Assignment 4 DQN report

1. Introduction

In classical Q-learning methods, the action value function Q is intractable with the increase of state space and action space. Deep Q network (DQN), which combines the advantage of Q-learning and the neural network, introduces the success of deep learning and has achieved a super-human level of play in atari games. In this experiment, I implement DQN algorithm and its improved algorithm, Double DQN and Dueling DQN and prioritized experience replay DQN, and play them in a classical reinforcement learning control environment–MountainCar.

MountainCar is on a one-dimensional track, positioned between two "mountains". The goal is to drive up the mountain on the right; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. It's observation and actions and reward are listed as follows.

- **Observation:**

Num	Observation	Min	Max
0	position	-1.2	0.6
1	velocity	-0.07	0.07

Actions:

Num	Action
0	push left
1	no push
2	push right

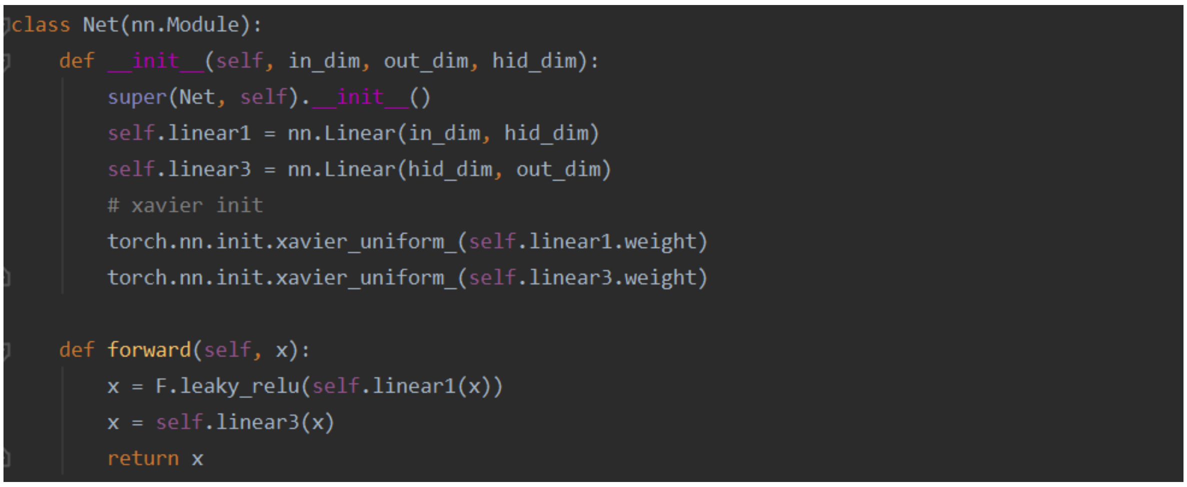
- **Reward:**
  - **-1 for each time step, until the goal position of 0.5 if reached.**

2. Procedure

Here I implement several DQNs, including basic DQN, Double DQN, Dueling DQN, priority experience replay(PER) DQN and their combinations.

