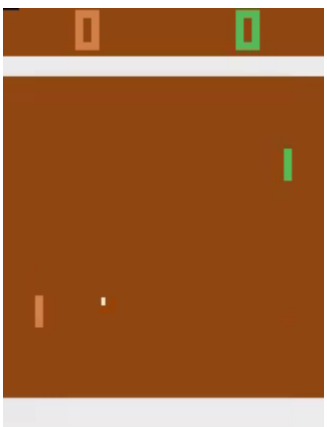
强化学习：第七次作业（二）

选一款Atari游戏，编写DQN算法进行训练

小组成员

冯帆 曹丁元 权玮虹 李浩然 孙健

Atari游戏：PongNoFrameskip-v4



橙色挡板（左）由电脑控制，绿色挡板（右）由玩家控制。游戏双方互相击打屏幕中的乒乓球，直至对方未能予以还击，首先获得20分的一方获胜。

实现过程

我们小组将项目工程托管在GitLab FengBear/rl-homework中。程序入口位于`./main.py`中，DQN算法及游戏流程控制的实现位于`./algorithm/pong`文件夹中，下图展示的是工程目录树：

```
├── algorithm
│   ├── __init__.py
```

```
│   ├── pong                                                    //程序主体
│   │   ├── __init__.py
│   │   ├── pong.py
│   │   └── wrapper.py
├── archieve                                                    //作业说明文档
│   ├── ...
│   ├── 强化学习：第七次作业.pdf
│   ├── 强化学习：第七次作业（二）.pdf
├── main.py                                                     //程序入口
├── output                                                      //DQN的网络参数
│   ├── ...
│   └── 07
│       ├── policy_params.pkl
│       └── target_params.pkl
├── video                                                       //游戏记录
│   └── 07
│       ├── openaigym.video.0.1514.video000000.meta.json
│       ├── openaigym.video.0.1514.video000000.mp4
│       ├── ...
└── ...
```

项目工程基于Pytorch-0.4 (cuda-9.0 version)实现，训练过程使用了单块GTX 1080 Ti 12GB显卡。

游戏介绍及封装

`PongNoFrameskip-v4`游戏的动作空间中共有6种动作，分别为：

```
['NOOP'(不动), 'FIRE'(不动), 'RIGHT'(上), 'LEFT'(下), 'RIGHTFIRE'(上), 'LEFTFIRE'(下)]
```

可以看到，每种动作均有两种，他们的概率分布是相同的（=1/3）。因此，在贪心策略进行随机选择的时候可以无视动作的概率分布进行随机选择。

为了加快训练的收敛，我们小组沿用参考文献[2,4]的操作，为游戏增添了一系列的`wrapper`，这些wraper的说明可以参见参考文献[4]的解释。

```
class NoopResetEnv(gym.Wrapper)     //reset时随机产生一定数量的NOOP动作
class FireResetEnv(gym.Wrapper)     //本工程没有使用该Wrapper，有些游戏需要以Fire作为起始状态
class EpisodicLifeEnv(gym.Wrapper)  //达到20分就终止本次Episode，以帮助收敛
class MaxAndSkipEnv(gym.Wrapper)    //重复相邻帧的动作选择以加快训练
class ProcessFrame84(gym.ObservationWrapper)    //游戏帧由RGB转为灰度并下采样至84长*84宽
class ClippedRewardsWrapper(gym.RewardWrapper)  //把游戏的奖赏约束在[-1, 1]
class LazyFrames(object)                        //observation
class FrameStack(gym.Wrapper)                   //将相邻的帧堆叠起来
class ScaledFloatFrame(gym.ObservationWrapper)  //本工程没有使用
class ImageToPyTorch(gym.ObservationWrapper)    //将帧数据转换成channel, w, h的格式
```

封装函数为`algorithm.pong.wrapper.wrapper(env)`，该函数中可以看到封装（帧数据处理）的顺序。

```
def wrapper(env):
    env = EpisodicLifeEnv(env)
    env = NoopResetEnv(env, noop_max=30)
    env = MaxAndSkipEnv(env, skip=4)
    env = ProcessFrame84(env)
    env = ImageToPyTorch(env)
    env = FrameStack(env, 4)
    env = ClippedRewardsWrapper(env)
    return env
```

