

强化学习大作业

2020212185张扬

强化学习大作业

MountainCar离散版本

1. 环境
2. 算法
 - 2.1 DQN
 - 2.2 Double DQN
 - 2.3 网络结构
3. 代码
 - 3.1 代码结构
 - 3.2 经验回放 DQNReplayer
 - 3.3 智能体 DoubleDQNAgent
 - 3.3.1 __init__ 方法
 - 3.3.2 build_network 方法
 - 3.3.3 learn 方法
 - 3.3.4 decide 方法
 - 3.4 play_qlearning 函数
 - 3.5 主循环
 - 3.5.1 训练模型
 - 3.5.2 测试模型
4. 效果展示

MountainCar连续版本

1. 环境
2. 算法
 - 2.1 Actor - Critic 网络
 - 2.2 DDPG 算法
 - 2.3 Ornstein-Uhlenbeck过程
3. 代码
 - 3.1 代码整体结构
 - 3.2 Actor 网络和 Critic 网络
 - 3.3 OUNoise 类
 - 3.3 DDPG 类
 - 3.3.1 __init__ 方法
 - 3.3.2 update 方法
 - 3.3.3 get_action 方法

3.3.4 `save_model` 方法和 `load_model` 方法

[3.4 训练模型](#)

[3.5 测试模型](#)

[3.6 超参数设置和调整](#)

4. 效果展示

[讨论与思考](#)

[关于离散版本](#)

[关于连续版本](#)

MountainCar离散版本

1. 环境

本次使用的环境是openai的[MountainCar-v0](#)

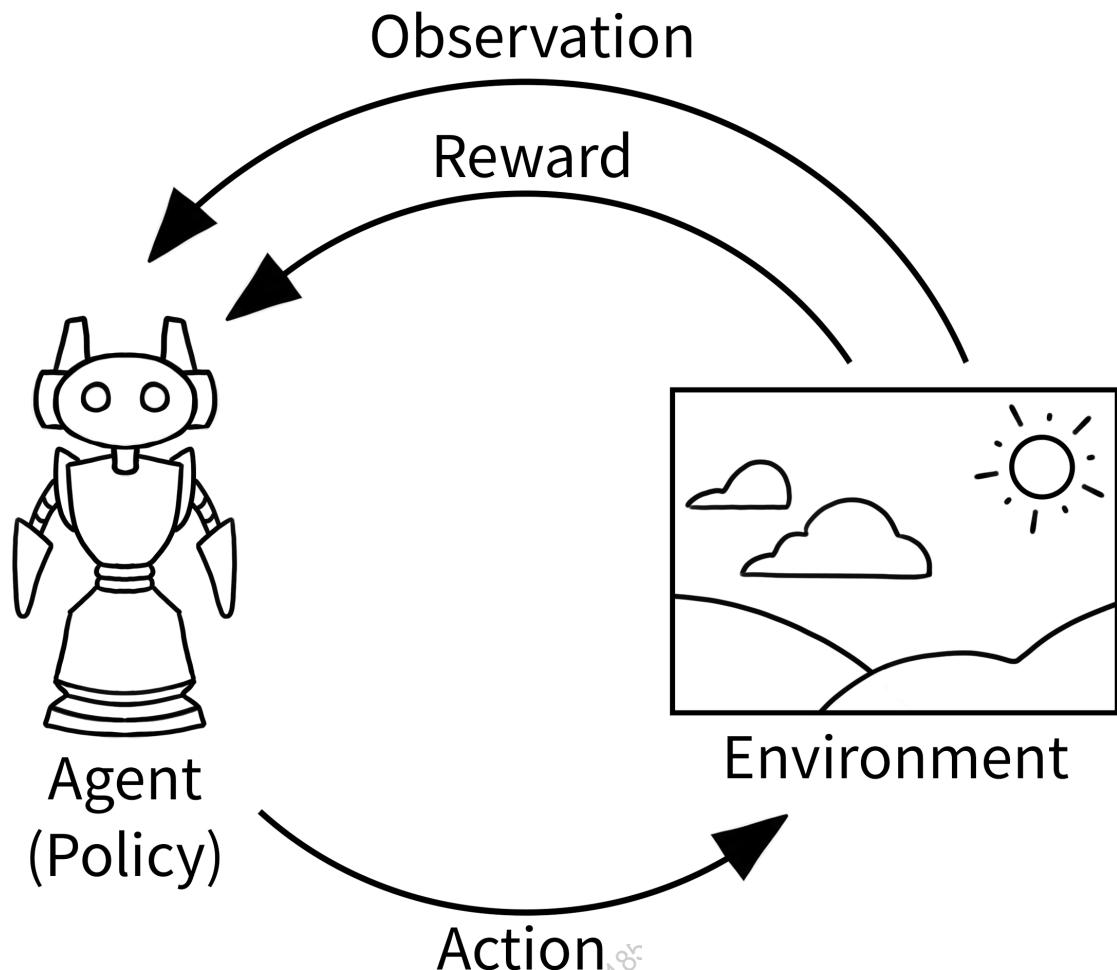
Mountain Car MDP是一个确定性的马尔科夫决策过程（MDP），它由一辆汽车被随机放置于一个正弦山谷的底部组成，汽车唯一可能的动作是向前或向后加速。MDP的目标是通过策略性地加速汽车，使其到达位于右山顶的目标状态。在gymnasium中有两个版本的Mountain Car领域：一个是使用离散动作的版本，另一个是使用连续动作的版本。这个版本是使用离散动作的版本。

