Homework 2

蔡云龙

21215068

Task 1 Implementing DQN

在Atari中实现最基本的DQN算法。Atari环境会返回大小为$(height \times width \times channels)$图片作为智能体的观测。而强化学习的观测一般是采用一维向量的形式，通常只需要利用全连接神经网络，算法就可以收敛。由于Atari环境返回的是图片，因此需要用CNN对图像信息预处理，并将处理完的信息reshape成一维向量的形式作为强化学习神经网络的输入。强化学习算法的评判标准主要有reward收敛值大小、reward收敛速度等。

DQN 和 Dueling DQN的网络结构

```
class QNetwork(nn.Module):
    def __init__(self, input_size, hidden_size, output_size, dueling=False):
        super(QNetwork, self).__init__()
        ###################
        # YOUR CODE HERE #
        ###################
        self.dueling = dueling
        self.conv_layer_1 = nn.Conv2d(input_size[0], 32, kernel_size=8, stride=4)
        self.conv_layer_2 = nn.Conv2d(32, 64, kernel_size=4, stride=2)
        self.conv_layer_3 = nn.Conv2d(64, 64, kernel_size=3, stride=1)
        self.fc_layer = nn.Linear(7 * 7 * 64, hidden_size)

        if self.dueling:
            print("Using Dueling DQN")
            # V(s) value of the state
            self.dueling_value = nn.Linear(hidden_size, 1)
            # Q(s,a) Q values of the state-action combination
            self.dueling_action = nn.Linear(hidden_size, output_size)
        else:
            self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, inputs):
        ###################
        # YOUR CODE HERE #
        ###################
        x = F.relu(self.conv_layer_1(inputs))
        x = F.relu(self.conv_layer_2(x))
        x = F.relu(self.conv_layer_3(x))
        x = F.relu(self.fc_layer(x.view(x.size(0), -1)))

        if self.dueling:
            value = self.dueling_value(x)
            advantage = self.dueling_action(x)
            q = value + (advantage - advantage.mean(1, keepdim=True))
            return q
        else:
            return self.fc(x)
```

```
    def __call__(self, inputs):
        return self.forward(inputs)
```

主要运行参数说明:

```
--env_name  # 环境名
--dueling   # 使用dueling DQN网络结构
--buffer_size  # 设置经验池大小
--reward  # 设置目标平均奖励值，达到此值时结束
--check_path  # 测试时指定保存的模型的位置
--target_update_freq  # target网络更新的频率
```

其他一些参数写在了`agent_dqn.py`的初始化函数内

AgentDQN主要的方法说明:

```
# agent_dqn.py
# 初始化函数，初始化环境最初状态、参数列表、gpu设备、训练网络和目标网络、经验池、epsilon、优化器等
def __init__(self, env, args):

# 通过训练网络得到动作值的函数，传入observation，返回动作值
def make_action(self, observation, test=True):

# 根据动作值生成规则得到动作值，并与环境交互得到一步操作后的参数，放入经验池，更新状态和奖励，并返回奖励
def step_env(self, epsilon=0.0):

# 传入从经验池采样到的batch，通过计算得到的状态动作值和下一个状态值计算期望的状态动作值，并与状态动作值求loss，进而反向传播
def train(self, batch):

# 训练的主函数，一个大循环，其中根据步数（帧数）更新探索的概率值：epsilon，采用线性变化的形式从initial_eps降到final_eps；调用step_env函数得到单步奖励并计算最后100步的平均奖励，和记录的最佳的平均奖励，当有更好的平均奖励时更新，直到满足设定的 --reward 参数；在参数target_update_freq整除步数是更新目标网络的参数为当前网络的参数；最后根据计算的loss反向传播。最后把训练数据绘图保存
def run(self):


# 下面两个函数用来测试模型时使用，在init_game_setting函数中读取参数中的check_path, 也即模型路径，并在test函数中进行测试
def init_game_setting(self):

def test(self):
```

DQN，Double DQN，Dueling DQN算法介绍:

DQN 和Dueling DQN的区别是网络结构的区别，dueling DQN除了输出动作取值向量还输出值，也就是有两个输出层，并且forward中计算返回值也不相同。

Q Network和dueling Q Network的网络结构如下：

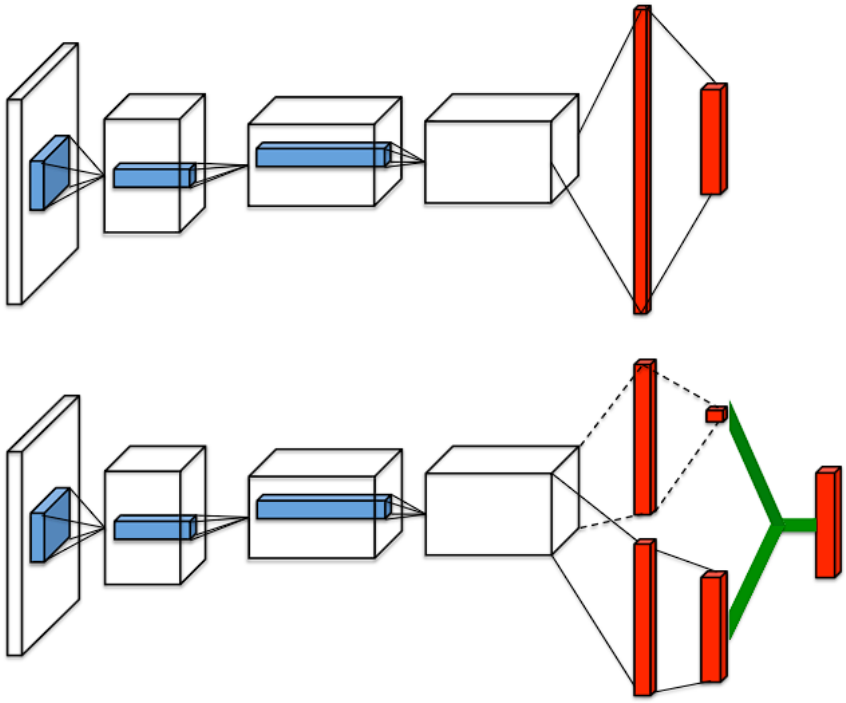


Figure 1. A popular single stream Q-network (**top**) and the dueling Q-network (**bottom**). The dueling network has two streams to separately estimate (scalar) state-value and the advantages for each action; the green output module implements equation (9) to combine them. Both networks output Q-values for each action.

DQN与Double DQN的区别主要在于求Loss的过程不一样：首先两者都通过当前训练的网络输入状态得到状态Q值$Q(s,a)$。然后DQN中，通过目标网络输入下一个状态向量得到下一个状态的值即$Q(s',a)$并取最大值，最后得到期望奖励为：$R+\gamma \max Q(s',a)$，将其与前面的$Q(s,a)$求MSELoss得到loss，所以最终loss计算形如$R+\gamma \max Q(s',a)-Q(s,a)$。而Double DQN中，首先利用当前网络输入下一个状态并取max得到下一个状态要采取的动作向量$a'=argmaxQ(s',a)$，基于这个动作向量和下一个状态向量输入到目标网络得到$Q(s',a)$，进而得到期望奖励为：$R+\gamma Q(s',a')$，最终Loss形如$R+\gamma Q(s',a')-Q(s,a)$, 具体区别看`train`函数中的`next_action_vector`变量的计算方式（即a'），下面为两种方式时此变量的计算方法：

```
# def train(self, batch):

# DQN
# compute the actual values we got for those transitions
next_state_values = self.target_QNet(next_states_vector).max(1)[0]
# make sure future values aren't being considered for end states
next_state_values[done_mask] = 0.0
next_state_values = next_state_values.detach()
# DQN Finish

# Double DQN
# 由当前网络得到下一个状态输入进去后得到的下一个动作向量，用来目标网络得到label
_, next_action_vector = self.current_QNet(next_states_vector).max(1)
# 同时由目标网络也得到一个作为label
next_state_values = self.target_QNet(next_states_vector)\
.gather(1, next_action_vector.unsqueeze(-1).type(torch.int64)).squeeze(-1)
```

```
next_state_values[done_mask] = 0.0
next_state_values = next_state_values.detach()
# Double DQN Finish
```

