中山大学计算机学院 人工智能 本科生实验报告

课程名称：Artificial Intelligence

学号	姓名
22336327	庄云皓

一、 实验题目

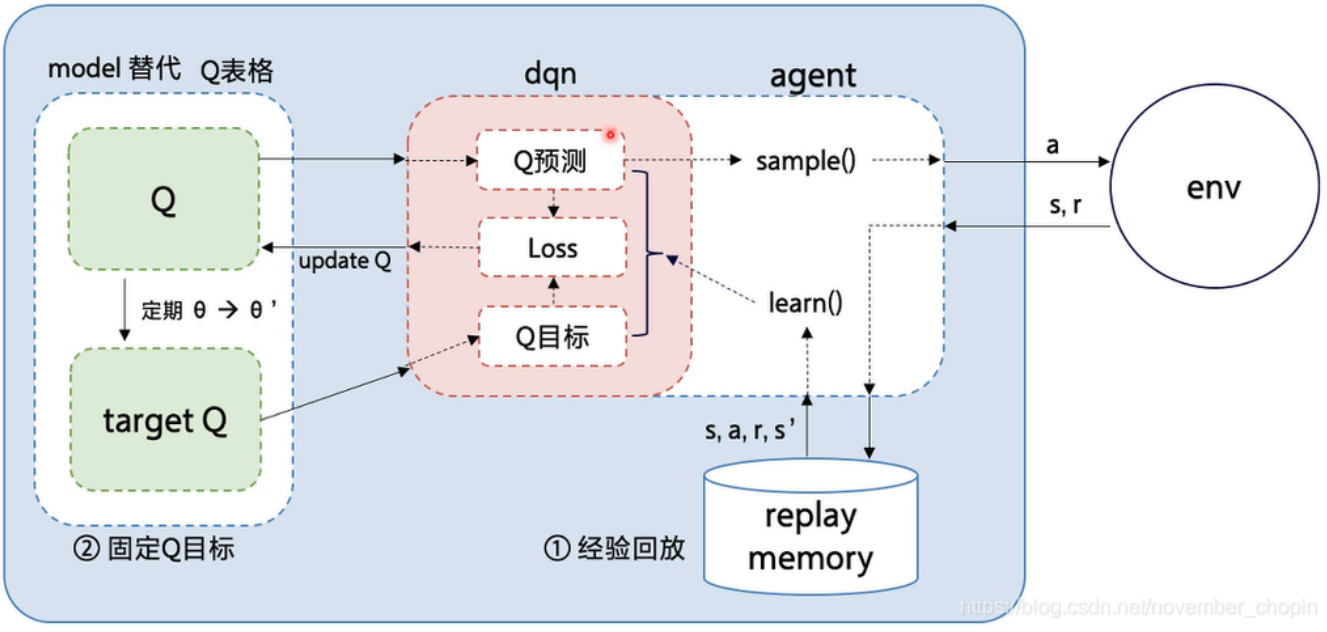
实验要求

在 `CartPole-v0`环境中实现DQN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.

二、 实验内容

1. 算法原理

   DQN



   智能体从环境中获得 $< s_t, a_t, r_t, s_{t+1} >$存入缓存区replay buffer，然后从当缓存区满且更新了一定量的数据之后，缓存区中随机抽取一定量数据，输入到神经网络中，得到Q值，使用 Q-Target 网络计算出 Q目标值，然后计算两者之间的损失函数。利用梯度下降来更新当前 网络参数，重复若干次后，把当前网络的参数复制给 Q-Target 网络。

   DQN算法采用的策略有：

   - 经验回放：具体来说就是用一块buffer ，用来存储每次探索获得数据$< s_t, a_t, r_t, s_{t+1} >$，每次训练从buffer里抽取一个batch的数据，然后进行训练，更新当前Q网络的参数
   - 固定目标网络：目标网络Q-Target是每隔一段时间，将当前网络的参数复制给Q-Target网络，这样在训练过程中，Q-Target网络的参数是固定的，不会随着训练的进行而变化，这样就可以避免训练过程中的不稳定。

