel.push_transition(transtion)

            observation = nextobservation

            if step > DQNModel.BufferSize+1:  # 存满后开始训练
                DQNModel.train()

            if done:
                rewardlist.append(totalreward)
                print('type:', type ,'episode:', episode, 'totalreward', totalreward)
                if (totalreward > maxReward and totalreward > -125) or totalreward > -90:
                    currentTime = time.strftime("%Y_%m_%d_%H_%M_%S", time.localtime())
                    savepath = 'parameters/' + str(type) + '_'+currentTime + "_"+ str(episode) + str(totalreward) + '.pkl'
                    torch.save(DQNModel.targetNet.state_dict(), savepath)
                    if totalreward>maxReward:
                        maxReward = totalreward
                break

    environment.close()
    return rewardlist
```

In this function, we modify the reward from $r = -1$ to $r = |position + 0.5|$. In this game, the initial position is -0.5 and we need to drive up the left mountain at first to obtain a big velocity if we want to drive up the right mountain, so we give a bigger reward if we keep farer away from the initial position. In this way, we can reach our goal faster.

2.5 Main Function

This function is used to call the **training** function by giving four different parameters and draw the final results.

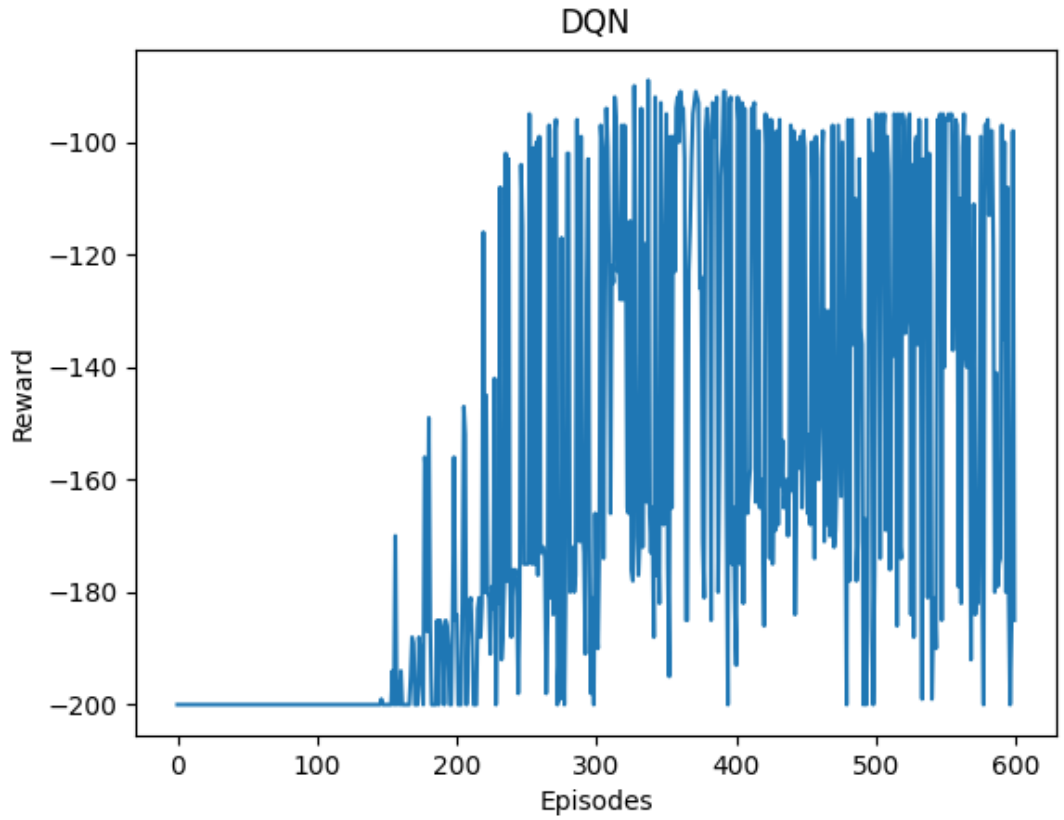
```
def main():
    set_seed(2)
    DQN_Reward = training(0)
    Double_DQN_Reward = training(1)
    Dueling_DQN_Reward = training(2)
    Dueling_Double_DQN_Reward = training(3)


    drawfig(DQN_Reward, 'DQN_Reward')
    drawfig(Double_DQN_Reward,  'Double_DQN_Reward')
    drawfig(Dueling_DQN_Reward,  'Dueling_DQN_Reward')
    drawfig(Dueling_Double_DQN_Reward,  'Dueling_Double_DQN_Reward')
```

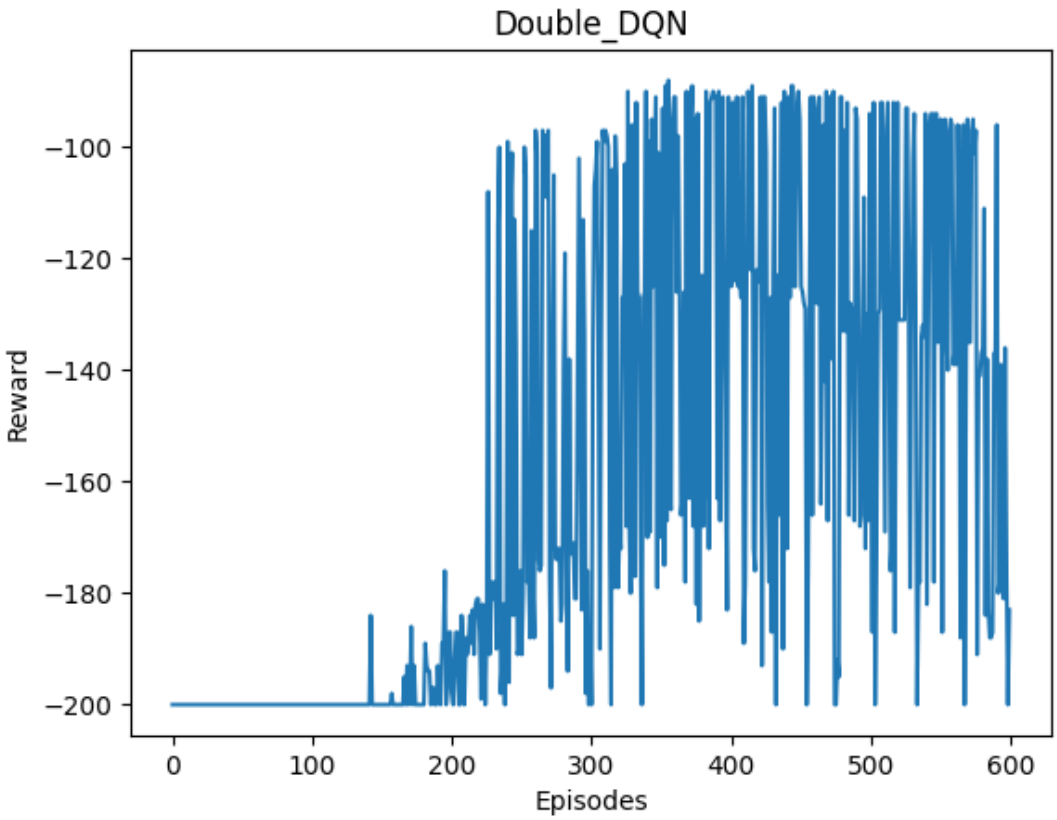
3 Experiment Results

We do experiment on the environment for four different DQNs and get the following results in training process.

