AI导论——DQL 算法实践

授课老师：邓浩老师　　1854062 许之博

AI导论——DQL 算法实践
1 背景
1.1 强化学习简述
1.2 强化学习的基本要素
1.3 时序差分法
1.4 时序差分离线控制算法Q-Learning
1.4.1 Q-Learning 引入
1.4.2 Q-Learning 概述
1.4.3 Q-Learning流程
1.5 Deep Q-Learning
1.5.1 价值函数的近似表示方法
1.5.2 Deep Q-Learning 算法思路
1.6 项目工具
2 实现
2.1 神经网络搭建
2.2 训练过程
2.3 测试过程
3 效果展示

1 背景

1.1 强化学习简述

强化学习的学习思路和人比较类似，是在实践中学习。强化学习是和监督学习，非监督学习并列的第三种机器学习方法。强化学习来和监督学习最大的区别是它是没有监督学习已经准备好的训练数据输出值的。强化学习只有奖励值，但是这个奖励值和监督学习的输出值不一样，它不是事先给出的，而是延后给出的。同时，强化学习的每一步与时间顺序前后关系紧密。而监督学习的训练数据之间一般都是独立的，没有这种前后的依赖关系。

1.2 强化学习的基本要素

1. 环境的状态S, t时刻环境的状态 S_t 是它的环境状态集中某一个状态。
2. 个体的动作A, t时刻个体采取的动作 A_t 是它的动作集中某一个动作。
3. 环境的奖励R,t时刻个体在状态S_t采取的动作A_t对应的奖励R_{t+1}会在t+1时刻得到。
4. 个体的策略(policy)π,它代表个体采取动作的依据，即个体会依据策略π来选择动作。
5. 个体在策略π和状态s时，采取行动后的价值（value），一般用$v_\pi(s)$表示。
6. 奖励衰减因子γ，在[0，1]之间。
7. 环境的状态转化模型，即在状态s下采取动作a,转到下一个状态s'的概率，表示为$P^a_{ss'}$。
8. 探索率ϵ，在训练选择最优动作时，会有一定的概率ϵ不选择使当前轮迭代价值最大的动作。

1.3 时序差分法

在强化学习问题中，对于模型状态转化概率矩阵P始终是已知的强化学习问题，一般称为基于模型的强化学习问题。也有很多强化学习问题，没有办法事先得到模型状态转化概率矩阵P，这种类型的问题就是不基于模型的强化学习问题。 时序差分法是不基于模型的强化学习问题求解方法，可以用于解决强化学习中的控制问题和预测问题。时序差分法也是一种不需要完整状态序列就可以求解强化学习问题的方法。而另一种不基于模型的强化学习问题求解方法——蒙特卡罗方法需要获得完整的状态序列。

- 预测问题：即给定强化学习的5个要素：状态集S, 动作集A, 即时奖励R，衰减因子γ, 给定策略π， 求解该策略的状态价值函数$v(\pi)$。
- 控制问题：也就是求解最优的价值函数和策略。给定强化学习的5个要素：状态集S, 动作集A, 即时奖励R，衰减因子γ, 探索率ϵ, 求解最优的动作价值函数q和最优策略π。

1.4 时序差分离线控制算法Q-Learning

1.4.1 Q-Learning 引入

Q-Learning算法是一种使用时序差分求解强化学习控制问题的方法，回顾控制问题的定义可以表示为：给定强化学习的5个要素：状态集S, 动作集A, 即时奖励R，衰减因子γ, 探索率ϵ, 求解最优的动作价值函数q_*和最优策略π_*。

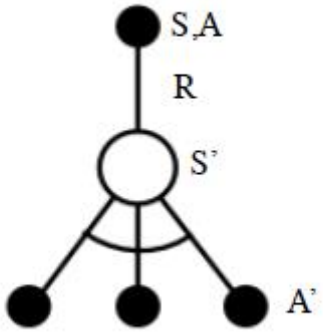
这一类强化学习的问题求解不需要环境的状态转化模型，是不基于模型的强化学习问题求解方法。对于它的控制问题求解，和蒙特卡罗法类似，都是价值迭代，即通过价值函数的更新，来更新策略，通过策略来产生新的状态和即时奖励，进而更新价值函数。一直进行下去，直到价值函数和策略都收敛。