2. 关键代码展示

分模块来介绍

ReplayBuffer是环境数据缓存

```
class ReplayBuffer:
 # 环境数据缓存
 def __init__(self, buffer_size: int, n_states: int):
     self.buffer_size = buffer_size # 最大的缓存空间
     self.n_states = n_states # 这里为4
     self.memory = np.array([[0.] *(n_states*2+3) for _ in range(self.buffer_size)])#一定要指定
type=float
     """
     memory: np.ndarray
         shape: (buffer_size, n_states*2+3)
         memory[i] = [s, a, r, t, s_]
     """
     self.buffer_counter = 0 #记录更新次数
     self.index = 0

 def __len__(self): #注意这里的长度是与环境交互后获得的数据数量
     return self.buffer_counter

 def push(self, *transition):
     """新增环境数据
     """

     # transition[state, action, reward, terminated, next_state]
     s = transition[0] # state: dim[4]
     a = transition[1] # action: int
     r = transition[2] # reward: float
     t = transition[3] # terminated: int
     s_ = transition[4] # next_state: dim[4]
     self.memory[self.index] = np.hstack((s, a, r, t, s_))
     self.index = (self.index + 1) % self.buffer_size

     self.buffer_counter += 1
     if self.buffer_counter % 100 == 0:
         logging.info(f"buffer_counter: {self.buffer_counter}, push_index: {self.index}")

 def sample(self, batch_size):
     """采样训练数据

     Args:
         batch_size (int): batch size

     Returns:
         np.ndarray: batch训练数据
     """

     sample_index = np.random.choice(self.buffer_size, batch_size)
     batch_buffer = self.memory[sample_index.T,:] #二维np.array切片获取某些行
     return batch_buffer

 def clean(self):
     self.memory = np.array([[0.] *(self.n_states*2+3) for _ in range(self.buffer_size)])#一定要指定
type=float
     self.buffer_counter = 0
     self.index = 0
```

AgentDQN类从环境获取数据，进行训练，做出决策，其中几个重要的函数如下：

初始化

```
class AgentDQN(Agent):
 def __init__(self, env, args):
     """
     Initialize every things you need here.
     For example: building your model
     """
     super(AgentDQN, self).__init__(env)

     self.env = env
     self.args = args
     self.epsilon = self.args.epsilon
     self.batch_size = self.args.batch_size

     self.init_game_setting()

 def init_game_setting(self):
     """初始化配置
     """
     self.n_states = self.env.observation_space.shape[0] # 4
     self.n_actions = self.env.action_space.n # 2

     self.learn_step_counter = 0
     self.buffer = ReplayBuffer(buffer_size=1000, n_states=self.n_states)
     self.q_net = QNetwork(input_size=self.n_states,
                           hidden_size=self.args.hidden_size,
                           output_size=self.n_actions)
     self.q_net.to(self.args.device)
     self.target_net = QNetwork(input_size=self.n_states,
                                hidden_size=self.args.hidden_size,
                                output_size=self.n_actions)
     self.target_net.to(self.args.device)
     self.optimizer = optim.Adam(self.q_net.parameters(), lr=self.args.lr)
     self.loss_func = nn.MSELoss()
```

训练模型 采用的是简单的三层感知机，两个隐藏层size为128 先把一个bacth_size里的是< s, a, r, s_{t+1} >取出来，作为神经网络的输入，计算损失，然后更新权重。

将s,a,r,s,t转换为tensor,可以看到它们的维度

```
torch.Size([100, 4])
torch.Size([100, 1])
torch.Size([100, 1])
torch.Size([100, 1])
torch.Size([100, 4])
```

