CS489 Assignment 4 Report

517030910214 Hongzhou Liu

0. Introduction

In this assignment, I implemented DQN and it's improvement Double DQN in MountainCar environment. In MountainCar, there's a car on a one-dimension track. The goal is to drive up the mountain on the right. Due to the limitation of engine, the car have to drive back and forth to build up momentum in order to reach the goal. There are three kinds of actions:

- 0: push left
- 1: do nothing
- 2: push right

Each time the car choose an action, it will gain a reward of -1. The state observed by agent from the environment consists of current horizontal position and velocity. The goal is at the position of 0.5. An episode ends when the car reaches the goal or the car takes 200 actions.

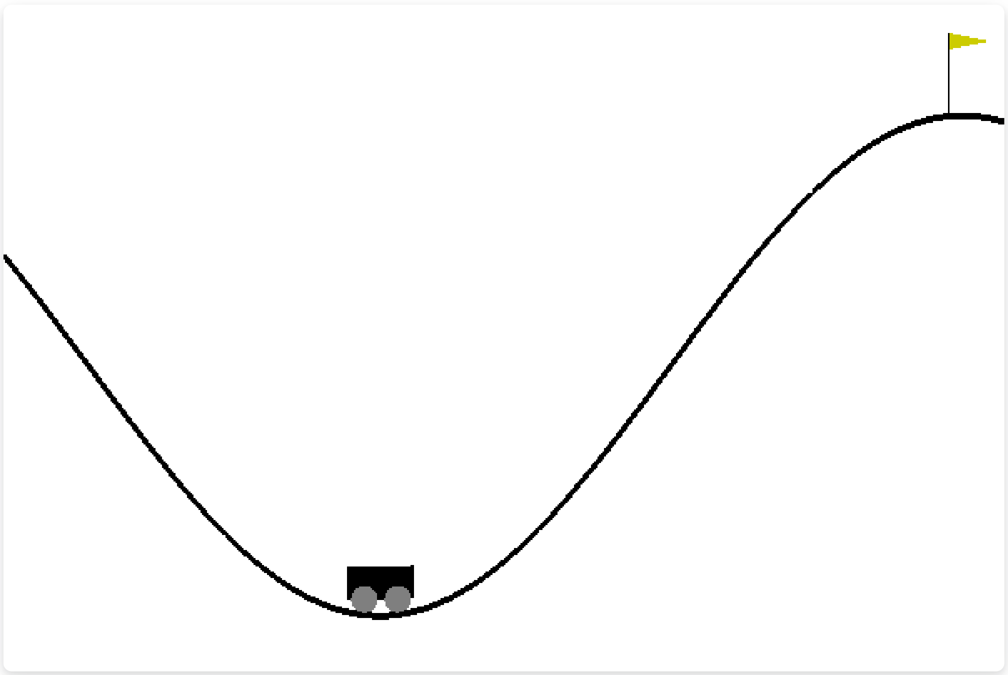

Fig.1 Mountain Car

Environment:

- Ubuntu 18.04 LTS
- Python 3.7.7

1. DQN

1.1 Implementation

1.1.1 Algorithm

Deep Q-Network is a milestone of Reinforcement Learning. It introduced neural network into the original Q-Learning algorithm and largely strengthened the ability of Q-Learning algorithm. The two features of DQN is:

- Fixed target net
- Replay buffer

The DQN algorithm starts with policy network Q and target network $\hat{Q}$, which is the same. In each episode, the agent interact with the environment using the given state s_t and an action a_t based on Q epsilon greedily. The algorithm stores the obtained reward r_t and next state s_{t+1} together with s_t and a_t into the buffer as (s_t, a_t, r_t, s_{t+1}). Then, it samples a batch from the buffer and estimate the target as

$$y = r + \gamma \max_a \hat{Q}(s, a)$$

Then, we will optimize the mean square error between the estimation and Q value as

$$\min(y - Q(s, a))^2$$

The algorithm fixes the target network $\hat{Q}$ and updates the policy network Q. And it will update $\hat{Q}$ by setting $\hat{Q} := Q$ every C steps.

1.1.2 Network

We will utilize `torch.nn` to construct our neural network for DQN.

```python
class DQN(nn.Module):
  def __init__(self, inputSize, hiddenSize, outputSize):
    super(DQN, self).__init__()
    self.fc1 = nn.Linear(inputSize, hiddenSize)
    self.fc2 = nn.Linear(hiddenSize, hiddenSize)
    self.fc3 = nn.Linear(hiddenSize, outputSize)

  def forward(self, x):
    x = F.leaky_relu(self.fc1(x))
    x = F.leaky_relu(self.fc2(x))
    x = F.leaky_relu(self.fc3(x))
    return x
```

Here, we construct a network with one hidden layer. And in this assignment, the size of the hidden layer is 256. We use leaky ReLU to avoid gradient vanishing. The input is state and the output is $Q(s, a)$ for each action a.

1.1.3 Replay Buffer

The replay buffer is like memory in our brain. It stores some recent experience in tuple (s_t, a_t, r_t, s_{t+1}). The size of the buffer is fixed, meaning it will forget as it interacts with the environment.

```python
class ReplayBuffer:
  def __init__(self, capacity):
    self.capacity = capacity
    self.buffer = []
    self.position = 0 #for recursive array

  def push(self, *args):
    if len(self.buffer) < self.capacity:
      self.buffer.append(None)
    self.buffer[self.position] = Exp(*args)
    self.position = (self.position + 1) % self.capacity

    def sample(self, batchSize):
        return random.sample(self.buffer, batchSize)
```

We can implement the buffer as a recursive array. If the buffer is full, the newly coming experiences will replace the old ones. Also, we can use `sample` method to sample a batch of experiences.