再回顾下时序差分法的控制问题，可以分为两类，一类是在线控制，即一直使用一个策略来更新价值函数和选择新的动作, 而另一类是离线控制，会使用两个控制策略，一个策略用于选择新的动作，另一个策略用于更新价值函数。这一类的经典算法就是Q-Learning。

对于Q-Learning，使用$\epsilon-$贪婪法来选择新的动作。但是对于价值函数的更新，Q-Learning使用的是贪婪法，而不再是$\epsilon-$贪婪法。

1.4.2 Q-Learning 概述

Q-Learning算法的拓扑图：



首先基于状态S，用$\epsilon-$贪婪法选择到动作A, 然后执行动作A，得到奖励R，并进入状态S'，用$\epsilon-$贪婪法选择A',然后来更新价值函数。但是Q-Learning则不同。对于Q-Learning，它基于状态S'，没有使用$\epsilon-$贪婪法选择A'，而是使用贪婪法选择A'，也就是说，选择使$Q(S',a)$最大的a作为A'来更新价值函数。用数学公式表示就是：

$$Q(S,A)=Q(S,A)+\alpha(R+\gamma max_a Q(S',a)-Q(S,A))$$

对应到上图中就是在图下方的三个黑圆圈动作中选择一个使$Q(S',a)$最大的动作作为A'。

此时选择的动作只会参与价值函数的更新，不会真正的执行。价值函数更新后，新的执行动作需要基于状态*S'*，用*ϵ*−贪婪法重新选择得到。

1.4.3 Q-Learning流程

算法输入：迭代轮数*T*，状态集*S*, 动作集*A*, 步长*α*，衰减因子*γ*, 探索率*ϵ*,

输出：所有的状态和动作对应的价值*Q*

1.随机初始化所有的状态和动作对应的价值*Q* 。对于终止状态其*Q*值初始化为0。

2. for i from 1 to T，进行迭代。

a) 初始化*S*为当前状态序列的第一个状态。

b) 用*ϵ*−贪婪法在当前状态*S*选择出动作*A*

c) 在状态*S*执行当前动作*A*,得到新状态*S'*和奖励*R*

d) 更新价值函数*Q*(*S*,*A*)

e) *S*=*S'*

f) 如果*S'*是终止状态，当前轮迭代完毕，否则转到步骤b)

1.5 Deep Q-Learning

1.5.1 价值函数的近似表示方法

由于问题的状态集合规模大，一个可行的建模方法是价值函数的近似表示。可以引入一个状态$\hat{v}$, 这个函数由参数*ω*描述，并接受状态*s*作为输入，计算后得到状态*s*的价值，并即期望：

$$\hat{v}(s,\omega) \approx v_{\pi}(s)$$

类似的，引入一个动作价值函数$\hat{q}$,这个函数由参数*ω*描述，并接受状态s与动作a作为输入，计算后得到动作价值，即期望：

$$\hat{q}(s,a,\omega) \approx q_{\pi}(s,a)$$

价值函数近似的方法很多，比如最简单的线性表示法，用*ϕ*(*s*)表示状态s的特征向量，则此时的状态价值函数可以近似表示为：

$$\hat{v}(s,\omega) = \phi(s)^T\omega$$

当然，除了线性表示法，还可以用决策树，最近邻，傅里叶变换，神经网络来表达状态价值函数。而最常见，应用最广泛的表示方法是神经网络。对于神经网络，可以使用DNN，CNN或RNN。