按照$Q_t = R_t + \gamma max_a Q_{t+1}(s_{t+1}, a)$更新q_target网络的参数，其中$\gamma$是折扣因子，$Q_t$是当前状态$s_t$采取动作a的Q值，$R_t$是奖励，$s_{t+1}$是下一步状态，$Q_{t+1}(s_{t+1}, a)$是下一步状态采取动作a的Q值。

```
def train(self):
     """网络训练
     """
     self.q_net.train()
     self.target_net.eval()

     # 获取训练数据，以及预处理
     b_memory = self.buffer.sample(batch_size=self.batch_size)
     #print(np.array(b_memory).shape)
     b_s = torch.FloatTensor(b_memory[:, :self.n_states]).to(self.args.device) # 当前状态
     b_a = torch.LongTensor(b_memory[:, self.n_states:self.n_states+1].astype(int))
.to(self.args.device)# 动作
```

```
    b_r = torch.FloatTensor(b_memory[:, self.n_states+1:self.n_states+2]).to(self.args.device) # 收益
    b_t = torch.LongTensor(b_memory[:, self.n_states+2:self.n_states+3]).to(self.args.device) # 结束标志
    b_s_ = torch.FloatTensor(b_memory[:, -self.n_states:]).to(self.args.device) # 下一步状态

    # 网络计算
    q = self.q_net(b_s).gather(1, b_a) # [batch, 1], 获取当前状态下对应b_a动作位置的q_value
    q_next = self.target_net(b_s_).detach().max(1)[0] # [batch]
    q_next = q_next.view(self.batch_size, 1) # [batch, 1]
    # q_target = b_r + self.args.gamma * q_next * (1 - b_t) # [batch, 1], truncated=1的话,target=reward;否则target=reward+gamma*next
    q_target = b_r + self.args.gamma * q_next # [batch, 1]

    # 损失计算
    loss = self.loss_func(q, q_target)

    # 梯度回传
    self.optimizer.zero_grad()
    loss.backward()
    self.optimizer.step()
```

主体部分，智能体与环境交互，每新获得10条数据并且buffer是满的，就从buffer采样一个batch_size的数据训练Q网络，训练20次Q网络后把参数赋给Q_target网络。整个训练过程结束后保存模型，记录训练结果。

```
def run(self):
        """Implement the interaction between agent and environment here
        """

        sample_step_counter = 0 # 用来计算采样步数
        train_step_counter = 0 # 用来计算训练步数

        train_reward_list = []
        eval_reward_list = []

        for i_epoch in range(self.args.epoch):

            s, _ = self.env.reset() # dim [4]
            total_reward = 0

            while True:
                sample_step_counter += 1

                self.env.render()
                a = self.make_action(s) # 0 / 1

                # 采取动作
                # terminated: Pole Angle is greater than ±12°, Cart Position is greater than ±2.4 (center of the cart reaches the edge of the display)
                # truncated: Episode length is greater than 500 (200 for v0)
                s_, r, terminated, truncated, _ = self.env.step(a)

                # 重新定义reward, reward越大越好
                x, x_dot, theta, theta_dot = s_ # x: 小车绝对位置, theta: 木棍的倾斜角度
                r1_ = (self.env.x_threshold - abs(x)) / self.env.x_threshold
                r2_ = (self.env.theta_threshold_radians - abs(theta)) / self.env.theta_threshold_radians
                r_ = r1_ + r2_

                total_reward += r # 累加收益

                self.buffer.push(s, a, r_, int(terminated), s_) # 记录样本

                if sample_step_counter > 10 and len(self.buffer) > self.buffer.buffer_size:
                    sample_step_counter = 0 # 新增10条数据, 采样训练一次
```

```
                    self.epsilon = min(self.epsilon * 1.01 , 1) # 更新epsilon
                    self.train()
                    train_step_counter += 1
                    if train_step_counter % 20 == 0:
                        # 每训练20轮，同步一次参数到target_net
                        self.target_net.load_state_dict(self.q_net.state_dict())
                        logging.info(f">>>>>>>> update target_net.")

                if terminated or truncated:
                    break

                s = s_

            if train_step_counter > 0:
                # 每次epoch结束输出一次测试结果
                eval_reward = self.eval()

                eval_reward_list.append(eval_reward)
                train_reward_list.append(total_reward)
                logging.info(f"epoch: {i_epoch}, eval: {eval_reward}, train: {total_reward},
epsilon: {self.epsilon}")

        # 保存模型

        saved_model_path = os.path.join(self.args.saved_dir, "model.pt")
        os.makedirs(os.path.dirname("./"+saved_model_path), exist_ok=True)
        print(f"save model to {saved_model_path}")
        torch.save(self.q_net.state_dict(), saved_model_path)

        # 保存训练过程
        saved_reward_path = os.path.join(self.args.saved_dir, "reward.json")
        reward_trace = {
            "train": train_reward_list,
            "eval": eval_reward_list
        }
        with codecs.open(saved_reward_path, "w", "utf-8") as f:
            json.dump(reward_trace, f, indent=4)
```

智能体根据Q网络的结果行动.根据test参数和探索率（epsilon）的设定，选择执行贪婪策略（greedy policy）或随机策略（random policy）。