1. **(coding)** 在Atari `PongNoFrameskip-v4` 环境中实现DQN、Double DQN、Dueling DQN算法。环境最终的reward至少收敛至17.0。

   下面的实验是基于Dueling DQN的网络和DQN的loss计算方法进行的实验。

   采用的参数:

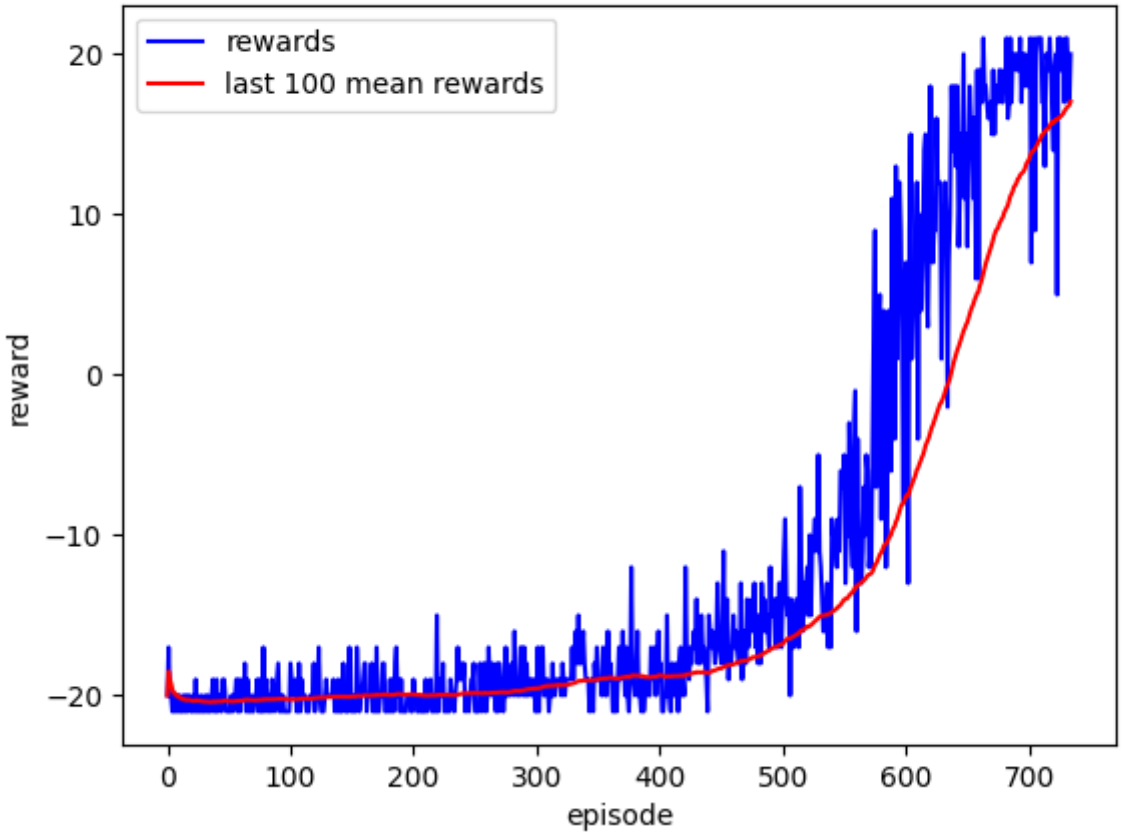
   ```
   lr = 0.0001; buffer_size = 10000; target_update_freq = 1000; reward = 17;
   ```

   执行的命令:

   ```
   python main.py --train_dqn True --dueling --reward 17
   ```

   环境名及其他参数使用默认值。在平均奖励大于等于17时停止

   训练结果:

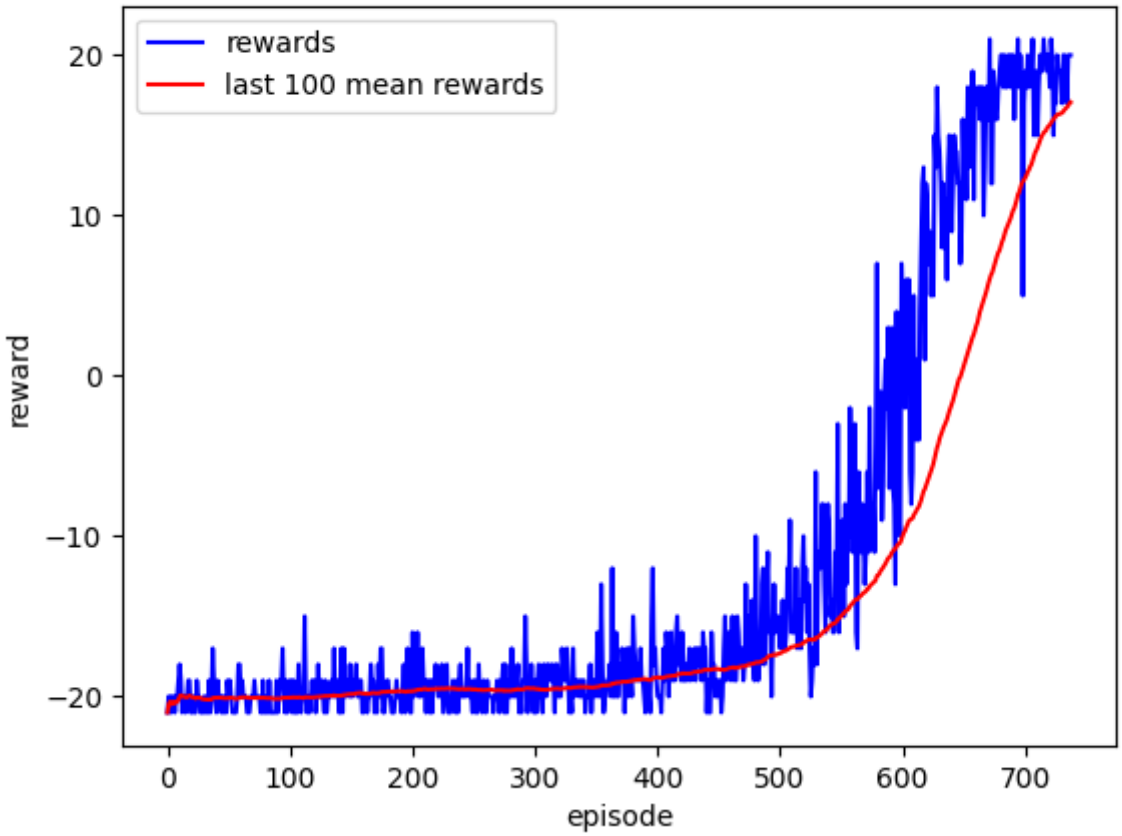


   log文件记录相应的输出记录，最后部分的结果如下（**后面的实验的log见相应文件夹，不再贴出**）：

```
episode: 721 | reward: 14.0 | mean_reward: 15.72 | epsilon: 0.02 | speed: 87.97 frames/s
episode: 722 | reward: 19.0 | mean_reward: 15.81 | epsilon: 0.02 | speed: 86.65 frames/s
episode: 723 | reward: 20.0 | mean_reward: 15.94 | epsilon: 0.02 | speed: 88.10 frames/s
episode: 724 | reward: 5.0 | mean_reward: 15.90 | epsilon: 0.02 | speed: 87.94 frames/s
episode: 725 | reward: 21.0 | mean_reward: 16.02 | epsilon: 0.02 | speed: 87.47 frames/s
episode: 726 | reward: 19.0 | mean_reward: 16.05 | epsilon: 0.02 | speed: 87.94 frames/s
episode: 727 | reward: 21.0 | mean_reward: 16.14 | epsilon: 0.02 | speed: 86.72 frames/s
episode: 728 | reward: 19.0 | mean_reward: 16.21 | epsilon: 0.02 | speed: 87.37 frames/s
episode: 729 | reward: 20.0 | mean_reward: 16.29 | epsilon: 0.02 | speed: 87.58 frames/s
episode: 730 | reward: 17.0 | mean_reward: 16.45 | epsilon: 0.02 | speed: 87.87 frames/s
episode: 731 | reward: 21.0 | mean_reward: 16.56 | epsilon: 0.02 | speed: 87.58 frames/s
episode: 732 | reward: 21.0 | mean_reward: 16.68 | epsilon: 0.02 | speed: 87.09 frames/s
episode: 733 | reward: 19.0 | mean_reward: 16.75 | epsilon: 0.02 | speed: 87.81 frames/s
episode: 734 | reward: 17.0 | mean_reward: 16.83 | epsilon: 0.02 | speed: 84.67 frames/s
episode: 735 | reward: 20.0 | mean_reward: 17.05 | epsilon: 0.02 | speed: 88.56 frames/s
```