1.5.2 Deep Q-Learning 算法思路

Deep Q-Learning算法的基本思路来源于Q-Learning。但是和Q-Learning不同的地方在于，它的Q值的计算不是直接通过状态值s和动作来计算，而是通过上面讲到的Q网络来计算的。这个Q网络是一个神经网络，我们一般简称Deep Q-Learning为DQN。DQN的输入是我们的状态s对应的状态向量*ϕ*(*s*)，输出是所有动作在该状态下的动作价值函数Q。Q网络可以是DNN，CNN或者RNN，没有具体的网络结构要求。

DQN主要使用的技巧是经验回放, 即将每次和环境交互得到的奖励与状态更新情况都保存起来，用于后面目标Q值的更新。通过经验回放得到的目标Q值和通过Q网络计算的Q值肯定是有误差的，那么我们可以通过梯度的反向传播来更新神经网络的参数*w*，当*w*收敛后，我们的就得到的近似的Q值计算方法，进而贪婪策略也就求出来了。

下面总结DQN的算法流程，基于NIPS 2013 DQN。

算法输入：迭代轮数T，状态特征维度n, 动作集A, 步长α，衰减因子γ, 探索率ϵ, Q网络结构, 批量梯度下降的样本数m。

输出：Q网络参数

1. 随机初始化Q网络的所有参数w，基于w初始化所有的状态和动作对应的价值Q。清空经验回放的集合D。
2. for i from 1 to T，进行迭代。

a) 初始化S为当前状态序列的第一个状态, 拿到其特征向量$\phi(S)$

b) 在Q网络中使用$\phi(S)$作为输入，得到Q网络的所有动作对应的Q值输出。用ϵ−贪婪法在当前Q值输出中选择对应的动作A

c) 在状态S执行当前动作A,得到新状态S'对应的特征向量$\phi(S')$和奖励R$,是否终止状态is_end

d) 将{$\phi(S)$,A,R,$\phi(S')$,is_end}这个五元组存入经验回放集合D

e) $S=S'$

f) 从经验回放集合D中采样m个样本{$\phi(S_j)$, A_j, R_j, $\phi(S'_j)$, is_end_j}, j=1,2.,,,m，*计算当前目标Q值* y_j：

$$y_j = \begin{cases} R_j & \text{is\_end }_j \text{ is true} \\ R_j + \gamma \max_{a'} Q\left(\phi\left(S'_j\right), A'_j, w\right) & \text{is\_end }_j \text{ is false} \end{cases}$$

g) 使用均方差损失函数，通过神经网络的梯度反向传播来更新Q网络的所有参数w

h) 如果S'是终止状态，当前轮迭代完毕，否则转到步骤b)

注意，上述第二步的f步和g步的Q值计算也都需要通过Q网络计算得到。另外，实际应用中，为了算法较好的收敛，探索率ϵ需要随着迭代的进行而变小。

1.6 项目工具

- PyCharm CE 2020.3.3
- python3.8，pytorch1.8.0, pygame2.0.1

2 实现

通过结合资料及个人学习，实现了使用DQL算法玩经典的flappy bird游戏。

2.1 神经网络搭建

搭建神经网络方面使用了现今常用的且个人较为熟悉的pytorch框架。总体架构包括三层卷积层，使用ReLu激活函数，两个线性层。

```
import torch.nn as nn

class DeepQNetwork(nn.Module):
    def __init__(self):
        super(DeepQNetwork, self).__init__()
        self.conv1 = nn.Sequential(nn.Conv2d(4, 32, kernel_size=8, stride=4),
nn.ReLU(inplace=True))
        self.conv2 = nn.Sequential(nn.Conv2d(32, 64, kernel_size=4, stride=2),
nn.ReLU(inplace=True))
        self.conv3 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=3, stride=1),
nn.ReLU(inplace=True))#三个卷积层 使用relu
        self.fc1 = nn.Sequential(nn.Linear(7 * 7 * 64, 512),
nn.ReLU(inplace=True))
```

```
        self.fc2 = nn.Linear(512, 2)
        self._create_weights()

    def _create_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):
                nn.init.uniform_(m.weight, -0.01, 0.01)
                nn.init.constant_(m.bias, 0)

    def forward(self, input):
        output = self.conv1(input)
        output = self.conv2(output)
        output = self.conv3(output)
        output = output.view(output.size(0), -1)
        output = self.fc1(output)
        output = self.fc2(output)

        return output
```