游戏的引入位于`algorithm.pong.pong()`函数中。额外的，我们小组提供了在训练时保存游戏界面的功能。这样除了在训练时可以观察训练的效果，也能在训练结束后有数据保留。视频数据位于`video/07`文件夹中，网络的参数数据位于`output/07`文件夹中。工程中设定每进行`100`次迭代就保存一次视频数据。

```
def pong():
    # 乒乓游戏的环境
    env = gym.make('PongNoFrameskip-v4')
    env = wrapper(env) //封装
    # 保存游戏界面
    env = gym.wrappers.Monitor(env, './video/07/', force=True, video_callable=lambda
count: count % 100 == 0)
    ...
```

训练数据存储：**ReplayMemory**类

单条训练数据的定义如下。在这里，有一个缺陷：即在计算奖赏时，需要根据`done`的状态来决定是否需要累积奖赏的值，然而在训练数据中我们并未记录`done`的值。由于单次训练网络的时间很长，我们小组也就没有修改这个缺陷了。

```
# 定义游戏记录的形式
Transition = namedtuple('Transition',
                        ('state', 'action', 'next_state', 'reward'))
```

对于DQN训练所需要使用的数据，我们小组采用FIFO队列实现，具体实现请参见`pong.ReplayMemory`类。

```
class ReplayMemory:
    def __init__(self, maxlen):
        # FIFO队列
        self.memory = deque(maxlen=maxlen)

    def push(self, *args):  //将数据插入队列
        self.memory.append(Transition(*args))

    def is_sample(self, tick):  //判断是否可以进行采样
        return False if tick < REPLAY_INIT_SIZE else True

    def sample(self, batch_size):   //依照BATCH_SIZE进行采样
        idx = np.random.choice(len(self.memory)-batch_size)
        return list(self.memory)[idx:idx+batch_size]
```

在工程里，我们小组设定训练数据队列的最大存储长度为`REPLAY_MEMORY_SIZE = 100000`。在每次迭代前，都会执行队列清空`buffer.memory.clear()`操作，以使得当前的训练不受前轮训练数据的影响。在迭代时，我们设定：当训练数据队列的数据达到一定长度后`REPLAY_INIT_SIZE=1000`，才进行采样。这样能够防止在队列中的数据不足时采样失效的问题。采样的数据维度为[BATCH_SIZE, 4, 84, 84]。

DQN网络

DQN网络由`algorithm.pong.DQN`类实现。参照参考文献[2,3,5]中DQN网络的实现方式，工程中的DQN网络由3层卷积层和2层全连接层构成。除最后的全连接层之外，每层之间均使用ReLU非线性激活函数。

网络的输入维度为[BATCH_SIZE, 4, one_frame_height, one_frame_width]。其中，`4`表示输入的通道数，本工程的DQN网络采用4帧连续的经过灰度转换的游戏帧作为网络的输入。经过图像变换，单帧的尺寸为[84, 84]。

3层卷积层的卷积核分别为[8, 4, 3]，卷积步长`stride`分别为[4, 2, 1]，输出通道分别为[32, 64, 64]。2层全连接层的输出维度分别为[512, `num_actions`]，其中，`num_actions`是`PongNoFrameskip-v4`游戏的动作空间中动作的数目，其值为6。

贪心策略的动作选择

我们小组采用ϵ-贪心策略来进行动作选择，具体实现参见`pong.select_action(state, net, num_actions, epsilon)`函数：

```
def select_action(state, net, num_actions, epsilon):
    if np.random.uniform(0, 1) > epsilon:
        with torch.no_grad():
            return net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[np.random.choice(num_actions)]], device=DEVICE,
dtype=torch.long)
```

在由`np.random.uniform`产生随机值高于`EPSILON=0.1`的情况下，程序会选择训练网络的输出值的最大值对应的动作作为选择动作，否则将随机从6种动作中选择。

网络训练的方法定义

DQN网络的训练超参如下：

```
# 超参
EPISODE = 50000                 # 训练轮次
REPLAY_MEMORY_SIZE = 100000     # 保留游戏记录的条数
REPLAY_INIT_SIZE = 10000        # 保留一定条数的记录后再开始训练
TARGET_SYNC = 1000              # 将policy网络的参数同步到target网络的步长
BATCH_SIZE = 32                 # 批次大小
LEARNING_RATE = 0.0001          # 网络的学习效率
EPSILON = 0.1                   # 贪心策略的参数
GAMMA = 0.9                     # 计算奖赏的折扣因子
# 是否使用CUDA
DEVICE = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

依据DQN算法，我们小组使用两个DQN网络来完成训练：

- policy_net：用于产生训练数据的网络
- target_net：实际进行游戏时的动作选择网络