如果test为True或者test为False且生成的随机数小于探索率（self.epsilon），则使用贪婪策略。在贪婪策略中，首先将神经网络模型（self.q_net）切换到评估模式（eval()）。然后，通过向前传递输入张量x，获取每个动作的Q值。接下来，使用torch.max方法找到Q值中的最大值，并返回对应的动作索引。最后，将动作索引转换为NumPy数组格式，并将其存储在变量action中。

如果不满足上述条件，则执行随机策略。随机策略通过使用np.random.randint方法在动作空间中随机选择一个动作。

```
    def make_action(self, observation, test=False):
        """预测下一步动作
        """
        # observation 是一个状态: dim[4]
        x = torch.unsqueeze(torch.FloatTensor(observation), 0)

        if test or (not test and np.random.uniform() < self.epsilon):
            # greedy policy
            self.q_net.eval()
            with torch.no_grad():
                actions_value = self.q_net(x)
            action = torch.max(actions_value, 1)[1].data.numpy()  # 获取最大Q值的动作索引
            action = action[0]
        else:
            # random
```

```
            action = np.random.randint(0, self.n_actions)
        return action
```

3. 创新点&优化

3.1 reward的定义：（阈值-当前值）/阈值 小车越接近原点或者杆越竖直则奖励越大

$$r_1 = \frac{\text{x\_threshold} - |x|}{\text{x\_threshold}}$$

$$r_2 = \frac{\text{theta\_threshold\_radians} - |\theta|}{\text{theta\_threshold\_radians}}$$

$$\text{reward} = r_1 + r_2$$

```
                    r1_ = (self.env.x_threshold - abs(x)) / self.env.x_threshold
                    r2_ = (self.env.theta_threshold_radians - abs(theta)) /
self.env.theta_threshold_radians
                    r_ = r1_ + r2_
```

3.2 设置ε-greedy策略，智能体一开始以更高该概率选择随机动作，后期更高概率选择最优动作，训练效果更好

np.random.uniform() < epsilon时选择最优动作,epsilon逐渐增加

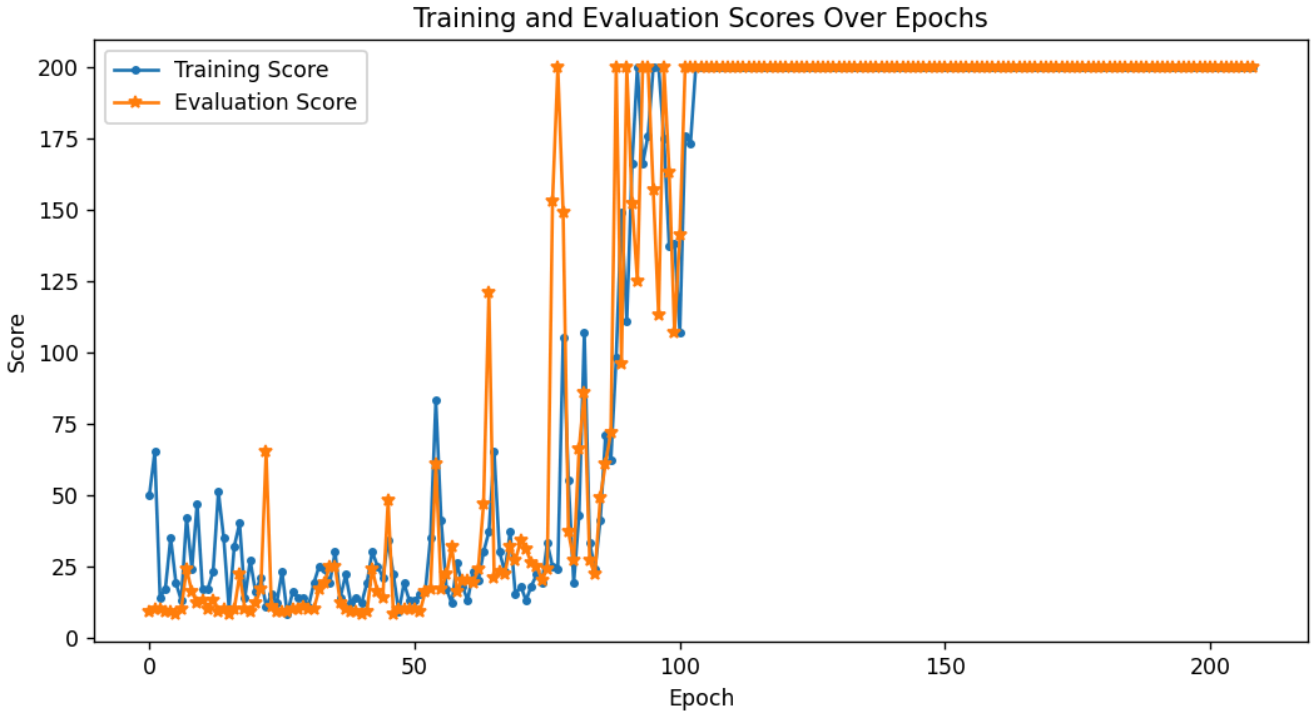
```
                if sample_step_counter > 10 and len(self.buffer) > self.buffer.buffer_size:
                    ...
                    self.epsilon = min(self.epsilon * 1.01 , 1) # 更新epsilon
```