2.1 Network

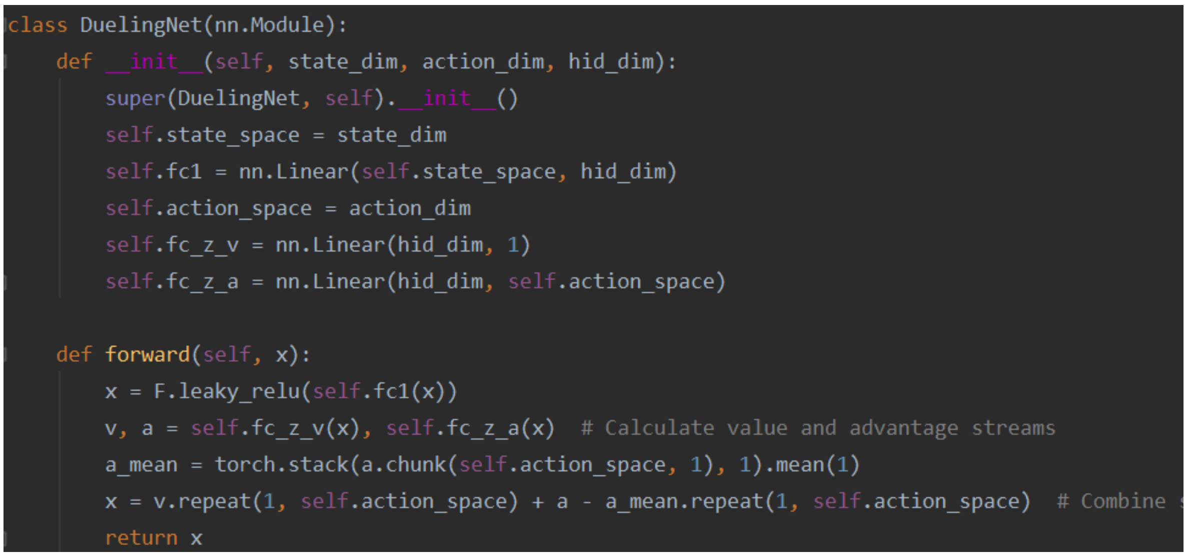
First, I implement the network of DQN uses as a independent class. The following code shows the basic network. Given that MountainCar is not very complex, this network is quit simple and only has one hidden layer.



```python
class Net(nn.Module):
    def __init__(self, in_dim, out_dim, hid_dim):
        super(Net, self).__init__()
        self.linear1 = nn.Linear(in_dim, hid_dim)
        self.linear3 = nn.Linear(hid_dim, out_dim)
        # xavier init
        torch.nn.init.xavier_uniform_(self.linear1.weight)
        torch.nn.init.xavier_uniform_(self.linear3.weight)

    def forward(self, x):
        x = F.leaky_relu(self.linear1(x))
        x = self.linear3(x)
        return x
```

I also implement Dueling DQN, which modifies the structure of network. The following code shows the the network of Dueling DQN. After a dense layer, features go to two branches, one for computing value function and the other one for computing action advantage function. Then add the two branches together and we get the final output.

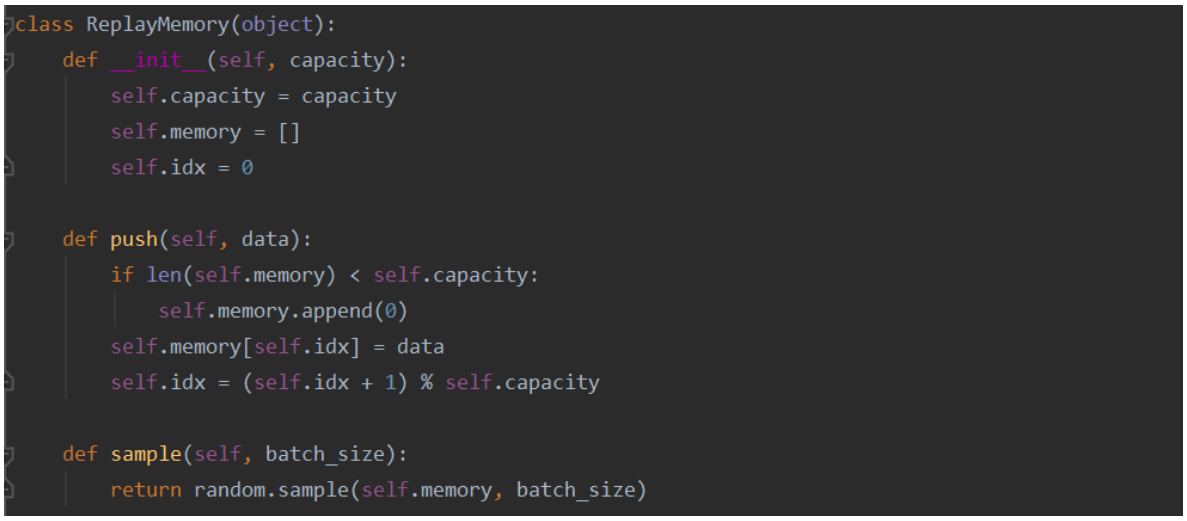


```python
class DuelingNet(nn.Module):
    def __init__(self, state_dim, action_dim, hid_dim):
        super(DuelingNet, self).__init__()
        self.state_space = state_dim
        self.fc1 = nn.Linear(self.state_space, hid_dim)
        self.action_space = action_dim
        self.fc_z_v = nn.Linear(hid_dim, 1)
        self.fc_z_a = nn.Linear(hid_dim, self.action_space)

    def forward(self, x):
        x = F.leaky_relu(self.fc1(x))
        v, a = self.fc_z_v(x), self.fc_z_a(x)  # Calculate value and advantage streams
        a_mean = torch.stack(a.chunk(self.action_space, 1), 1).mean(1)
        x = v.repeat(1, self.action_space) + a - a_mean.repeat(1, self.action_space)  # Combine
        return x
```