2.2 训练过程

```
while iter < opt.num_iters:#训练轮数
        prediction = model(state)[0]#从state状态生成Q值输出
        epsilon = opt.final_epsilon + ((opt.num_iters - iter) *
(opt.initial_epsilon - opt.final_epsilon) / opt.num_iters)
        u = random()
        random_action = u <= epsilon
        if random_action:#ε-贪婪法选择动作
            print("Perform a random action")
            action = randint(0, 1)
        else:
            action = torch.argmax(prediction)
        #获得下阶段状态
        next_image, reward, terminal = game_state.next_frame(action)
        next_image = pre_processing(next_image[:game_state.screen_width,
:int(game_state.base_y)], opt.image_size,opt.image_size)
        next_image = torch.from_numpy(next_image)
        if torch.cuda.is_available():
            next_image = next_image.cuda()
        next_state = torch.cat((state[0, 1:, :, :], next_image))[None, :, :, :]
        #插入经验回放列表
        replay_memory.append([state, action, reward, next_state, terminal])
        if len(replay_memory) > opt.replay_memory_size:
            del replay_memory[0]
        #从经验回放列表中随机采样
        batch = sample(replay_memory, min(len(replay_memory), opt.batch_size))
        state_batch, action_batch, reward_batch,next_state_batch,terminal_batch
= zip(*batch)
        state_batch = torch.cat(tuple(state for state in state_batch))
        action_batch = torch.from_numpy(
            np.array([[1, 0] if action == 0 else [0, 1] for action in
action_batch], dtype=np.float32))
        reward_batch = torch.from_numpy(np.array(reward_batch, dtype=np.float32)
[:, None])
        next_state_batch = torch.cat(tuple(state for state in next_state_batch))

        if torch.cuda.is_available():
```

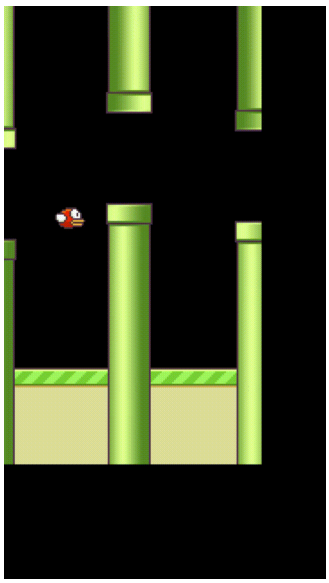
```
            state_batch = state_batch.cuda()
            action_batch = action_batch.cuda()
            reward_batch = reward_batch.cuda()
            next_state_batch = next_state_batch.cuda()
        current_prediction_batch = model(state_batch)
        next_prediction_batch = model(next_state_batch)
        #根据前文中的公式以及经验回放列表采样计算目标Q值y
        y_batch = torch.cat(
            tuple(reward if terminal else reward + opt.gamma *
torch.max(prediction) for reward, terminal, prediction in
                  zip(reward_batch, terminal_batch, next_prediction_batch)))
        #根据经验回放列表中的采样计算的当前Q值
        q_value = torch.sum(current_prediction_batch * action_batch, dim=1)

        optimizer.zero_grad()#设置梯度为0
        loss = criterion(q_value, y_batch)#求解损失函数 MSE
        loss.backward()#反向传播求损失函数梯度
        optimizer.step()#自适应梯度下降 更新模型参数

        state = next_state
        iter += 1
```