```
policy_net = DQN(*env.observation_space.shape, env.action_space.n).to(DEVICE)
target_net = DQN(*env.observation_space.shape, env.action_space.n).to(DEVICE)
```

依据DQN算法和参考文献[3,5]，我们小组实现了DQN的网络训练方法`pong.train_model(data, policy_net, target_net, optimizer, criterion)`。对于Q值的计算及损失的计算参见如下代码：

```
def train_model(data, policy_net, target_net, optimizer, criterion):
    # 从data中提取信息
    states, actions, next_states, rewards = Transition(*zip(*data))
    states = torch.cat(states).to(DEVICE)
    actions = torch.cat(actions).to(DEVICE)
    next_states = torch.cat(next_states).to(DEVICE)
    rewards = torch.cat(rewards).to(DEVICE)

    # 根据data中的当前的state，计算Q值-policy net
    current_q_values = policy_net(states).gather(1, actions)
    # 根据data中的下一步state，计算Q值-target net
    with torch.no_grad():
        next_state_q_values = target_net(next_states).max(1)[0].view(BATCH_SIZE,
1).data
    expected_values = rewards.to(DEVICE) + GAMMA * next_state_q_values.to(DEVICE)

    # 计算损失
    loss = criterion(current_q_values, expected_values)
    optimizer.zero_grad()
    # 反向传播
    loss.backward()
    # 为了加快收敛，强制将policy net中的参数置为[-1, 1]
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()
```

此外，训练网络使用的优化器为Adam优化器，损失函数为SmoothL1Loss函数。

```
    # 定义优化器（梯度下降的方法）
    optimizer = optim.Adam(policy_net.parameters(), lr=LEARNING_RATE)

    # 定义损失函数
    criterion = nn.SmoothL1Loss()
```

网络训练的流程

网络的训练附着在gym游戏的流程中。总的来说：

- 在每次进行`env.render()`之后，利用`env.step(action)`产生的游戏信息生成`Transition`类型的训练数据，并将其插入到记录游戏数据的`buffer`中。
- 利用从`buffer`中、按照`BATCH_SIZE`随机采样训练数据。

- 使用上述步骤采样的训练数据计算`policy_net`在`state、action`的Q值及`target_net`在`next_state、max_action`的Q值，计算损失并反向传播。需要注意target_net中的参数不参与反向传播，因此需要添加`with torch.no_grad()`的限定。
- 每迭代`TARGET_SYNC`步就使用`policy_net`的参数来更新`target_net`的参数（Q值）。

```
    for i in range(EPISODE):
        # 每一轮次获取初始state
        last_state = torch.from_numpy(np.array(env.reset())).to(DEVICE).unsqueeze(0)
        # 清空replay buffer
        buffer.memory.clear()
        # 保存训练参数
        torch.save(policy_net.state_dict(), './output/07/policy_params.pkl')
        torch.save(target_net.state_dict(), './output/07/target_params.pkl')

        for tick in count():
            # 贪心算法进行动作决策
            action = int(select_action(last_state, policy_net, env.action_space.n,
EPSILON))
            # 根据动作生成下一状态及奖赏
            env.render()
            observation, reward, done, info = env.step(action)
            next_state =
torch.from_numpy(np.array(observation)).to(DEVICE).unsqueeze(0)
            buffer.push(last_state, torch.tensor(action).view(1, 1), next_state,
torch.tensor(reward).view(1, 1))
            # 更新状态
            last_state = next_state
            # 训练policy net：保留一定条数的记录后再开始训练
            # 从buffer中采样
            if buffer.is_sample(tick):
                data = buffer.sample(BATCH_SIZE)
                train_model(data, policy_net, target_net, optimizer, criterion)
            # 同步target net：间隔一定条数的记录后再进行同步
            if tick % TARGET_SYNC == 0:
                target_net.load_state_dict(policy_net.state_dict())
            if done:
                break
    env.close()
```

训练结果

还在训练中，训练完了我会补上2333

参考文献及代码片段

1. PyTorch文档
2. openai/baselines: OpenAI Baselines is a set of high-quality implementations of reinforcement learning algorithms.
3. Shmuma/ptan: PTAN stands for PyTorch AgentNet -- reimplementation of AgentNet library for PyTorch
4. Speeding up DQN on PyTorch: how to solve Pong in 30 minutes
5. mrussek/pytorch-pong: *Pong* agent using *PyTorch* and OpenAI gym