下面同样使用Dueling DQN网络结构，但是基于Double DQN的loss计算方法进行实验：

参数和命令同上，手动修改`train`函数中的计算方式来运行，得到结果：



下面读取保存的模型文件对Pong游戏进行一轮测试。**(测试部分在相应的agent内的test函数中，未使用给的`test.py`文件)**

测试结果：

使用命令：

```
python main.py --test_dqn True --dueling --check_path .\models\dqn_pong\dqn_PongNoFrameskip-v4_best.dat
```

结果：

```
D:\Python\Python37\lib\site-packages\gym\envs\registration.py:14: PkgResourcesDeprecationWarning: Parameters to load are deprecated.  Call .resolve and .require separately.
  result = entry_point.load(False)
cuda:0
Using Dueling DQN
QNetwork(
  (conv_layer_1): Conv2d(4, 32, kernel_size=(8, 8), stride=(4, 4))
  (conv_layer_2): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2))
  (conv_layer_3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1))
  (fc_layer): Linear(in_features=3136, out_features=512, bias=True)
  (dueling_value): Linear(in_features=512, out_features=1, bias=True)
  (dueling_action): Linear(in_features=512, out_features=6, bias=True)
)
Using Dueling DQN
args: Namespace(batch_size=32, buffer_size=10000, check_path='.\\models\\dqn_pong\\dqn_PongNoFrameskip-v4_best.dat', dueling=True, env_name='PongNoFrameskip-v4', gamma=0.99, grad_norm_clip=10, hidden_size=512, learning_freq=1, lr=0.0001, n_frames=400000, reward=17, seed=11037, target_update_freq=1000, test=False, test_dqn=True, train_dqn=False, train_pg=False, use_cuda=True)
load dict successfully
D:\Python\Python37\lib\site-packages\pyglet\image\codecs\wic.py:434: UserWarning: [WinError -2147417850] 无法在设置线程模式后对其加以更改。
  warnings.warn(str(err))
Test Reward is:  21.0
```

使用double DQN的测试结果不再赘述（只是训练过程的loss计算方式不同，训练的模型对结果的影响差异不大）

2. **(coding)**在Atari `BreakoutNoFrameskip-v4`环境中实现DQN、Double DQN、Dueling DQN算法。环境最终的reward至少收敛至200.0。

   使用的参数：

   ```
   buffer_size = 1000000; target_update_freq = 10000; lr = 0.00025;
   initial_eps = 1.0; final_eps = 0.02; 经过1000000步从1.0线性降到0.02;
   ```

   执行的命令：

   ```
   python main.py --train_dqn True --dueling --env_name BreakoutNoFrameskip-v4 --reward 200 --buffer_size 1000000 --target_update_freq 10000 --lr 0.00025
   ```

   但是跑了25430轮，平均奖励也才38不到。

```
Best mean reward updated from 36.770 -> 36.780 and model saved
episode: 25419 | reward: 39.0 | mean_reward: 37.02 | epsilon: 0.02 | speed: 11.75 frames/s
Best mean reward updated from 36.780 -> 37.020 and model saved
episode: 25420 | reward: 24.0 | mean_reward: 36.89 | epsilon: 0.02 | speed: 11.95 frames/s
episode: 25421 | reward: 44.0 | mean_reward: 37.00 | epsilon: 0.02 | speed: 11.75 frames/s
episode: 25422 | reward: 26.0 | mean_reward: 36.92 | epsilon: 0.02 | speed: 11.63 frames/s
episode: 25423 | reward: 36.0 | mean_reward: 37.09 | epsilon: 0.02 | speed: 11.55 frames/s
Best mean reward updated from 37.020 -> 37.090 and model saved
episode: 25424 | reward: 47.0 | mean_reward: 37.41 | epsilon: 0.02 | speed: 11.62 frames/s
Best mean reward updated from 37.090 -> 37.410 and model saved
episode: 25425 | reward: 33.0 | mean_reward: 37.21 | epsilon: 0.02 | speed: 11.81 frames/s
episode: 25426 | reward: 16.0 | mean_reward: 37.19 | epsilon: 0.02 | speed: 11.71 frames/s
episode: 25427 | reward: 51.0 | mean_reward: 37.63 | epsilon: 0.02 | speed: 11.75 frames/s
Best mean reward updated from 37.410 -> 37.630 and model saved
episode: 25428 | reward: 40.0 | mean_reward: 37.55 | epsilon: 0.02 | speed: 11.88 frames/s
episode: 25429 | reward: 41.0 | mean_reward: 37.53 | epsilon: 0.02 | speed: 11.91 frames/s
episode: 25430 | reward: 45.0 | mean_reward: 37.70 | epsilon: 0.02 | speed: 11.74 frames/s
Best mean reward updated from 37.630 -> 37.700 and model saved
```

Task 2 Implementing Policy Gradient

网络结构

```
# 策略网络的结构：
class PGNetwork(nn.Module):
    def __init__(self, input_size=4, hidden_size=16, output_size=2):
        super(PGNetwork, self).__init__()
        ##################
        # YOUR CODE HERE #
        ##################
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, output_size)

    def forward(self, inputs):
        ##################
        # YOUR CODE HERE #
        ##################
        x = F.relu(self.fc1(inputs))
        x = self.fc2(x)
        x = F.softmax(x, dim=1)
        return x

# 当使用 baseline 方法时，还需要一个网络来生成状态值函数作为baseline，因此还需要一个网络：
class StateValueNetwork(nn.Module):

    def __init__(self, input_size=4, hidden_size=16):
        super(StateValueNetwork, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, 1)

    def forward(self, inputs):
        x = self.fc1(inputs)
        x = F.relu(x)
        state_value = self.fc2(x)
        return state_value
```

两个网络都是由两个全连接层构成的简单网络，输出维度不同。

AgentPG主要方法说明：

```
# agent_pg.py
# 初始化函数，初始化环境最初状态、参数列表、gpu设备、策略网络和状态值函数网络、优化器等
def __init__(self, env, args):

# make_action函数，输入状态，根据策略网络得到动作的概率，得到分布并采样得到动作以及选用该动作时的概率的对数值
def make_action(self, observation, test=False):

# get_discounted_rewards函数，根据一个轨迹的每步奖励值计算每步累计折扣回报列表并返回
def get_discounted_rewards(self, rewards):

# train_policy_net根据策略优化器、累计折扣回报（无baseline，形参为deltas）和概率的负对数值计算loss并优化网络；有baseline时需使用累计折扣回报与状态值函数的差作为参数（即deltas）
def train_policy_net(self, deltas, log_probs):

# train_state_value_net函数根据累计折扣回报和状态值计算loss来优化状态值网络
```

```
def train_state_value_net(self, discount_rewards, state_vals):

# run函数为训练主体函数，init_game_setting为测试时读取模型的初始化函数，test为测试函数
def run(self):
def init_game_setting(self):
def test(self):
```