1.1.4 Optimize Object Function

The kernel of Deep Q-Learning is to optimize the mean-squared error between Q and the estimation y. When implementing this part, some coding tricks are used.

```python
    def optimzeDQN(self, buf):
        if len(buf) < self.batchSize:
            print("Can't fetch enough exp!")
            return
        exps = buf.sample(self.batchSize)
        batch = Exp(*zip(*exps))  # batch => Exp of batch
        stateBatch = torch.cat(batch.state) # batchSize * stateSpace.shape[0]
        actionBatch = torch.cat(batch.action) # batchSize * 1
        rewardBatch = torch.cat(batch.reward) # batchSize * 1
        nextStateBatch = torch.cat(batch.nextState) # batchSize * stateSpace.shape[0]
        doneBatch = torch.cat(batch.done) # batchSize * 1

        Q = self.net(stateBatch).gather(1, actionBatch)  # get Q(s_t, a)
        targetQ = self.targetNet(nextStateBatch).max(1)[0].view(-1, 1) # max_a Q'(s_t+1, a)
        y = (targetQ * self.gamma) * doneBatch + rewardBatch
        loss = F.mse_loss(Q, y)
        estQMean = np.mean(targetQ.detach().cpu().numpy())

        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()
        return loss.item(), estQMean
```

After sampling a batch of experiences from buffer, we should separate the batches of s_t, a_t, r_t and s_{t+1} from the experience batch. Here, `zip()` function and `*` operator helps us reshape the batch and construct a new named tuple.

When calculating Q, `gather(1, actionBatch)` helps to find $Q(s, a)$ according to `actionBatch` as indices. And `max(1)[0]` to get $\max_a \hat{Q}(s, a)$. Notice that y is slightly different when s_t is the last state of an episode. `doneBatch` consists of flags indicating if s_t is of such state. We multiply it on $\gamma \max_a \hat{Q}(s, a)$ to make sure $y = r_t$ if s_t is the last state.

1.1.5 Learning

```python
def DeepQLearning(self, env, buf, episodeNum, ifDecay=True):
    totalN = 0
    scores = []
    steps = []
    losses = []
```

```python
        Qmeans = []
        self.targetNet.load_state_dict(self.net.state_dict())
        for i in range(episodeNum):
            score = 0.0
            rewardDecay = 1.0
            state = env.reset()
            for t in count():
                env.render()
                action = self.getAction(torch.tensor(state.reshape(1, self.stateSize), device=device, dtype=torch.float))

                nextState, reward, done, _ = env.step(action.item())

                buf.push(torch.tensor(state.reshape(1, self.stateSize), device=device, dtype=torch.float),
                         action,
                         torch.tensor([[reward]], device=device, dtype=torch.float),
                         torch.tensor(nextState.reshape(1, self.stateSize), device=device, dtype=torch.float),
                         torch.tensor([[not done]], device=device, dtype=torch.long))
                state = nextState
                score += rewardDecay * reward
                rewardDecay *= self.gamma

                loss, mean = self.optimzeDQN(buf)
                losses.append(loss)
                Qmeans.append(mean)

                if ifDecay:
                    self.epsilonDecay(totalN)
                totalN += 1

                if totalN % self.updateStride == 0:
                    self.targetNet.load_state_dict(self.net.state_dict())

                if done or t + 1 >= env.max_episode_steps:
                    scores.append(score)
                    steps.append(t + 1)
                    print("Episode %d ended after %d timesteps with score %f, epsilon=%f" % (i + 1, t + 1, score, self.epsilon))
                    break
```

As seen, in the `for` loop, the DQN algorithm is properly implemented. And `self.getAction` implements epsilon greedy.

1.1.6 Some Discussions

Epsilon Decay

```python
def epsilonDecay(self, N):
    if (N >= 50000):
        self.epsilon = self.EPS_END + (self.EPS_START - self.EPS_END) * math.exp(-1.0 * (N - 50000) / self.EPS_DECAY)
```

Customized Reward

```python
def step(self, action):
    state, reward, done, info = self.env.step(action)
    if done:
      reward = 1000
    else:
      reward = (self.height(state[0]) - 0.1) * 10 + 50 * state[1] * state[1]
    return state, reward, done, info

  def height(self, xs):
    return np.sin(3 * xs) * .45 + .55
```

1.2 Result

2. Double DQN

2.1 Implementation

The double DQN is an improvement of natural DQN algorithm. It modified the method of estimating TD target and helps reduce overestimating of Q value greatly. The modified estimating of TD target is

$$y = r + \gamma \hat{Q}(s, \arg\max_a Q(s, a))$$

Now,

- If Q overestimate a, then $\hat{Q}$ will give it a proper value
- If $\hat{Q}$ overestimate a, then a will not be selected

Thus, we can avoid overestimation in this way.

To implement Double DQN, we can just modify the `optimizeDQN` function and change the part of calculating Q and y as

```python
Q = self.net(stateBatch).gather(1, actionBatch)  # get Q(s, a)
    targetAction = self.net(nextStateBatch).max(1)[1].view(-1, 1)  # get argmax Q'(s_t+1, a)
    targetQ = self.targetNet(nextStateBatch).gather(1, targetAction)
    y = (targetQ * self.gamma) * doneBatch + rewardBatch
    loss = F.mse_loss(Q, y)
    estQMean = np.mean(targetQ.detach().cpu().numpy())
```

As seen, now we use $\arg\max \hat{Q}(s_{t+1}, a)$ as the action batch when estimating `targetQ`. Here, `max(1)[1]` is used to get the index of maximum value in a row of a tensor in two dimensions.

2.2 Result & Discussion