2.2 Replay Buffer & Priority Replay Buffer

The implementation of replay buffer is quiet simple. If it is full, replace the oldest data. If not, just add data to the end of the array. When sampling, just randomly pick some.

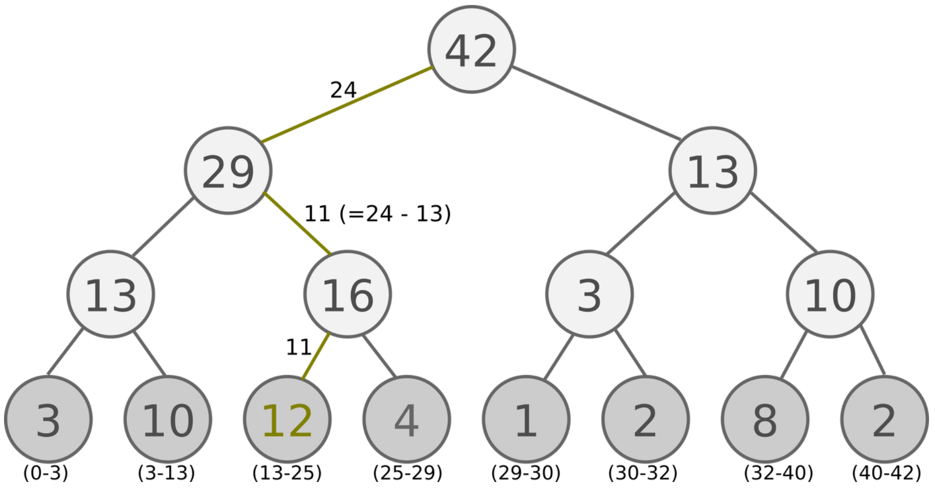


```python
class ReplayMemory(object):
    def __init__(self, capacity):
        self.capacity = capacity
        self.memory = []
        self.idx = 0

    def push(self, data):
        if len(self.memory) < self.capacity:
            self.memory.append(0)
        self.memory[self.idx] = data
        self.idx = (self.idx + 1) % self.capacity

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)
```

The implementation of priority replay buffer is based on Sum Tree. Sum Tree is a Binary Tree where the value of a node is equal to sum of the nodes present in its left subtree and right subtree. Here, we put priority on leaf node and associate it to our data. The root value x is the sum of all priority. Then we sample uniformly in range[0, x], and the larger the priority is, the more likely associated transition will be chosen.



In priority replay buffer, I slightly modify the TD error and use it as priority. It's important to point out that my implementation of PER DQN is a simplified version. I just use Sum Tree to perform weighted sampling. In original version, every experience is associated with a importance-sampling weight, which also used as a weight when we compute loss.

```
class PERMemory:   # stored as ( s, a, r, s_ ) in SumTree
    e = 0.01
    a = 0.6

    def __init__(self, capacity):
        self.tree = SumTree(capacity)

    def _getPriority(self, error):
        return (error + self.e) ** self.a

    def push(self, error, sample):
        p = self._getPriority(error)
        self.tree.add(p, sample)
```

Next, to sample a batch of size k, the range [0, total_priority] will be divided into k ranges. A value is uniformly sampled from each range.

```
def sample(self, n):
    batch_idx = []
    batch = []
    segment = self.tree.total() / n
    for i in range(n):
        a = segment * i
        b = segment * (i + 1)
        s = random.uniform(a, b)
        (idx, p, data) = self.tree.get(s)
        batch.append(data)
        batch_idx.append(idx)
    return batch
```