在`cartpole`中实现policy gradient及其改进算法。策略梯度更新公式如下：

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s)Q^{\pi_\theta(s,a)}]$$

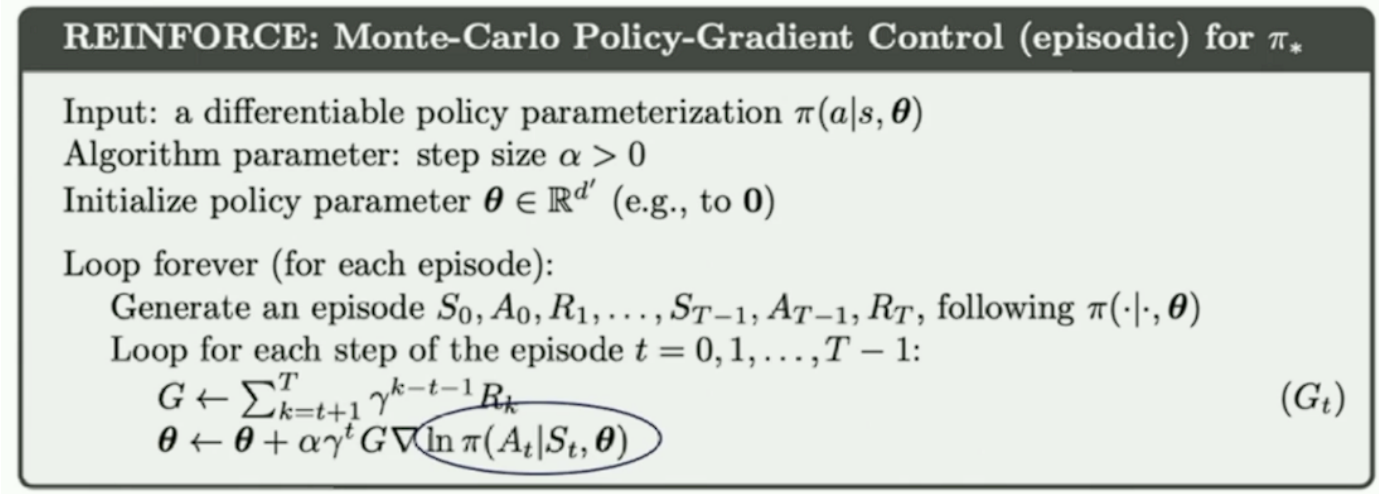
1. **(coding)** 在`cartpole`环境中实现基本的REINFORCE算法，即使用蒙特卡洛采样G_t作为$Q^{\pi_\theta(s,a)}$的无偏估计，更新公式如下：

$$J(\theta) = \frac{1}{\sum T_i}\sum_{i=1}^{|D|}\sum_{t=1}^{T_i}[log\pi_\theta(a_t^i|s_t^i)G_t^i],$$

其中，$\tau^i = (s_0^i, a_0^i, r_0^i, s_1^i, \ldots, s_{T_i}^i, a_{T_i}^i, r_{T_i}^i)$，$D$是在环境中执行策略$\pi_\theta$产生的所有轨迹的集合。最终算法性能的评判标准：环境最终的reward至少收敛至180.0。

算法说明：

REINFFORCE算法是MC的策略梯度算法，它是基于策略梯度的一种算法，策略梯度算法是指先找到一个评价指标，然后利用随机梯度上升的方法来更新参数使评价指标不断的上升。流程：收集整个episode的轨迹（Trajectory，包括状态，奖励，动作，概率对数值等），然后根据整个episode的奖励值计算discounted累计回报，与对应动作的概率的log相乘取负号记为loss，之后进行一轮更新。**类比监督学习，这里的label就是动作概率的负对数值。** 应用中需要一个策略网络返回可选动作分布。算法流程如下：



实验：

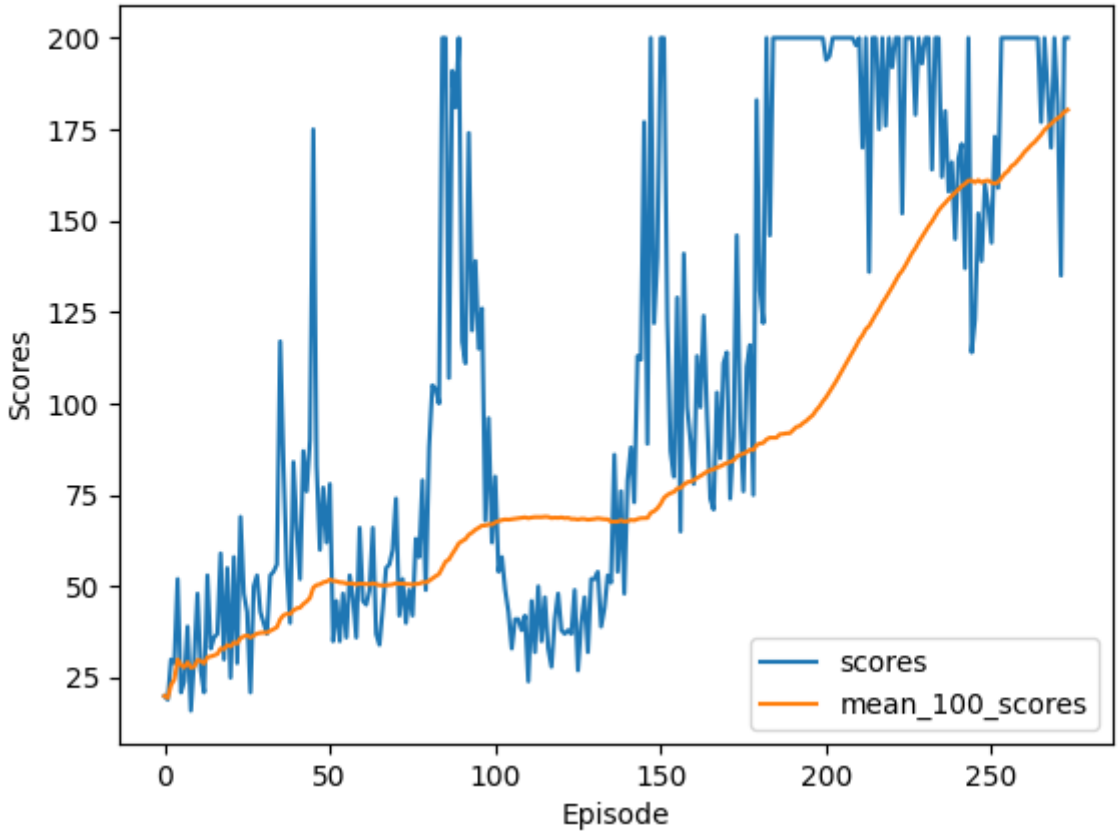
主要采用的参数

```
lr = 0.02; hidden_size = 16; reward = 180.0; //不过都在argument.py文件中设为默认值了
```

执行的命令：

```
python main.py --train_pg True  //参数默认
```

训练的可视化结果：



2. **(writing & coding)** 在`CartPole-v0`环境中实现REINFORCE算法的变种算法。虽然蒙特卡洛采样得到G_t是对reward的无偏估计，但环境的不确定性以及策略的随机性将会导致蒙特卡洛采样具有较大的方差。为了降低方差，在计算策略梯度的时候可以减去一个baseline，$b_\phi(s)$，常用的baseline形式就是状态值函数$V^{\pi_\theta}(s)$，具体公式：

$$J(\theta) = \frac{1}{\sum T_i}\sum_{i=1}^{|D|}\sum_{t=1}^{T_i}[log\pi_\theta(a_t^i|s_t^i)\hat{A}_t^i],$$

$$\hat{A}_t^i = G_t^i - b_\phi(s_t^i),$$

其中$\hat{A}_t^i$被称为优势函数（advantage function）。从理论层面来说，优势函数不会影响策略梯度，请给出理论证明。最终算法性能的评判标准：环境最终的reward至少收敛至180.0.