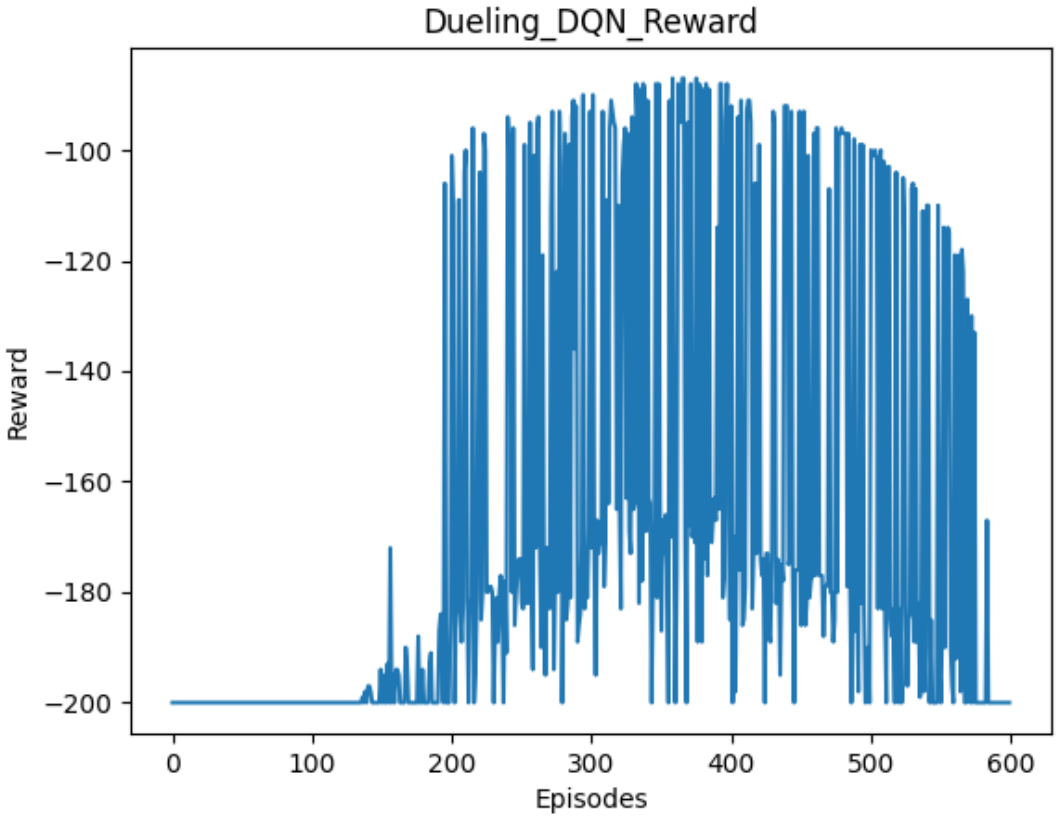
- **DQN**



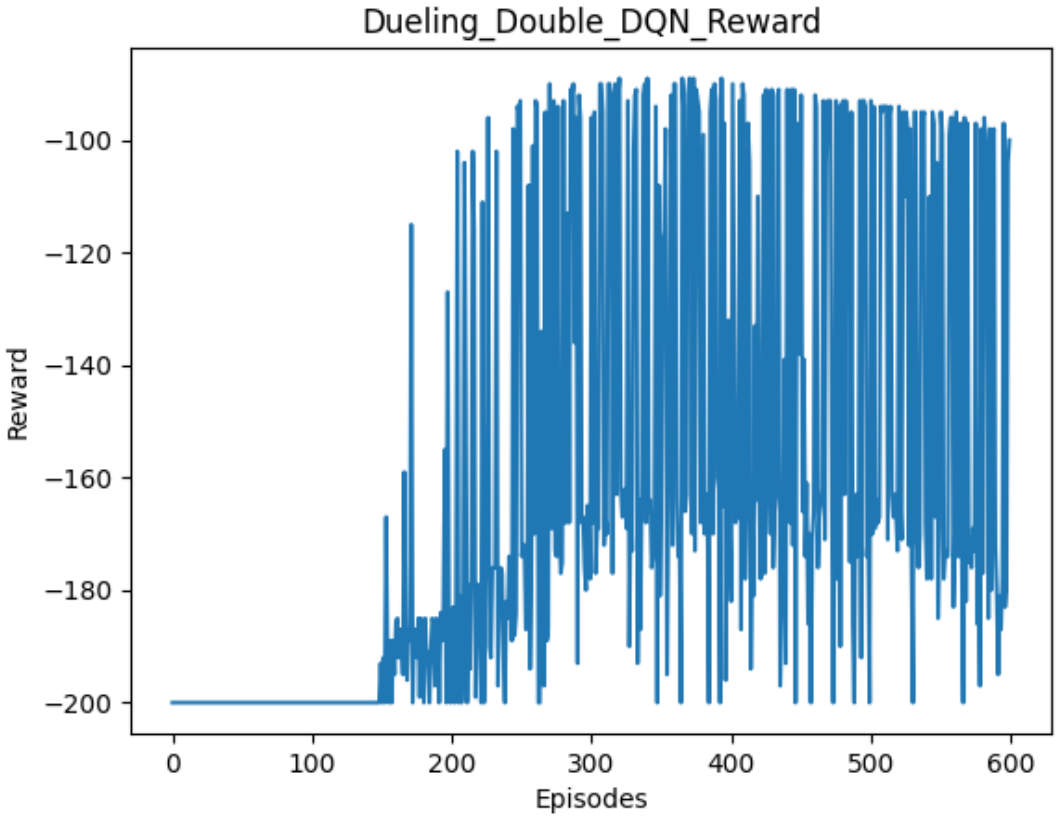
- **Double DQN**



- **Dueling DQN**

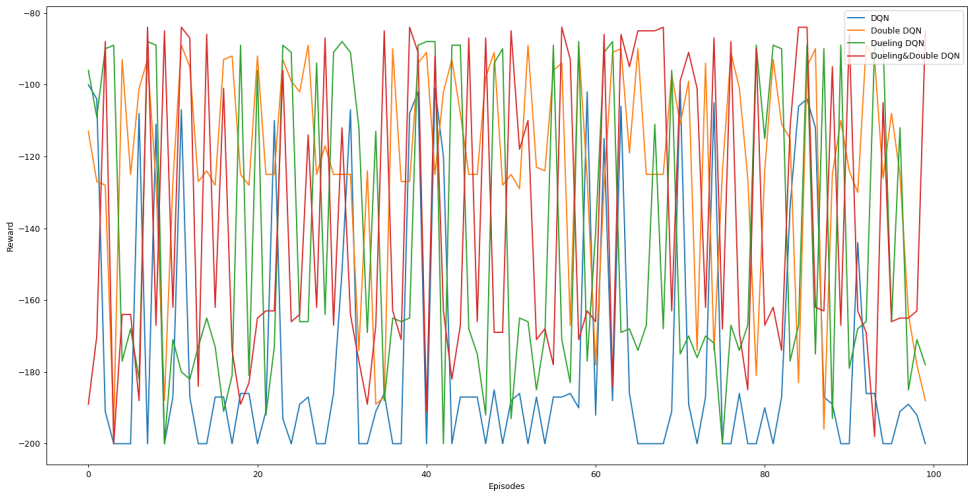


- **Double&Dueling DQN**



- From above figures, we can see that all methods begin to reach the goal after 150 episodes trained and all methods is fluctuation in the training process.

Then we test the trained network by performing 100 episodes on the environment and we get the following results.




```
D:\Anaconda\envs\pytorch\python.exe E:/大三下/强化学习/作业/homework4/code/DQN.py
DQN average:  -175.65
Double DQN average:  -121.3
Dueling DQN average:  -143.11
Dueling&Double DQN average:  -137.69
```

- From the two figures, we can see that Double DQN, Dueling DQN and Dueling&Double DQN have a better performance than DQN.

- Double DQN always reached the goal in the 100 episodes and Dueling DQN failed for several times.
- Dueling&Double DQN is not better than DQN in our experiment. This may be caused by hyper parameters.