该API包含四个关键函数：`make`、`reset`、`step` 和 `render`。

Gymnasium的核心是 `Env`，它是一个高水平的python类，代表强化学理论中的马尔科夫决策过程（这不是一个完美的重构，缺少MDP的几个组成部分）。

在gymnasium中，环境（MDPs）与 `Env` 一起被实现，`Wrappers` 可以改变传递给用户的结果。

下图的经典 "代理-环境循环" 是Gymnasium实现的强化学习的简化表示。



下面展示了环境的详细参数：

```
qhxx - show_env.py
1 import gymnasium as gym
2
3 env = gym.make('MountainCar-v0')
4 print('观测空间 = {}'.format(env.observation_space))
5 print('动作空间 = {}'.format(env.action_space))
6 print('位置范围 = {}'.format((env.unwrapped.min_position, env.unwrapped.max_position)))
7 print('速度范围 = {}'.format((-env.unwrapped.max_speed, env.unwrapped.max_speed)))
8 print('目标位置 = {}'.format(env.unwrapped.goal_position))
```

```
(qianghuaxuexidazuoye) E:\Code\qhxx>E:/Software/conda/envs/qian
ghuaxuexidazuoye/python.exe e:/Code/qhxx/test/py
观测空间 = Box([-1.2 -0.07], [0.6 0.07], (2,), float32)
动作空间 = Discrete(3)
位置范围 = (-1.2, 0.6)
速度范围 = (-0.07, 0.07)
目标位置 = 0.5
```

2. 算法

2.1 DQN

深度Q网络（Deep Q-Network, DQN）是强化学习中的一种算法，它结合了深度神经网络和Q学习。

Q-learning是一个值迭代算法，用于估计一个动作-价值函数，即Q函数。而深度神经网络用于对这个Q函数进行函数逼近，以便在复杂、高维度的状态空间中进行泛化。

DQN的主要步骤是：

1. **初始化网络**：初始化一个深度神经网络用于Q值的估计。这个网络接收环境的状态作为输入，输出各个可能动作的预期回报。
2. **经验回放**：创建一个经验回放缓冲区用于存储各个时刻的状态、动作、奖励和下一个状态。在每个时间步，先执行一个动作并将得到的经验存储在经验回放缓冲区，然后从经验回放缓冲区随机取样一些经验用于训练。
3. **计算目标Q值**：使用奖励和下一个状态的最大Q值（由另一个目标网络估计）计算目标Q值。
4. **优化网络**：使用目标Q值和网络预测的Q值之间的差（即TD误差）来优化网络的参数。
5. **同步网络**：定期用估计网络的参数来更新目标网络。

DQN的主要优点是可以处理高维度、连续的状态空间，并且可以通过经验回放技术解决强化学习中的样本相关性问题和非稳态分布问题。然而，DQN也有其局限性，例如它不能直接处理连续的动作空间，对超参数选择敏感，以及可能会遇到过度估计的问题等。

2.2 Double DQN

Double DQN 是对深度Q网络（DQN）的一个改进，其出发点是解决DQN的一个主要问题，即对优化行动价值（Q值）的过度估计。这个过度估计的问题源于DQN在更新Q值时，既使用相同的网络来选择行动，也用来计算该行动的Q值。

Double DQN 的工作方式与DQN类似，但在更新Q值时，它将行动选择和Q值计算分离。具体来说，**Double DQN** 使用一个网络（在线网络）来选择最佳行动，然后使用另一个网络（目标网络）来计算被选择行动的Q值。

Double DQN 的优势是减少了过度估计的问题，从而使得学习过程更稳定，并且在许多任务中能够提高性能。这是因为过度估计可能导致过度优化某些行动，从而忽视其他可能更优的行动。通过引入 **Double DQN**，我们可以更准确地估计行动的真实价值，从而做出更好的决策。