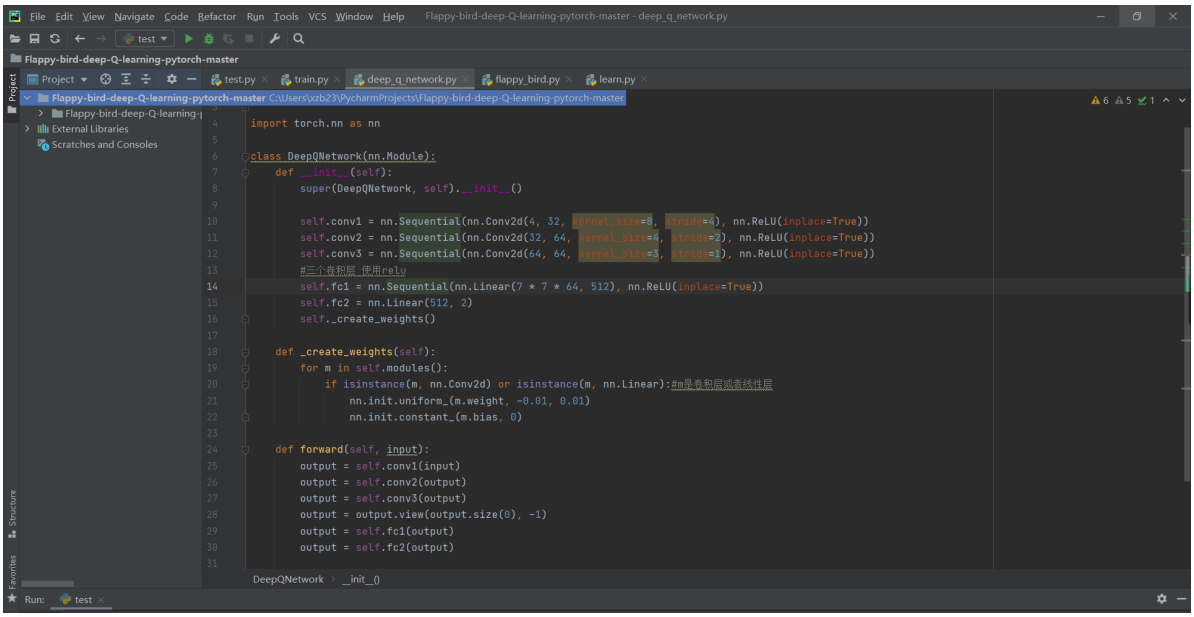
2.3 测试过程

使用已经训练好的模型，运行test.py文件，可以顺利的玩flappy bird游戏，其中flappy bird游戏使用的是pygame, 在此不赘述。

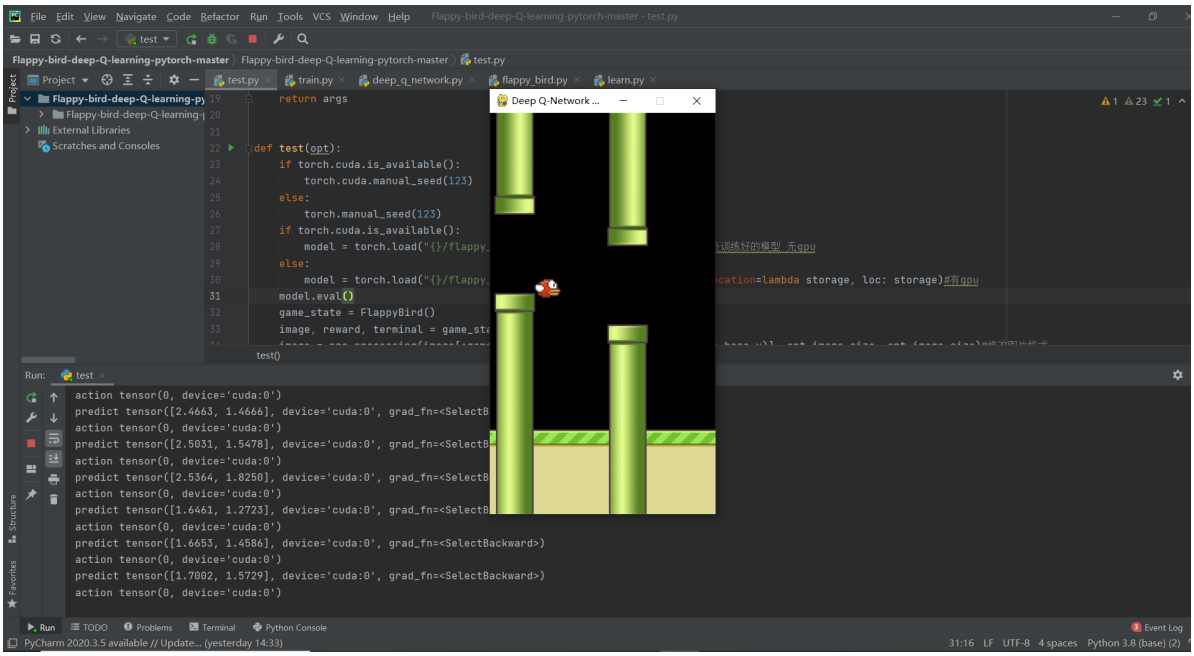
```
 #model为已经预训练好的模型
    while True:
        prediction = model(state)[0]#根据状态输出预测
        print('predict',prediction)
        action = torch.argmax(prediction)
        print('action',action)
        next_image, reward, terminal = game_state.next_frame(action)
        #下一个游戏帧
        next_image = pre_processing(next_image[:game_state.screen_width,
:int(game_state.base_y)], opt.image_size,opt.image_size)
        next_image = torch.from_numpy(next_image)
        if torch.cuda.is_available():
            next_image = next_image.cuda()
        next_state = torch.cat((state[0, 1:, :, :], next_image))[None, :, :, :]#
下一个游戏状态
        state = next_state
```

3 效果展示





```python
import torch.nn as nn

class DeepQNetwork(nn.Module):
    def __init__(self):
        super(DeepQNetwork, self).__init__()

        self.conv1 = nn.Sequential(nn.Conv2d(4, 32, kernel_size=8, stride=4), nn.ReLU(inplace=True))
        self.conv2 = nn.Sequential(nn.Conv2d(32, 64, kernel_size=4, stride=2), nn.ReLU(inplace=True))
        self.conv3 = nn.Sequential(nn.Conv2d(64, 64, kernel_size=3, stride=1), nn.ReLU(inplace=True))
        #三个卷积层 使用relu