理论证明：

已知策略π是在状态s时动作空间A的取值分布（用a表示具体取值，A表示a的取值空间）。

首先，证明当baseline：$b_\phi(s)$独立于动作空间A时，将策略梯度的Q换为$b_\phi(s)$时，所求期望值为0。

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s) \cdot b_\phi(s)] \\
&= \mathbb{E}_{A\sim\pi}[\frac{\partial log\pi(A|s;\theta)}{\partial\theta} \cdot b_\phi(s)] \\
&\text{由于}b_\phi(s)\text{独立于}A\text{，因此}b_\phi(s)\text{可以提到期望外面：} \\
&= b_\phi(s) \cdot \mathbb{E}_{A\sim\pi}[\frac{\partial log\pi(A|s;\theta)}{\partial\theta}] \\
&\text{以离散动作空间为例，期望可以写为对动作}a\text{的求和形式：} \\
&= b_\phi(s) \cdot \sum_{a\in A}\pi(a|s;\theta) \cdot \frac{\partial log\pi(a|s;\theta)}{\partial\theta} \\
&\overset{\text{求导}}{=} b_\phi(s) \cdot \sum_{a\in A}\pi(a|s;\theta) \cdot \frac{1}{\pi(a|s;\theta)} \cdot \frac{\partial\pi(a|s;\theta)}{\partial\theta} \\
&= b_\phi(s) \cdot \sum_{a\in A}\frac{\partial\pi(a|s;\theta)}{\partial\theta} \\
&\text{由于是对}\theta\text{求导，因此求和符号能放到偏导内} \\
&= b_\phi(s) \cdot \frac{\partial\sum_{a\in A}\pi(a|s;\theta)}{\partial\theta} \\
&= b_\phi(s) \cdot \frac{\partial 1}{\partial\theta} \\
&= 0
\end{aligned}
$$

则引入baseline后的策略梯度为：

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s) \cdot (Q^{\pi_\theta(s,a)} - b_\phi(s))] \\
&= \mathbb{E}_{A\sim\pi}[\frac{\partial log\pi(A|s;\theta)}{\partial\theta} \cdot Q^{\pi_\theta(s,a)}] - \mathbb{E}_{A\sim\pi}[\frac{\partial log\pi(A|s;\theta)}{\partial\theta} \cdot b_\phi(s)] \\
&= \mathbb{E}_{A\sim\pi}[\frac{\partial log\pi(A|s;\theta)}{\partial\theta} \cdot Q^{\pi_\theta(s,a)}] - 0 \\
&= \mathbb{E}_{A\sim\pi}[\frac{\partial log\pi(A|s;\theta)}{\partial\theta} \cdot Q^{\pi_\theta(s,a)}] \\
&= \mathbb{E}_{\pi_\theta}[\nabla_\theta log\pi_\theta(a|s) Q^{\pi_\theta(s,a)}]
\end{aligned}
$$

结果表明当$b_\phi(s)$ 独立于A时，引入baseline后，使用优势函数作为$Q^{\pi_\theta(s,a)}$并不会影响策略梯度的值。证毕

通常baseline取值为状态值函数$V^{\pi_\theta}(s)$, 状态值函数为下一个状态值加折扣因子乘以下一个状态的预期奖励，然后取期望，因此只依赖于当前状态s_t, 而当前状态s_t与当前的动作空间无关，因此状态值函数不依赖于当前的动作A_t, 满足上述条件。

为什么通常选状态值函数呢？引入baseline的目的是为了降低方差，因此应该选择尽可能接近Q的b，而状态值函数V与动作值函数Q之间有如下关系：

$$
V_\pi(s) = \mathbb{E}_{A\sim\pi}[Q_\pi(s,A)]
$$

因此状态值函数与动作值函数很接近，能使方差尽可能小，算法收敛地更快。

算法说明：整体思路与REINFOECE相同，不过需要一个状态值网络生成状态值函数，在更新网络的时候，首先通过计算的discounted累计回报和状态值网络输出的状态值更新状态值网络，然后将discounted累计回报与状态值网络输出的状态值相减得到新的“discounted累计回报”，用此差值和对应动作的概率负对数来更新策略网络。部分代码如下：

```
state_vals = []
for state in states:
    state = torch.from_numpy(state).float().unsqueeze(0).to(self.device)
    state_vals.append(self.state_value_Net(state))  # 状态值网络输出状态值函数
state_vals = torch.stack(state_vals).squeeze()

self.train_state_value_net(discounted_rewards, state_vals)  # 用累计折扣回报和状态值函数更新状态值网络

deltas = [dt - val for dt, val in zip(discounted_rewards, state_vals)]
deltas = torch.tensor(deltas).to(self.device)

self.train_policy_net(deltas, log_probs)    # 用差值和动作的概率负对数更新策略网络
```

实验部分：

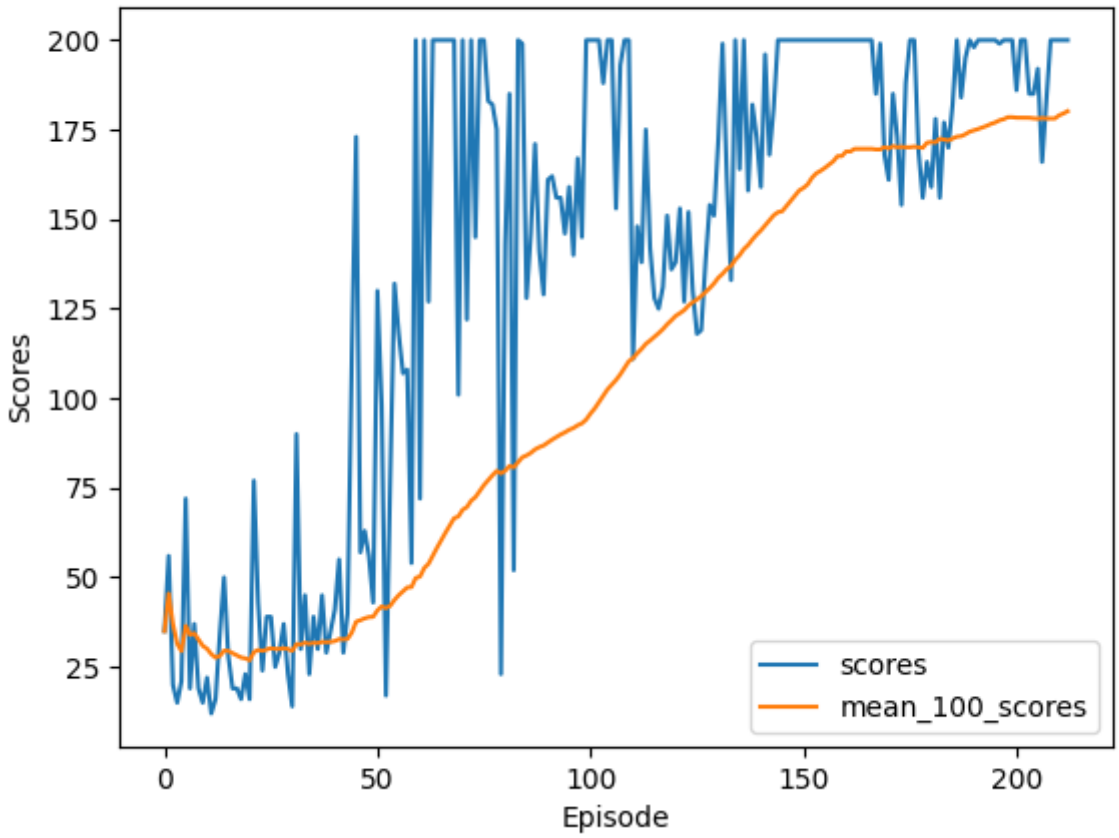
主要采用的参数

```
lr = 0.02; hidden_size = 16; reward = 180.0; //不过都在argument.py文件中设为默认值了 与无baseline一致
```