总的来说，使用 **Double DQN** 是因为它提供了更准确的Q值估计，使得学习过程更稳定，并且在许多任务中能够提高性能。

本次任务中使用 **Double DQN**。

2.3 网络结构

基于输入层、输出层和这两个隐藏层，**evaluate_net** 和 **target_net** 的完整结构如下：

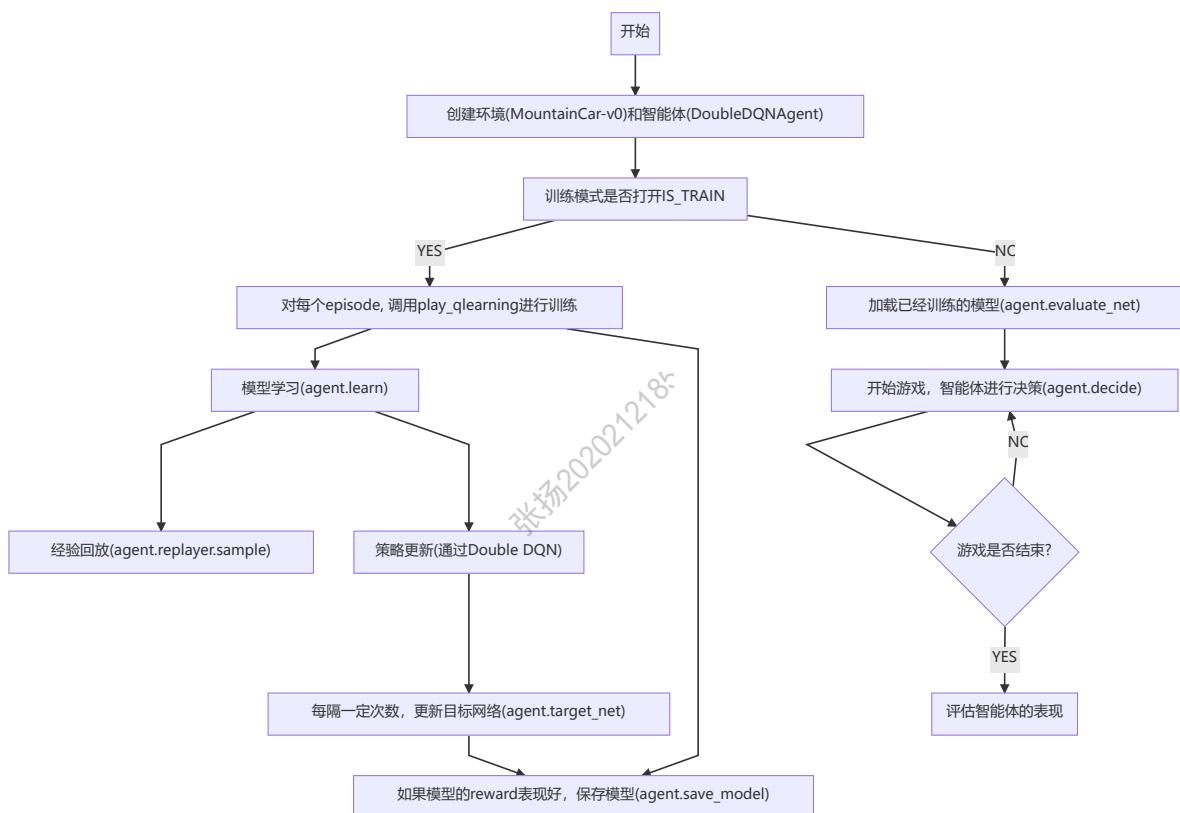
1. **输入层**：输入层的节点数等同于环境状态的维度，即2，一个是位置，一个是速度。
2. **隐藏层**：隐藏层用于处理输入信息，并提取有用的特征。在这个模型中，有两个隐藏层。第一个隐藏层有64个神经元，第二个隐藏层有128个神经元。这两个隐藏层都使用ReLU激活函数。
3. **输出层**：输出层的节点数等同于可能的行动数量，即3，分别是向左、向右和不动。

在训练过程中，输入层接收环境状态作为输入，通过隐藏层进行处理和特征提取，然后在输出层输出对应每个可能行动的预测Q值。神经网络的参数（权重和偏置）会根据反向传播算法和优化器（这里使用的是Adam优化器）进行调整，以减小预测Q值和目标Q值之间的差距。

3. 代码

3.1 代码结构

张扬202012185



3.2 经验回放 DQNReplayer

```
qhxx - main.py

1 class DQNReplayer:
2     def __init__(self, capacity):
3         self.memory = pd.DataFrame(index=range(capacity),
4                                     columns=['observation', 'action', 'reward',
5                                               'next_observation', 'done'])
5         self.i = 0
6         self.count = 0
7         self.capacity = capacity
8
9
10    def store(self, *args):
11        self.memory.loc[self.i] = args
12        self.i = (self.i + 1) % self.capacity
13        self.count = min(self.count + 1, self.capacity)
14
15    def sample(self, size):
16        indices = np.random.choice(self.count, size=size)
17        return (np.stack(self.memory.loc[indices, field])) for field in
18            self.memory.columns
```