More details can be found in memory.py

2.3 DQN

The initialization of this class need many parameters. `state_dim` and `action_dim` are the dimension of state space and action space. `hidden_dim` is how many hidden units evaluate net and target net have. `memory_size` is the max capacity of replay buffer. `target_update` is the frequency of target net updating. `use_per`, `use_double` and `use_dueling` control whether model use those method or not.

```
def __init__(self, state_dim, action_dim, gamma=0.95, batch_size=128, lr=1e-3,
             epsilon=0.1, memory_size=1000, target_update=200, hidden_dim=100,
             use_per=False, use_double=False, use_dueling=False):
```

The following code shows how `use_per` and `use_dueling` affect the initialization of model.

```
self.replay_buffer = ReplayMemory(memory_size) if not use_per else PERMemory(memory_size)
# initialize net
if use_dueling:                             # whether use Dueling-DQN
    self.eval_net = DuelingNet(state_dim, action_dim, hidden_dim)
    self.target_net = DuelingNet(state_dim, action_dim, hidden_dim)
else:
    self.eval_net = Net(state_dim, action_dim, hidden_dim)
    self.target_net = Net(state_dim, action_dim, hidden_dim)
self.optimizer = torch.optim.Adam(self.eval_net.parameters(), lr=self.learning_rate)
self.loss_func = nn.MSELoss()               # optimizer & loss function
```

Function `choose_action()` uses epsilon-greedy algorithm to choose action from results outputted by evaluate net. Action chosen by this function will be used to interact with environment. In this class, there is another function `choose_greedy_action()` , which uses greedy algorithm to choose actions.

```
def choose_action(self, state):
    # epsilon-greedy, used for train
    state = torch.unsqueeze(torch.tensor(state).float(), 0)
    if np.random.uniform() > self.epsilon - self.train_step_cnt*0.0002:
        actions_value = self.eval_net.forward(state)
        action = torch.argmax(actions_value, dim=1).numpy()[0]
    else:
        action = np.random.randint(0, self.action_num)
    return action
```

Function `get_td_error()` computes TD error, that is $R + \gamma max_a Q'(S', a) - Q(S, A)$. This function will be used when we use PER DQN because we use TD error as priority .

```
def get_td_error(self, packed):
    # calculate td_error as priority
    state = torch.unsqueeze(torch.tensor(packed.state).float(), 0)
    next_state = torch.unsqueeze(torch.tensor(packed.next_state).float(), 0)
    action = torch.unsqueeze(torch.tensor([packed.action]), 0)
    q_eval = self.eval_net.forward(state).gather(1, action)[0]
    q_next = self.target_net.forward(next_state)
    q_target = packed.reward + self.gamma * q_next.max(1)[0] * (1 - packed.done)
    td_error = abs((q_target - q_eval).item())
    return td_error
```

Last but not least, function `update()` is about how to update the network. First, update target net if it reaches the frequency of target net updating.

```python
# update target net
if self.train_step_cnt % self.target_update == 0:
    self.target_net.load_state_dict(self.eval_net.state_dict())
self.train_step_cnt += 1
```

Then, get mini-batch from replay buffer and convert its content to tensor. It is noteworthy that `Transition` is a namedtuple and its definition is `Transition = namedtuple('Transition', ('state', 'action', 'reward', 'next_state', 'done'))`

```python
# get batch data
batch = self.replay_buffer.sample(self.batch_size)
batch = Transition(*zip(*batch))
cur_states = torch.tensor(batch.state).float()
actions = torch.unsqueeze(torch.tensor(batch.action), 1)
rewards = torch.unsqueeze(torch.tensor(batch.reward), 1)
next_states = torch.tensor(batch.next_state).float()
done = torch.unsqueeze(torch.tensor(batch.done).float(), 1)
```

The following part is calculating loss.