三、 实验结果及分析

1. 实验结果展示示例

在命令行输入 `python main.py --train_dqn`开始训练，如果需要改一些超参数在后面加上即可，如 `python main.py --train_dqn --lr 0.005 --hidden_size 128 --batch_size 100`

在train_pg=False, train_dqn=True, device=None, env_name='CartPole-v0', hidden_size=128, lr=0.01, gamma=0.9, epsilon=0.1, batch_size=100, epoch=250时的结果：



2. 评测指标展示及分析

分析了一些超参数对训练时间，收敛时的轮数的影响

我们认为reward_list最后10个数相同并且>=180则认为达到收敛，取此时倒数第十个数的index为epochsToConvergence

lr	initial epison	hidden_size	batch_size	training_time_250epochs(s)	epochsToConvergence	image
0.05	0.1	64	100	14.002144575119019	128(不稳定)	
0.01	0.1	64	100	15.32469129562378	128	
0.005	0.1	64	100	7.702778577804565	207	
0.001	0.1	64	100	1.1269330978393555	250轮内不收敛	
0.01	0.05	64	100	6.769510984420776	215	
0.01	0.3	64	100	11.671736001968384	208	
0.01	0.1	128	100	15.382553815841675	145	
0.01	0.1	128	200	18.231366395950317	128	
0.01	0.1	256	200	22.39528775215149	127	
0.01	0.1	64	50	16.257569789886475	126	
0.01	0.1	64	400	14.638672590255737	152	

分析：随着lr增大，收敛速度加快，但是有可能收敛到局部最优解，训练时间增加，initial epison增大和lr增大情况类似。hidden_size增大训练时间增加，收敛时的epoch几乎不变。batch_size增加会使得要达到收敛的

epoch数增加。

对几个超参数是如何影响结果的思考：

1. Learning rate:
    - 学习率决定了权重更新的速度。
    - 较高的学习率可以加快收敛速度,但可能导致训练不稳定或无法收敛。
    - 较低的学习率则会减缓收敛过程,但可能使训练更稳定。
    - 合适的学习率需要通过实验来确定,通常在0.001到0.0001之间。
2. Initial epsilon:
    - 初始epsilon值决定了智能体最初的探索程度。
    - 较高的初始epsilon值会促进更多的exploitation,有利于在训练初期发现更好的策略。
    - 较低的初始epsilon则会更早地向exploration,可能会错过一些潜在的更好策略。
3. Hidden size:
    - 隐藏层的大小决定了模型的复杂度和表达能力。
    - 较大的隐藏层可以让模型学习到更复杂的特征和策略,但可能需要更长的训练时间。
    - 较小的隐藏层则会降低模型的学习能力,但可能会加快收敛速度。
    - 隐藏层大小的选择需要权衡模型复杂度和训练时间。
4. Batch size:
    - Batch size决定了每次参数更新时使用的样本数量。
    - 较大的batch size可以提高训练的稳定性和效率,但可能需要更多的内存。
    - 较小的batch size则可能导致训练更不稳定,但可以更快地进行参数更新。
    - 合适的batch size需要通过实验确定,通常在32到256之间。

四、 参考资料

Numpy 中 np.vstack() 和 np.hstack() 简单解析-CSDN博客

强化学习 7—— 一文读懂 Deep Q-Learning（DQN）算法_deep q learning-CSDN博客