DQNReplayer 类是用于实现经验回放（Experience Replay）机制的。

经验回放是DQN的关键组件之一，主要目的是打破数据之间的相关性和减少训练样本的浪费，通过存储智能体的经验并从中随机抽样进行学习。

下面是类的每个部分的详细解释：

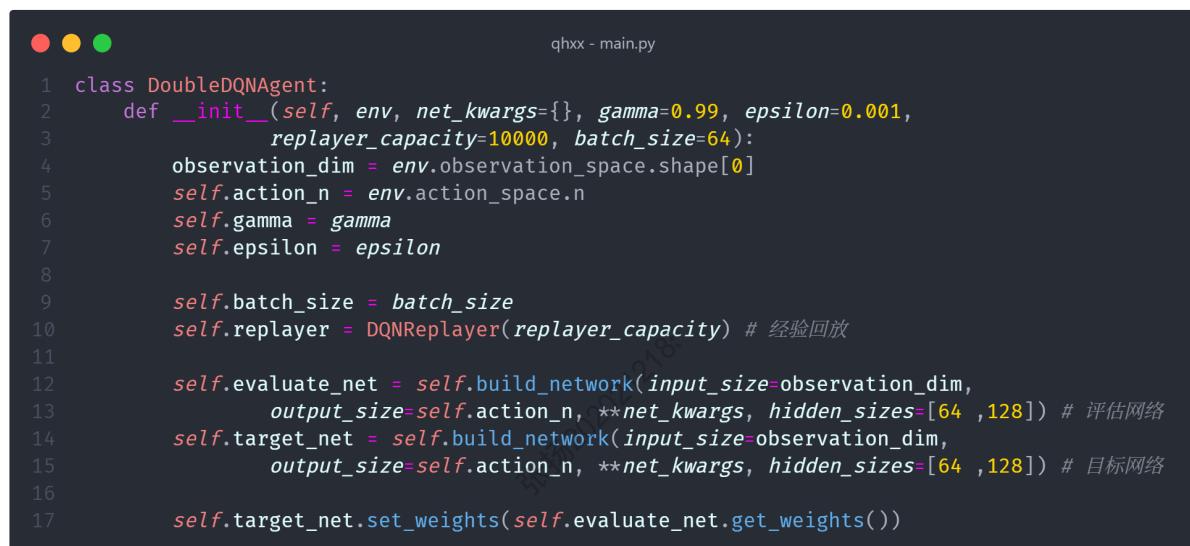
- **__init__(self, capacity)**：这是类的初始化函数。
`capacity` 参数决定了经验回放池的大小。经验回放池是一个 `DataFrame`，有5个列：`'observation'`, `'action'`, `'reward'`, `'next_observation'` 和 `'done'`，用于存储每一步的经验。`self.i` 是一个指针，指向当前要插入新经验的位置。`self.count` 记录了目前存储的经验的数量。
- **store(self, *args)**：这个方法用于将一次智能体的经验存储在经验回放池中。`*args` 包括 `'observation'`, `'action'`, `'reward'`, `'next_observation'` 和 `'done'` 这五个元素。每次存储经验后，指针 `self.i` 就会向前移动一个位置，如果到达池子的容量上限，就会回到起点（这是通过 `% self.capacity` 实现的，这也就是为什么叫它经验回放“池”，因为它是一个循环的）。同时，`self.count` 会增加1，直到达到 `self.capacity`。

- `sample(self, size)`：这个方法用于从经验回放池中随机抽取一些经验进行学习。参数 `size` 决定了抽取的数量。这个方法首先从当前存储的所有经验中随机抽取 `size` 个索引，然后返回这些索引对应的经验。这就是经验回放的主要操作，也是DQN中非常关键的一步。

整体来看，`DQNReplayer`类提供了经验存储和随机抽取的功能，它是实现经验回放机制的关键部分。

3.3 智能体 DoubleDQNAgent

3.3.1 `__init__` 方法



```

1 class DoubleDQNAgent:
2     def __init__(self, env, net_kwargs={}, gamma=0.99, epsilon=0.001,
3                  replayer_capacity=10000, batch_size=64):
4         observation_dim = env.observation_space.shape[0]
5         self.action_n = env.action_space.n
6         self.gamma = gamma
7         self.epsilon = epsilon
8
9         self.batch_size = batch_size
10        self.replayer = DQNReplayer(replayer_capacity) # 经验回放
11
12        self.evaluate_net = self.build_network(input_size=observation_dim,
13                                              output_size=self.action_n, **net_kwargs, hidden_sizes=[64, 128]) # 评估网络
14        self.target_net = self.build_network(input_size=observation_dim,
15                                              output_size=self.action_n, **net_kwargs, hidden_sizes=[64, 128]) # 目标网络
16
17        self.target_net.set_weights(self.evaluate_net.get_weights())

```

下面是对每个参数的解释：

- `env`：环境对象，用于获取观察的维度和动作的数量。
- `net_kwargs`：字典参数，包含了构建网络模型所需的关键参数。
- `gamma`：衰减因子，用于计算未来奖励的折扣值。
- `replayer_capacity`：经验回放的容量大小，即最多可以存储的经验数量。
- `batch_size`：每次更新模型时，从经验回放中抽取的批次大小。

在初始化函数中，主要进行了以下操作：

- `observation_dim`：从环境中获取观察的维度。
- `self.action_n`：从环境中获取可能的动作数量。
- `self.gamma` 和 `self.epsilon`：设置衰减因子和 ϵ 值。
- `self.batch_size`：设置每次训练的批次大小。

- `self.replayer` : 创建一个经验回放对象，初始化其容量为 `replayer_capacity`。
- `self.evaluate_net` : 创建一个用于评估的神经网络，输入维度为观察的维度，输出维度为动作的数量，隐藏层大小为`[64, 128]`。
- `self.target_net` : 创建一个用于目标的神经网络，其结构和 `self.evaluate_net` 相同。该网络用于生成Q-Learning更新公式中的目标Q值。
- `self.target_net.set_weights(self.evaluate_net.get_weights())` : 将评估网络的权重新复制给目标网络。