If we use Double DQN, the target is $r_t + Q'(s_{t+1}, argmax_a Q(s_{t_1}, a))$, where Q is evaluate net and Q' is target net. If not, the target is $r_t + max_a Q'(s_{t+1}, a)$. As we only update evaluate net, here target net uses distach. Then, we will update evaluate net via gradient calculated by pytorch.

```python
# calculate loss
q_eval = self.eval_net(cur_states)
qa_eval = q_eval.gather(1, actions)            # shape (batch, action_dim)
q_next = self.target_net(next_states)          # don't calculate the gradient of q_next
if self.use_double:                            # whether use double DQN
    qa_next = q_next.gather(1, torch.unsqueeze(torch.argmax(q_eval, dim=1), 1)).detach()
else:
    qa_next = torch.unsqueeze(q_next.max(1)[0], 1).detach()
q_target = rewards + self.gamma * qa_next * (1 - done)
loss = self.loss_func(qa_eval, q_target)
```

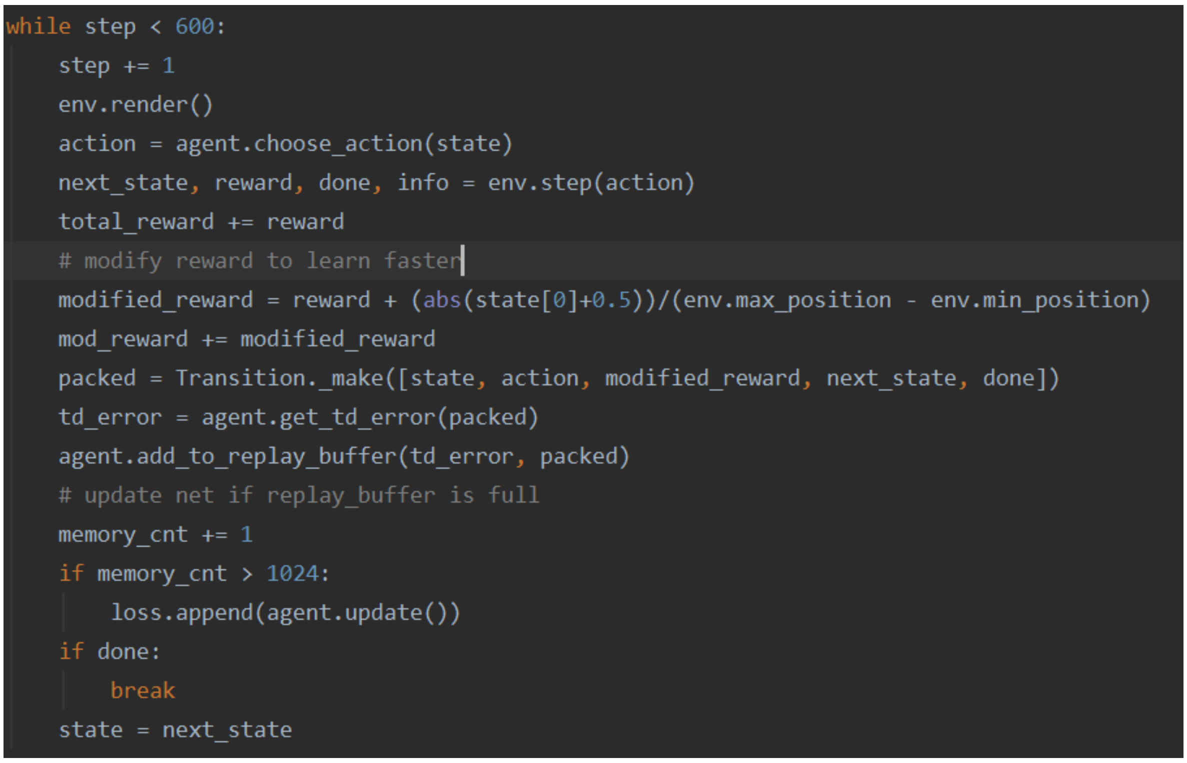
There are several simple functions that I don't mention. Function `add_to_replay_buffer()` will add transition to replay buffer. Function `load()` and `save()` are about how to load and save the model.