执行的命令：

```
python main.py --train_pg True --baseline   // 添加baseline参数
```

训练的可视化结果：

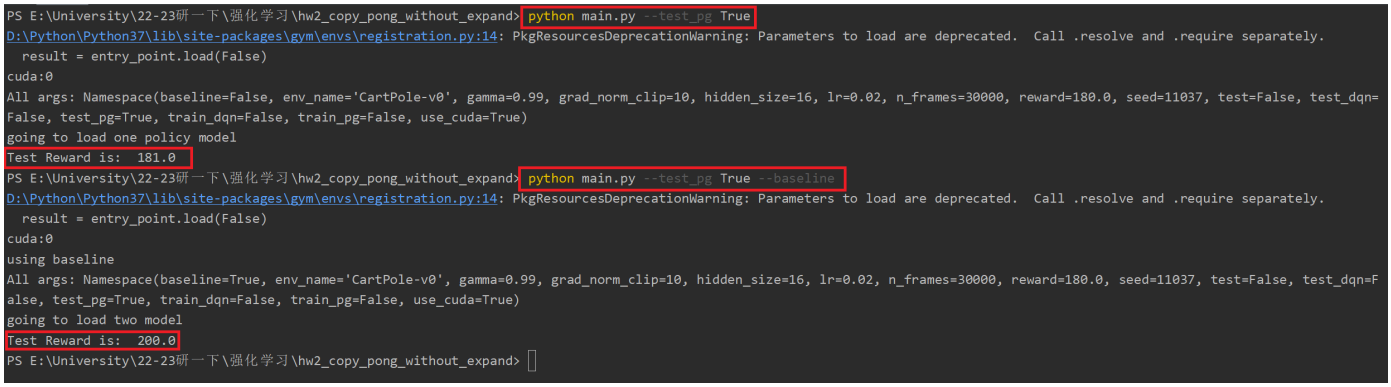


以上两种的测试结果为：

采用的命令：

```
python main.py --test_pg True
//和
python main.py --test_pg True --baseline
```

结果：



3. **(coding)** 在`CartPole-v0`环境中实现A2C算法.最终算法性能的评判标准：环境最终的reward至少收敛至180.0.

   下面实验实现了n步的A2C算法。需要两个网络：actor网络和critic网络，如下：

```
class Actor(nn.Module):
    def __init__(self, input_size=4, output_size=2):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(input_size, 64),
            nn.Tanh(),
            nn.Linear(64, 32),
            nn.Tanh(),
            nn.Linear(32, output_size),
            nn.Softmax()
        )

    def forward(self, x):
        return self.model(x)


class Critic(nn.Module):
    def __init__(self, input_size=4):
        super().__init__()
        self.model = nn.Sequential(
            nn.Linear(input_size, 64),
            nn.ReLU(),
            nn.Linear(64, 32),
            nn.ReLU(),
            nn.Linear(32, 1)
        )

    def forward(self, x):
        return self.model(x)
```

整体与前面的两个网络相同，不过有三层全连接层。

AgentA2C主要方法说明

```
# 初始化函数，初始化两个网络，两个Adam优化器，一些参数等
def __init__(self, env, args):

# make_action函数输入状态，返回动作以及分布
def make_action(self, observation, test=False):

# train_process函数为计算loss和反向传播的函数
def train_process(self, q_val):

# run函数为主函数，其中每个episode中，每经过n步就更新一次网络参数
def run(self):

# init_game_setting函数和test函数为测试时环境初始化、读取模型的函数和实际测试的函数
def init_game_setting(self):
def test(self):
```

算法说明:

Advantage Actor Critic使用优势值更新Actor和Critic网络，在使用优势的Q值中由两部分组成：$Q(s,a) = V(s) + A(s,a)$，则优势值为Q值减去value值。使用队列保存value值，每n步获取数据并更新一次。

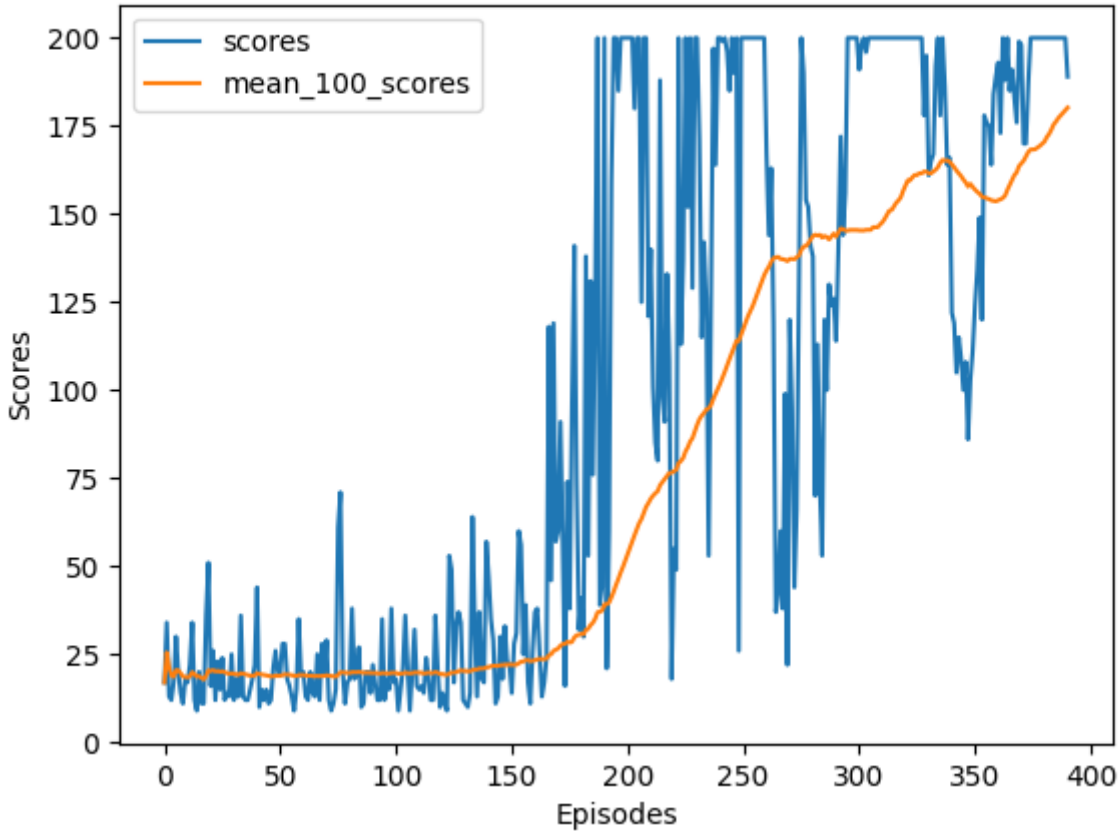
主要采用的参数:

```
lr = 0.001; n_step = 20; reward = 180;  //n_step控制n步a2c
```

运行命令:

```
python main.py --train_a2c True --lr 0.001
```

训练结果:



执行以下命令进行测试:

```
python main.py --test_a2c True
```

测试结果:

```
PS E:\University\22-23研一下\强化学习\hw2_copy_pong_without_expand> python main.py --test_a2c True
D:\Python\Python37\lib\site-packages\gym\envs\registration.py:14: PkgResourcesDeprecationWarning: Parameters to load are deprecated.  Call .resolve and .require separately.
  result = entry_point.load(False)
args: Namespace(baseline=False, env_name='CartPole-v0', gamma=0.99, grad_norm_clip=10, hidden_size=16, lr=0.02, n_frames=30000, n_step=20, reward=180.0, seed=11037, test=False, test_a2c=True, test_dqn=False, test_pg=False, train_a2c=False, train_dqn=False, train_pg=False, use_cuda=True)
D:\Python\Python37\lib\site-packages\torch\nn\modules\container.py:139: UserWarning: Implicit dimension choice for softmax has been deprecated. Change the call to include dim=X as an argument.
  input = module(input)
Test Reward is:  200.0
```

Task 3 Implementing DDPG（选做）

在`Box2D`中实现DDPG算法。由于DQN需要评估所有动作的Q值，故无法直接应用于连续动作的环境。DDPG可以在actor中计算连续策略的高斯分布，因此可以从高斯分布中采样得到智能体的动作。

1. **(coding)** 在`LunarLanderContinuous-v2`环境中实现DDPG算法，其中`Lunar Lander`是连续动作环境。最终算法性能的评判标准：环境最终的reward至少收敛至180.0。