这个初始化函数为智能体的训练和决策提供了必要的设置和工具。

3.3.2 `build_network` 方法

```
qhxx - main.py
1 def build_network(self, input_size, hidden_sizes, output_size,
2                   activation=tf.nn.relu, output_activation=None,
3                   learning_rate=0.01): # 构建网络
4     model = keras.Sequential()
5     # 隐藏层
6     for layer, hidden_size in enumerate(hidden_sizes):
7         kwargs = dict(input_shape=(input_size,)) if not layer else {}
8         model.add(keras.layers.Dense(units=hidden_size, activation=activation, **kwargs))
9     # 输出层
10    model.add(keras.layers.Dense(units=output_size, activation=output_activation))
11    # 优化器
12    optimizer = tf.optimizers.Adam(lr=learning_rate)
13
14    model.compile(loss='mse', optimizer=optimizer)
15    return model
```

`build_network` 方法是用来构建神经网络模型的，这里的网络模型是用来作为Double DQN算法中的Q函数的近似。

以下是该函数的参数以及每一步操作的解释：

参数解释：

- `input_size` : 神经网络输入层的节点数，等同于环境状态的维度。
- `hidden_sizes` : 一个列表，包含了每个隐藏层的节点数。
- `output_size` : 神经网络输出层的节点数，等同于可能的行动数量。
- `activation` : 隐藏层的激活函数，默認為ReLU函数。
- `output_activation` : 输出层的激活函数，默認為None，表示输出层不使用激活函数。

- **learning_rate** : 学习率，用于在优化器中调整模型参数的步长。

方法的步骤如下：

1. `model = keras.Sequential()` : 初始化一个顺序模型对象。
2. 在隐藏层部分，函数遍历 `hidden_sizes` 列表，在模型中加入相应数量的全连接层（Dense）。每个全连接层的单元数为 `hidden_sizes` 中的一个元素，激活函数为 `activation`。对于第一个隐藏层，需要提供 `input_shape` 参数，即输入的维度。
3. `model.add(keras.layers.Dense(units=output_size, activation=output_activation))` : 在模型最后加入一个全连接层作为输出层，其单元数为 `output_size`，激活函数为 `output_activation`。
4. `optimizer = tf.optimizers.Adam(lr=learning_rate)` : 定义优化器，这里使用的是Adam优化器，学习率为 `learning_rate`。
5. `model.compile(loss='mse', optimizer=optimizer)` : 编译模型，设置损失函数为均方误差 (`mse`) 和优化器为Adam。

最后，函数返回创建好的模型。

总的来说，`build_network` 方法是用来创建一个多层感知器（MLP）模型的，这个模型将在后续的DQN训练中被用来估计Q值。

3.3.3 `learn` 方法



```

qhxx - main.py
1 def learn(self, observation, action, reward, next_observation, done):
2     self.replayer.store(observation, action, reward, next_observation,
3     done) # 存储经验
4     observations, actions, rewards, next_observations, dones = \
5         self.replayer.sample(self.batch_size) # 经验回放
6     next_eval_qs = self.evaluate_net.predict(next_observations)
7     next_actions = next_eval_qs.argmax(axis=-1)
8     next_qs = self.target_net.predict(next_observations)
9     next_max_qs = next_qs[np.arange(next_qs.shape[0]), next_actions]
10    us = rewards + self.gamma * next_max_qs * (1. - dones)
11    targets = self.evaluate_net.predict(observations)
12    targets[np.arange(us.shape[0]), actions] = us
13    self.evaluate_net.fit(observations, targets, verbose=0)
14
15    if done:
16        self.target_net.set_weights(self.evaluate_net.get_weights())

```

Learn 方法在智能体进行学习的过程中起核心作用，主要执行以下步

骤：

1. **存储经验**: 当前观察 (`observation`)，行为 (`action`)，奖励 (`reward`)，下一个观察 (`next_observation`)，以及“是否结束”标志 (`done`) 被存储到经验回放存储器 (`replayer`) 中。
2. **采样经验**: 从经验回放存储器中随机采样一批经验，每批大小为 `batch_size`。
3. **计算目标Q值**: 对于每个采样的经验，我们使用评估网络 (`evaluate_net`) 来预测下一个观察的Q值，然后选择具有最高Q值的行动 (`next_actions`)。然后我们使用目标网络 (`target_net`) 来预测下一个观察的Q值，只选择我们在前一步中选择的行动对应的Q值 (`next_max_qs`)。最后，我们计算目标Q值 (`us`) 为立即奖励加上折扣后的未来最大Q值，但是如果该经验对应的步骤是结束步骤，那么就没有未来的Q值，因此要乘以 `(1. - dones)`。
4. **更新评估网络**: 我们首先使用评估网络来预测每个采样经验的当前观察的所有可能行动的Q值，然后用我们刚刚计算出的目标Q值替换掉其中我们实际执行的行动的Q值，得到新的目标Q值向量 (`targets`)。然后我们使用这些观察和目标Q值向量来进行一次评估网络的训练。
5. **同步目标网络**: 如果该经验对应的步骤是结束步骤 (即 `done=True`)，我们就把评估网络的权重新复制到目标网络，使得目标网络保持最新。