        self.fc1 = nn.Sequential(nn.Linear(7 * 7 * 64, 512), nn.ReLU(inplace=True))
        self.fc2 = nn.Linear(512, 2)
        self._create_weights()

    def _create_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or isinstance(m, nn.Linear):##m是卷积层或者线性层
                nn.init.uniform_(m.weight, -0.01, 0.01)
                nn.init.constant_(m.bias, 0)

    def forward(self, input):
        output = self.conv1(input)
        output = self.conv2(output)
        output = self.conv3(output)
        output = output.view(output.size(0), -1)
        output = self.fc1(output)
        output = self.fc2(output)
```



```
action tensor(0, device='cuda:0')
predict tensor([2.4663, 1.4666], device='cuda:0', grad_fn=<SelectB
action tensor(0, device='cuda:0')
predict tensor([2.5031, 1.5478], device='cuda:0', grad_fn=<SelectB
action tensor(0, device='cuda:0')
predict tensor([2.5364, 1.8250], device='cuda:0', grad_fn=<SelectB
action tensor(0, device='cuda:0')
predict tensor([1.6461, 1.2723], device='cuda:0', grad_fn=<SelectB
action tensor(0, device='cuda:0')
predict tensor([1.6653, 1.4586], device='cuda:0', grad_fn=<SelectBackward>)
action tensor(0, device='cuda:0')
predict tensor([1.7002, 1.5729], device='cuda:0', grad_fn=<SelectBackward>)
action tensor(0, device='cuda:0')
```