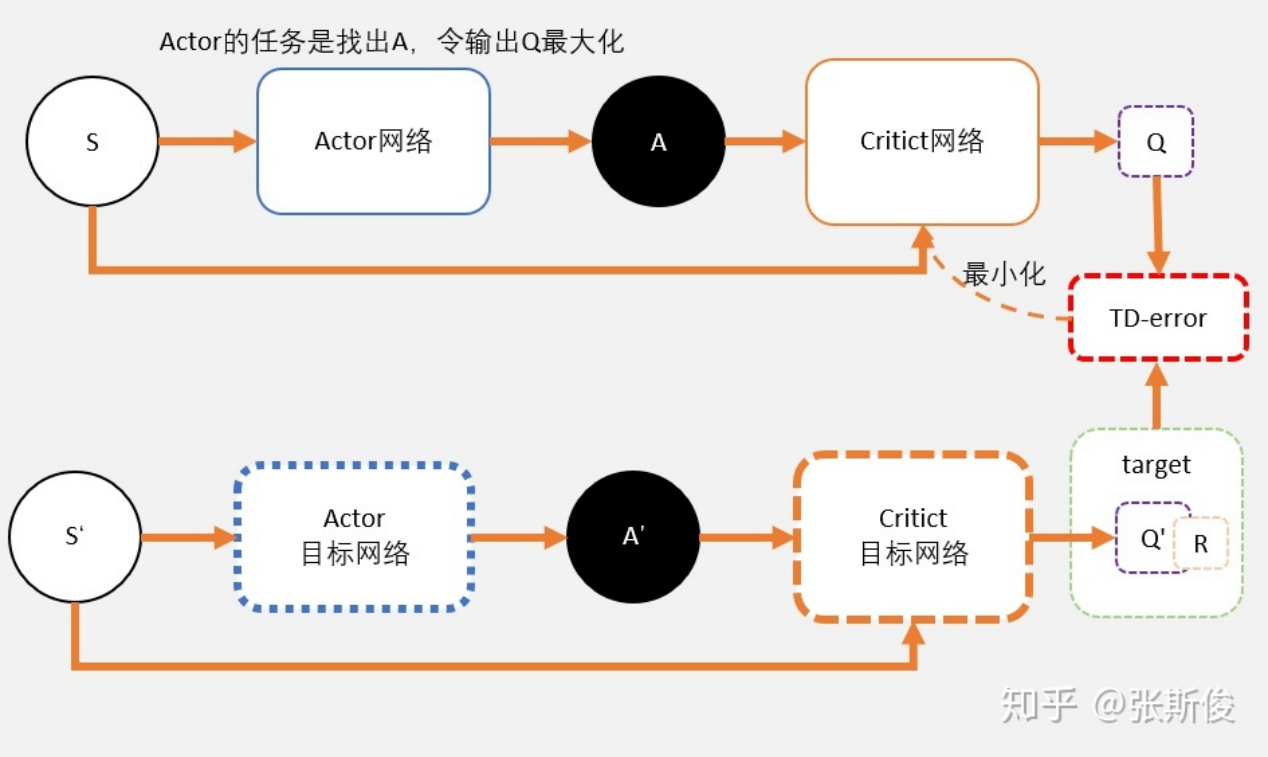
   Actor和Critic网络结构:

```
class Actor(nn.Module):
    def __init__(self, input_size, output_size, seed, hidden_size_1=128, hidden_size_2=128):
        super(Actor, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(input_size, hidden_size_1)
```

```
        self.fc2 = nn.Linear(hidden_size_1, hidden_size_2)
        self.fc3 = nn.Linear(hidden_size_2, output_size)
        self.reset_parameters()

    def reset_parameters(self):
        self.fc1.weight.data.uniform_(*hidden_init(self.fc1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, inputs):
        x = F.relu(self.fc1(inputs))
        x = F.relu(self.fc2(x))
        return torch.tanh(self.fc3(x))

class Critic(nn.Module):
    def __init__(self, input_size, output_size, seed, hidden_size_1=128, hidden_size_2=128):
        super(Critic, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fcs1 = nn.Linear(input_size, hidden_size_1)
        self.fc2 = nn.Linear(hidden_size_1 + output_size, hidden_size_2)
        self.fc3 = nn.Linear(hidden_size_2, 1)
        self.reset_parameters()

    def reset_parameters(self):
        self.fcs1.weight.data.uniform_(*hidden_init(self.fcs1))
        self.fc2.weight.data.uniform_(*hidden_init(self.fc2))
        self.fc3.weight.data.uniform_(-3e-3, 3e-3)

    def forward(self, state, action):
        # (state, action) pairs -> Q-values
        xs = F.relu(self.fcs1(state))
        x = torch.cat((xs, action), dim=1)
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

算法理解说明:

类似DQN，是off policy的，因此每个网络都需要两个实例，即local的和target的网络。Critic的更新与DQN相同，需要计算累计回报和目标网络输出。Actor网路的更新需要使用local actor得到预测值，local critic根据预测值计算loss，并取负均值作为最后的actor的 loss，用来更新Actor网络。整体流程如下：



实验部分:

主要参数设置:

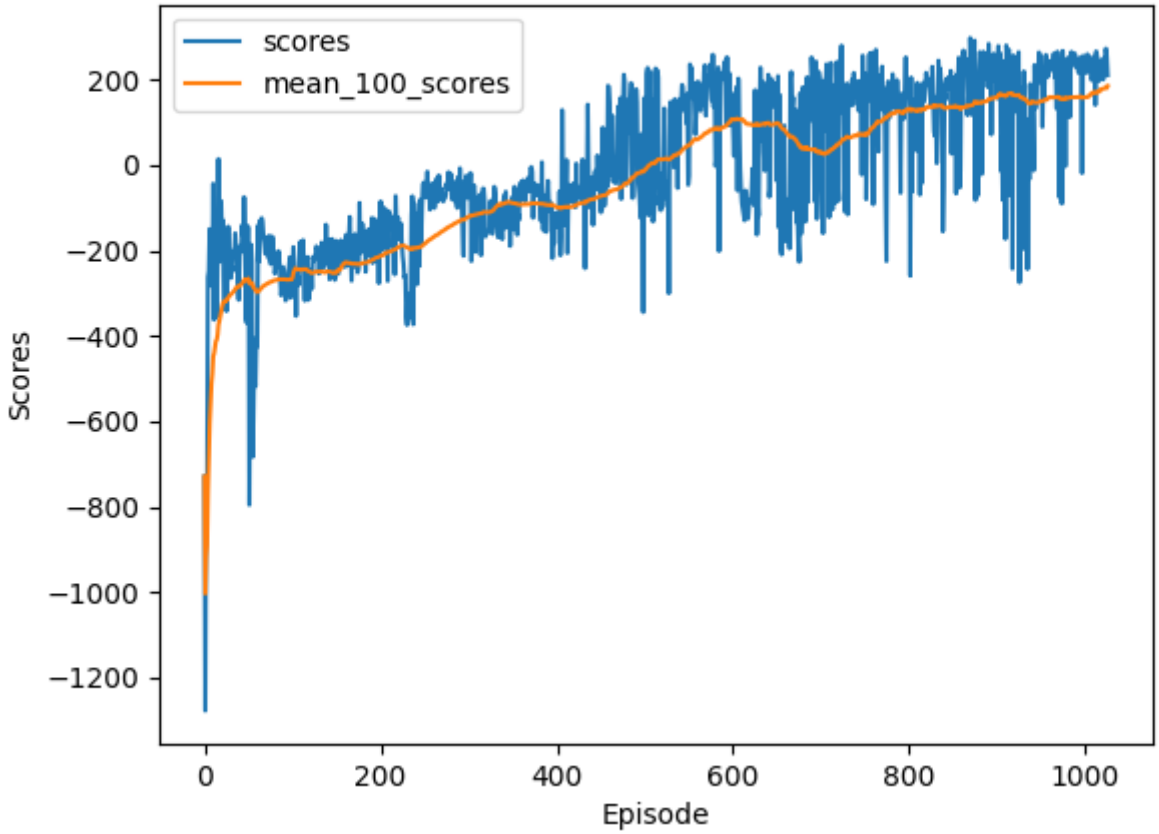
按照`argument.py`中的`ddpg_arguments`函数中的默认参数运行

```
buffer_size = 1000000; batch_size = 256; reward = 180.0;
```

执行命令:

```
python main.py --train_ddpg True
```

训练结果:



测试结果：

执行如下命令：

```
python main.py --test_ddpg True
```

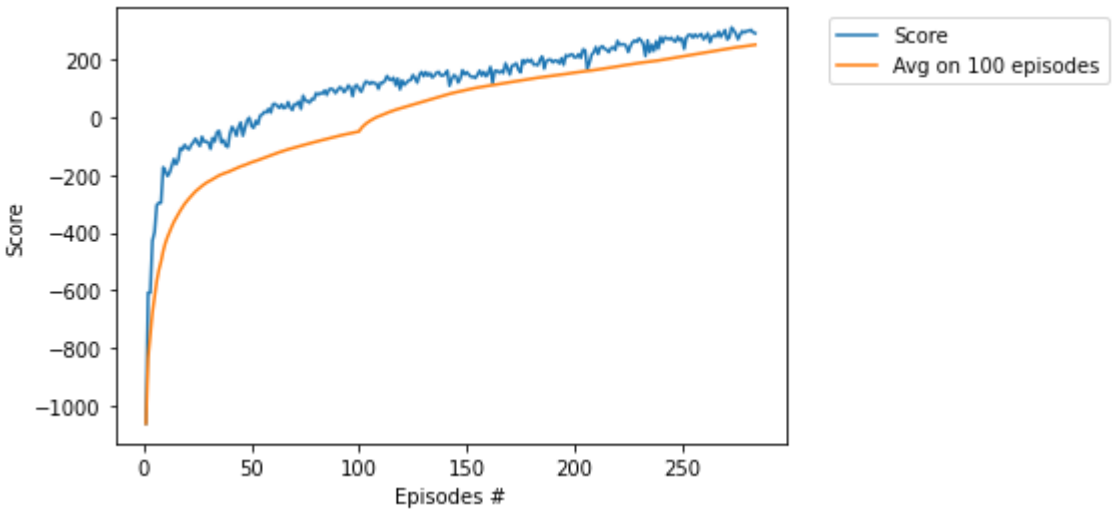
测试结果发现，即使最后平均奖励达到了180，但仍有很大的不确定性，从上图的训练结果中也能看出来，抖动很严重：

```
PS E:\University\22-23研一下\强化学习\作业\Homework_2\RL_21215068_蔡云龙_homework2\hw2_code> python main.py --test_ddpg True
args: Namespace(batch_size=256, buffer_size=1000000, env_name='LunarLanderContinuous-v2', gamma=0.99, reward=180.0, seed=8, tau=0.001, test_a2c=False, test_ddpg=True, test_dqn=Fal
se, test_pg=False, train_a2c=False, train_ddpg=False, train_dqn=False, train_pg=False)
Test Reward is -97.51118672570817
PS E:\University\22-23研一下\强化学习\作业\Homework_2\RL_21215068_蔡云龙_homework2\hw2_code> python main.py --test_ddpg True
D:\Python\Python37\lib\site-packages\gym\envs\registration.py:14: PkgResourcesDeprecationWarning: Parameters to load are deprecated.  Call .resolve and .require separately.
  result = entry_point.load(False)
args: Namespace(batch_size=256, buffer_size=1000000, env_name='LunarLanderContinuous-v2', gamma=0.99, reward=180.0, seed=8, tau=0.001, test_a2c=False, test_ddpg=True, test_dqn=Fal
se, test_pg=False, train_a2c=False, train_ddpg=False, train_dqn=False, train_pg=False)
Test Reward is 149.3918659907028
PS E:\University\22-23研一下\强化学习\作业\Homework_2\RL_21215068_蔡云龙_homework2\hw2_code> python main.py --test_ddpg True
D:\Python\Python37\lib\site-packages\gym\envs\registration.py:14: PkgResourcesDeprecationWarning: Parameters to load are deprecated.  Call .resolve and .require separately.
  result = entry_point.load(False)
args: Namespace(batch_size=256, buffer_size=1000000, env_name='LunarLanderContinuous-v2', gamma=0.99, reward=180.0, seed=8, tau=0.001, test_a2c=False, test_ddpg=True, test_dqn=Fal
se, test_pg=False, train_a2c=False, train_ddpg=False, train_dqn=False, train_pg=False)
Test Reward is 207.9048443273147
```

2. **(coding)** 在`Bipedal Walker`环境中实现DDPG、A2C、PPO 三个算法中的一个，其中`Bipedal Walker`是连续动作环境。最终算法性能的评判标准：环境最终的reward至少收敛至250.0。

   只跑了一下别人的代码（使用PPO算法）体验一下，参见`BipedalWalker-PPO_test`文件夹。效果如下：

   （训练结果和训练过程绘图，以及测试结果都在.ipynb文件中）



models目录说明:

此部分为训练的模型和输出内容的保存目录

- `dqn_pong`：为在PongNoFrameskip-v4环境中使用Dueling DQN网络和DQN的loss计算方式训练的结果，包括网络的数据文件，输出数据和训练曲线
- `double_dqn_pong`：同上，不过使用的是Double DQN方法
- `reinforce 和 reinforce_baseline`：为在CartPole-v0环境中分别实现REINFORCE和REINFORCE with baseline算法的训练结果，包括策略网络数据，训练曲线和输出数据；`reinforce_baseline`中额外有一个状态值网络的数据。
- `a2c`：在CartPole-v0环境中实现A2C算法的结果，包括两个网络（Actor和Critic）的数据，训练过程的输出和训练结果曲线
- `ddpg_LLC`：为在LunarLanderContinuous-v2环境中实现DDPG算法的训练结果，包括Actor和Critic网络的数据、输出数据和训练曲线。

主要文章和代码参考:

> DQN:
>
> 深度强化学习方法（DQN）玩转Atari游戏（pong）*libenfan*的博客-*CSDN*博客atari dqn
>
> Pong - Gym Documentation (gymlibrary.ml)
>
> collections中 deque的使用阿常呓语的博客-*CSDN*博客collections.deque
>
> alik-git/Pong-DQN: A Pong player using Deep Reinforcement Learning with Pytorch and OpenAI Gym based on a DQN model (github.com)
>
> DQN调整超参数体会万德*1010*的博客-*CSDN*博客dqn调参
>
> How to match DeepMind's Deep Q-Learning score in Breakout | by Fabio M. Graetz | Towards Data Science
>
> Double DQN:
>
> Double DQN - 简书 (jianshu.com)
>
> DeepRL系列(8): Double DQN(DDQN)原理与实现 - 知乎 (zhihu.com)
>
> Dueling DQN:
>
> [1511.06581v3] Dueling Network Architectures for Deep Reinforcement Learning (arxiv.org)

强化学习(十二) Dueling DQN - 刘建平Pinard - 博客园 (cnblogs.com)

Policy Gradient (REINFORCE and REINFORCE with baseline):

强化学习（三）--Reinforce算法*BUAA*小乔的博客-*CSDN*博客reinforce算法

第四章 策略梯度 (datawhalechina.github.io)

Policy Gradients: REINFORCE with Baseline | by Cheng Xi Tsou | Nerd For Tech | Medium

策略梯度中的Baseline (1/4) - YouTube

A2C:

强化学习（十三 ）--AC、A2C、A3C算法 - 知乎 (zhihu.com)

强化学习-A2C我的辉的博客-*CSDN*博客a2c pytorch

REINFORCE与A2C的异同 (策略梯度中的Baseline 4_4)哔哩哔哩bilibili

Advantage Actor Critic Tutorial: minA2C | by Mike Wang | Towards Data Science

Advantage Actor Critic (A2C) implementation | by Alvaro Durán Tovar | Deep Learning made easy | Medium

DDPG:

一文带你理清DDPG算法（附代码及代码解释） - 知乎 (zhihu.com)

Rafael1s/Deep-Reinforcement-Learning-Algorithms

PPO 实现Bipedal Walker的代码

Deep-Reinforcement-Learning-Algorithms/BipedalWalker-PPO-VectorizedEnv at master · Rafael1s/Deep-Reinforcement-Learning-Algorithms (github.com)