3.3.4 `decide` 方法

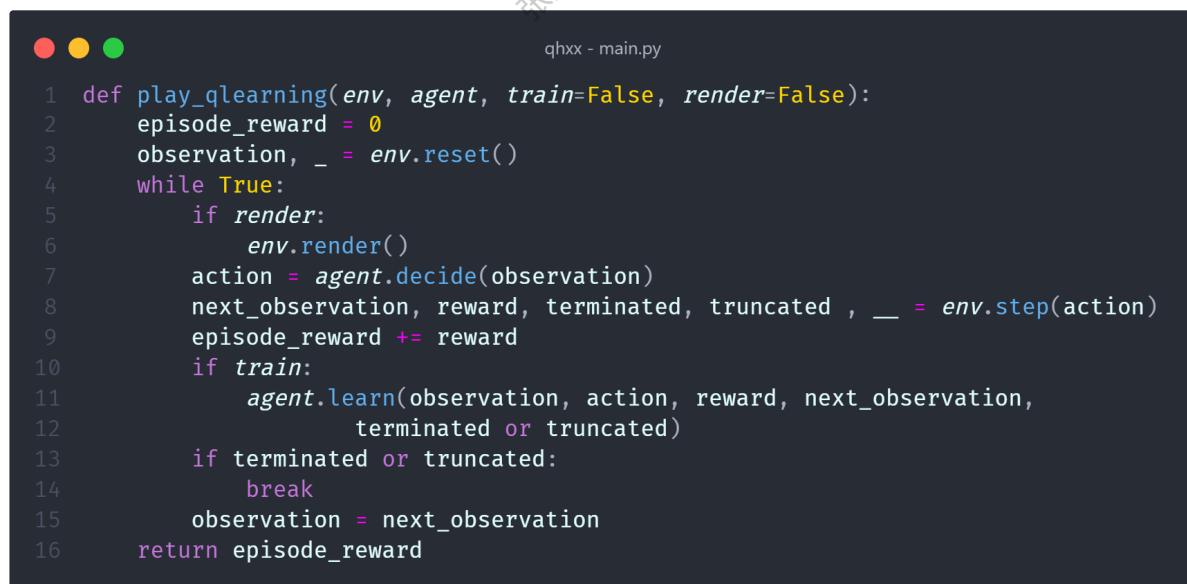
```
qhxx - main.py
1 def decide(self, observation): # epsilon贪心策略
2     if np.random.rand() < self.epsilon:
3         return np.random.randint(self.action_n)
4     qs = self.evaluate_net.predict(observation[np.newaxis])
5     return np.argmax(qs)
```

`decide` 方法是基于 **ϵ -greedy** (ϵ -greedy) 策略做出决策的方法。 ϵ -greedy策略是在探索 (exploration) 和利用 (exploitation) 之间寻找平衡的一种策略。以下是这个函数的详细步骤：

1. **探索**：以 ϵ 的概率随机选择一个行动。这里的 ϵ 被设定为一个介于0和1之间的值（如0.1），`np.random.rand()` 函数会生成一个从0到1的随机数，如果这个随机数小于 ϵ ，那么我们就随机选择一个行动。这可以帮助智能体在一开始的时候对环境有所了解，而不是过早地固化策略。
2. **利用**：以 $1-\epsilon$ 的概率选择当前知识中的最佳行动。
`self.evaluate_net.predict(observation[np.newaxis])` 会返回评估网络对每个可能行动的预测Q值，然后我们通过 `np.argmax(qs)` 选择Q值最大的行动作为最佳行动。

ϵ -greedy策略的目标是在探索和利用之间找到平衡，通过引入一定的随机性，让智能体有可能选择到不是当前最优但可能是长期最优的行动，从而有更大可能找到全局最优策略。

3.4 play_qlearning 函数



```
qhxx - main.py
1 def play_qlearning(env, agent, train=False, render=False):
2     episode_reward = 0
3     observation, _ = env.reset()
4     while True:
5         if render:
6             env.render()
7         action = agent.decide(observation)
8         next_observation, reward, terminated, truncated, __ = env.step(action)
9         episode_reward += reward
10        if train:
11            agent.learn(observation, action, reward, next_observation,
12                        terminated or truncated)
13        if terminated or truncated:
14            break
15        observation = next_observation
16    return episode_reward
```

`play_qlearning` 函数是执行一次Q-learning算法的过程。这里的一次过程指的是从环境的初始化状态开始，智能体不断地与环境互动，直到达到终止条件 (`terminated`) 或者被截断 (`truncated`)。函数的执行流程如下：