2.4 DQN train & test

When training, the number of episode will be inputted. The following code shows what we do in an episode. First, reset the environment and begin this episode. Agent chooses action by epsilon-greedy and environment returns what the action gets.

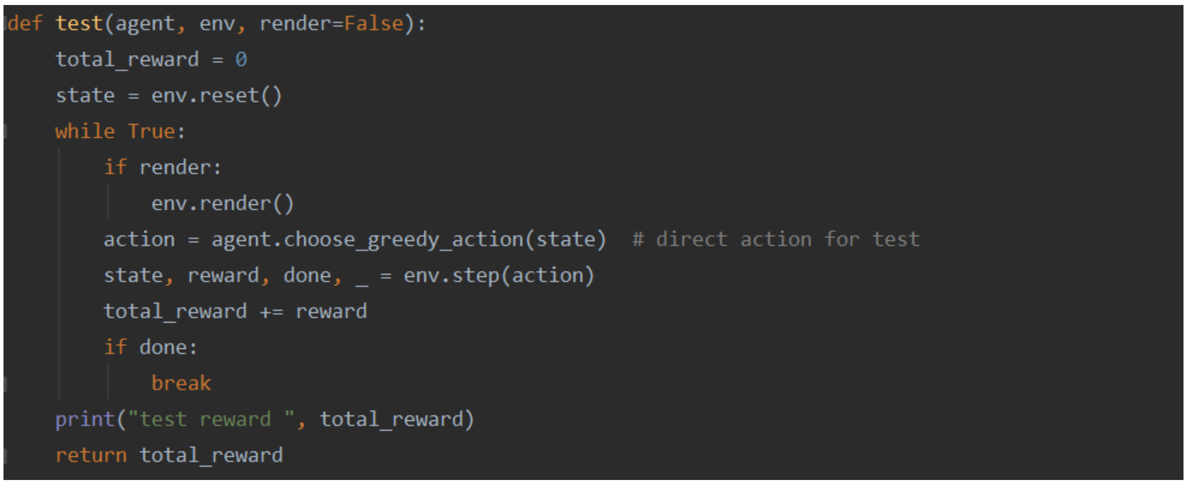
The reward of MountainCar is very sparse. It gets positive reward only when car reached the peak and gets -1 reward otherwise, which make it hard to train. So, I modify the reward and encourage car to reach higher. Later, I talk more about the effect of modifying reward in Part 4.

Then, pack all those we get to a namedtuple and add it to replay buffer. The update step will not begin until the replay buffer reaches 1024.



```python
while step < 600:
    step += 1
    env.render()
    action = agent.choose_action(state)
    next_state, reward, done, info = env.step(action)
    total_reward += reward
    # modify reward to learn faster
    modified_reward = reward + (abs(state[0]+0.5))/(env.max_position - env.min_position)
    mod_reward += modified_reward
    packed = Transition._make([state, action, modified_reward, next_state, done])
    td_error = agent.get_td_error(packed)
    agent.add_to_replay_buffer(td_error, packed)
    # update net if replay_buffer is full
    memory_cnt += 1
    if memory_cnt > 1024:
        loss.append(agent.update())
    if done:
        break
    state = next_state
```

Test step is more straightforward. We don't need to update net or add transition to replay buffer. Agent just chooses best action and perform it. The total reward will be returned.



```python
def test(agent, env, render=False):
    total_reward = 0
    state = env.reset()
    while True:
        if render:
            env.render()
        action = agent.choose_greedy_action(state)  # direct action for test
        state, reward, done, _ = env.step(action)
        total_reward += reward
        if done:
            break
    print("test reward ", total_reward)
    return total_reward
```

2.5 Other

The function `train_in_diffrent_setting()` in DQN.py will train model according to input setting , save model and results, and plot picture of loss and rewards.

Function `plot_data()` in Plotting.py is responsible for plotting figures.

3. Results

3.1 Parameters

Here are the main parameters I use in training and they remain the same when I try different DQN. Because `env = gym.envs.make("MountainCar-v0")` automatically sets the max step of an episode to 200. I use `env = gym.envs.make("MountainCar-v0").unwrapped` which allows infinite step per episode.