1. **初始化**: 首先初始化一个用来记录本次过程总奖励的变量 `episode_reward`，并获取环境的初始观察值 `observation`。
2. **循环互动**: 然后进入一个无限循环，每次循环都代表智能体与环境的一次互动。
 - **渲染环境**: 如果参数 `render` 为True，那么就渲染（显示）环境的当前状态。
 - **决策**: 智能体根据当前的观察值 `observation` 做出决策，得到要执行的行动 `action`。
 - **执行行动**: 智能体执行行动 `action`，环境返回新的观察值 `next_observation`、行动的奖励 `reward`，以及是否达到终止条件 `terminated` 或被截断 `truncated`。
 - **累计奖励**: 将本次行动的奖励 `reward` 累加到总奖励 `episode_reward` 中。
 - **学习**: 如果参数 `train` 为True，那么就用这一次的经验（包括初始观察、行动、奖励、新的观察和是否结束）来让智能体进行学习。
 - **判断结束**: 如果达到终止条件或被截断，那么就结束这次过程，并跳出循环。
 - **更新观察**: 将新的观察值 `next_observation` 更新为当前观察值 `observation`，为下一次循环做准备。
3. **返回总奖励**: 函数返回本次过程的总奖励 `episode_reward`。

3.5 主循环

3.5.1 训练模型

```

● ● ● qhxx - main.py
1 positions = []
2 velocities = []
3 env = gym.make('MountainCar-v0')
4 np.random.seed(0)
5 agent = DoubleDQNAgent(env)
6 IS_TRAIN = False
7 if IS_TRAIN:
8     for episode in range(1000):
9         episode_reward = play_qlearning(env, agent, train=True)
10        print('episode: ', episode, 'episode_reward:', episode_reward)
11
12        if episode_reward > -200:
13            agent.save_model(filepath=f'model_at_episode_{episode}_reward_{episode_reward}.h5')

```

训练阶段的目的是训练一个深度Q网络（Double DQN）智能体，使其能够学习如何最优地在MountainCar-v0环境中行动。

下面是训练过程的详细步骤：

1. 首先，环境被初始化，这里是MountainCar-v0环境，然后一个 **DoubleDQNAgent** 智能体被创建。
2. 然后开始最多1000次的训练周期（也就是最多1000个 **episodes**）。每个周期都会通过 **play_qlearning** 函数与环境进行一次互动过程。
3. 在 **play_qlearning** 函数中，每个周期开始时，环境会被重置。然后智能体会在环境中执行动作，直到环境终止（也就是达到环境的终止条件，例如到达目标位置或最大步数）或者在某一步被截断。
4. 在每个周期中，智能体会根据当前观察到的环境状态，选择一个动作。这个动作的选择有两种可能：一是随机选择（以 ϵ 的概率），二是根据当前的策略网络选择最优动作。
5. 当智能体选择动作并执行后，环境会返回新的状态、奖励以及是否终止的信息。然后智能体会将这个经验（包括旧的状态、动作、奖励、新的状态和是否终止）存储到经验回放缓冲区。
6. 然后，智能体从经验回放缓冲区中随机取出一批经验，然后用这些经验来更新自己的策略网络（通过梯度下降来最小化预测的Q值和目标Q值之间的差距）。这个步骤会反复进行，直到环境终止或被截断。
7. 在每个周期结束后，会打印出当前周期的总奖励。
8. 如果当前周期的总奖励大于-200，智能体会保存当前的模型。

这个过程一直持续，直到到达满意的训练效果。

3.5.2 测试模型

```

1 else:
2     agent.evaluate_net = keras.models.load_model('model_at_episode_99_reward_-134.0.h5')
3     positions, velocities = [], []
4     observation, _ = env.reset()
5     while True:
6         positions.append(observation[0])
7         velocities.append(observation[1])
8         action = agent.decide(observation)
9         print(action)
10        # print("位置 = {:.3f}\t速度 = {:.3f}\n".format(observation[0], observation[1]))
11        next_observation, reward, terminated, truncated, __ = env.step(action)
12        if terminated or truncated:
13            break
14        observation = next_observation
15
16    if next_observation[0] > 0.5:
17        print('成功到达')
18    else:
19        print('失败退出')
20
21    # 绘制位置和速度图像
22    fig, ax = plt.subplots()
23    ax.plot(positions, label='position')
24    ax.plot(velocities, label='velocity')
25    ax.legend()
26    plt.show()