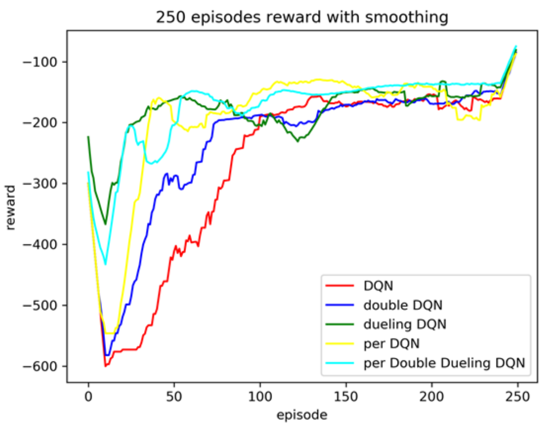
Parameter Name	Value	Parameter Name	Value
epsilon	0.1	batch size	256
memory size	20000	learning rate	0.001
gamma	1	target update step	100
max step of an episode	600	hidden unit	100

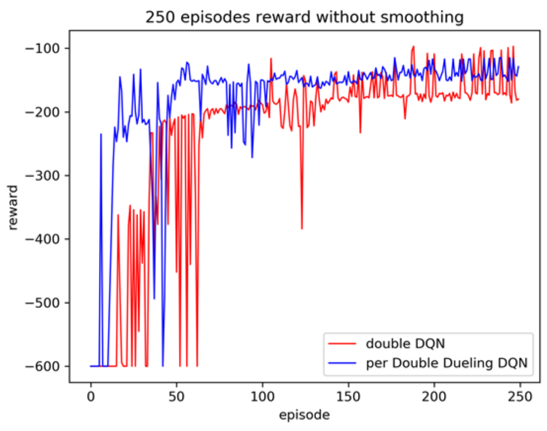
3.2 Results

I've tried five settings, basic DQN, double DQN, dueling DQN, PER DQN, and PER double Dueling DQN.

The left diagram shows smoothed rewards of these five setting. Because of smoothing method, the begin and end part of curve are distorted and should be ignored. The right diagram shows non-smoothed rewards. Given that non-smoothed rewards are complex and it's hard to discern five non-smoothed rewards curves in a single diagram, I only take double DQN and PER double Dueling DQN as an example.

As we can see in the left diagram, compared with basic DQN, other four DQNs do perform better and converge faster. PER double Dueling DQN performs best not only because of its fast converging speed, but also because it's more stable and gets higher reward. The left diagram, rewards without smoothing, makes it clearer that PER double Dueling DQN converges faster and has less fluctuation and gets higher rewards.





4. Discussion

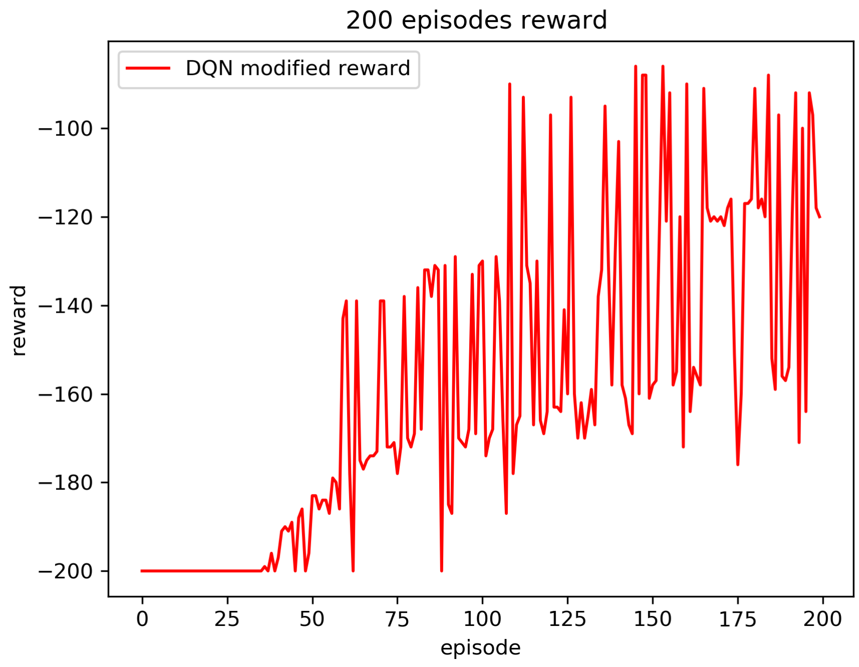
4.1 The Effect of Modifying Reward

When I use `env = gym.envs.make("MountainCar-v0")` , which set the max step of an episode to 200, I find that the original reward of MountainCar is very sparse. It only has negative reward -1 and it's hard for car to reach the peak randomly. This makes it hard to train because the possibility of get positive reward is very small and we may be stuck in all those negative rewards.

So, we need to modify the rewards and encourage car to reach higher. The ideal reward should meet the following demands

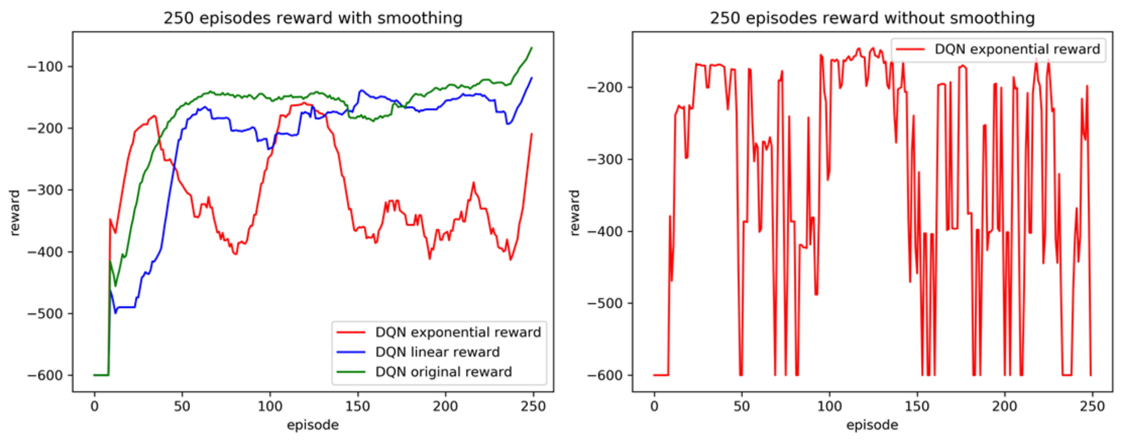
- The higher the position is, the larger the reward is.
- Make car reach the peak as soon as possible.

I find that modifying the reward to $reward + e^{|state[0]+0.5|}$ works very well. If I modify reward, car can reach the peak within 40 episodes. If not, the car can't reach the peak even if I train 200 episodes.



But when I use `env = gym.envs.make("MountainCar-v0").unwrapped` and set larger max step of an episode, I find the reward modification has little impact on results or even has negative impact on final results. Here, linear modified reward is $reward + \frac{state[0]+0.5}{max\_pos - min\_pos}$ and exponential modified reward is $reward + e^{|state[0]+0.5|}$.

The following diagram shows how reward modification affects the final rewards we get. Under the current setting, it's unnecessary to modify the reward, since the performance of linear modification is about the same with original reward(non-modified) and the performance of exponential modification is even worse.



The reason behind the poor performance of exponential modification may be that the modified rewards are all positive. Agent tends to get more rewards but reaching peak as soon as possible may not bring more rewards. Despite agent get less original rewards, it may get more modified rewards. So it doesn't converge and has large fluctuations.

4.2 What if swap Q and Q' in double DQN

In Double DQN, I try to swap the position of target net and evaluate net and expect to get similar results. That is I use $r_t + Q(s_{t+1}, argmax_a Q'(s_{t_1}, a))$ rather than $r_t + Q'(s_{t+1}, argmax_a Q(s_{t_1}, a))$, where Q is evaluate net and Q' is target net. But I find it can't even converge.

Finally, I figure out that if we swap them, the target is calculated by evaluate net, which is ever changing. Actually that's what we want to avoid and the reason we use a fixed target net.