```

在推理阶段，智能体将根据训练阶段学习到的策略，与环境进行交互并尝试达成任务。

以下是推理阶段的详细步骤：

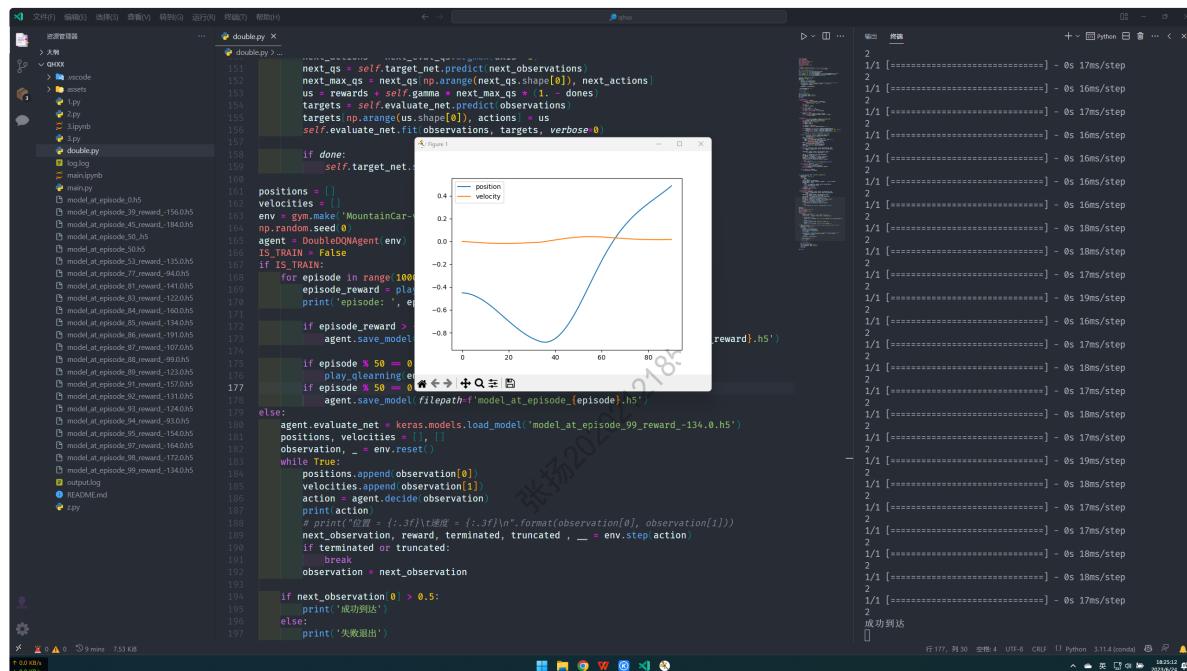
1. 训练好的模型被加载到智能体中。这个模型是在训练阶段保存的，包含了训练得到的权重。
2. 然后初始化两个列表 **positions** 和 **velocities**，用于保存每一步的位置和速度信息，以便之后进行可视化。
3. 接着重置环境，即将环境初始化到初始状态。
4. 接下来开始与环境交互的循环。在每一步中，智能体都会根据当前环境的状态选择一个动作。这个选择是根据训练好的模型进行的，不再涉及随机选择。
5. 执行智能体选择的动作后，环境会返回新的状态、奖励、是否终止的信息。这一步中智能体不会进行学习，即不再更新网络的权重。
6. 将每一步的位置和速度信息添加到 **positions** 和 **velocities** 列表中。
7. 持续以上步骤，直到环境返回的 **terminated** 或 **truncated** 为 **True**，即环境达到终止条件或者在某一步被截断。
8. 然后根据最后一步的位置信息判断智能体是否成功到达目标位置，打印出相应的信息。

9. 最后，使用matplotlib库绘制出智能体在每一步的位置和速度变化，进行可视化展示。

以上就是推理阶段的过程，这个过程主要是用来验证和展示智能体在训练阶段学习到的策略的效果，看是否能成功解决问题。

4. 效果展示

下图是选用的是第99个周期模型的测试结果，可以看到，小车能够先向左加速再向右加速，最终到达右侧山顶，这说明智能体已经学会了如何在 MountainCar-v0 环境中行动。



MountainCar连续版本

对于具有连续动作空间的问题，传统的Q学习和DQN不再适用，因为它们依赖于对每个可能的动作评估一个Q值。当动作是无限多个时，这就不再可能了。

不过，有一些已经被提出的解决方法可以处理连续动作空间的问题：

- 动作离散化：**尽管这种方法对于一些问题可能会导致精度损失，但在许多情况下，它仍然可以给出令人满意的结果。方法是将连续的动作空间离散化为有限数量的动作，然后使用这些离散动作来应用DQN或Q学习。

2. **策略梯度方法**: 这是一类直接在策略空间中进行优化的方法，例如 REINFORCE 算法、Actor - Critic 方法等。这类方法可以在连续动作空间中工作，因为它们不需要为每个可能的动作评估一个 Q 值。
3. **确定性策略梯度方法**: 如深度确定性策略梯度 (DDPG) 算法和连续深度 Q 学习 (CDQN)。这些方法结合了策略梯度和 Q 学习的思想，旨在解决连续动作空间的问题。DDPG 等方法将动作选择问题转化为一个参数化函数的优化问题，这个函数会直接输出最优的动作。
4. **正态分布参数化**: 有些策略使用神经网络来表示一个正态分布的参数（均值和方差），然后从这个分布中采样动作，这是连续控制问题中常见的一种做法，比如在使用 Proximal Policy Optimization (PPO) 算法时。

这些都是处理连续动作空间问题的常见方法，选择哪种方法需要根据具体任务和问题属性进行判断。

1. 环境

在连续版本的 MountainCar-v0 环境中，智能体的目标是让小车到达右侧山顶。与离散版本不同的是，这里的动作空间是连续的，智能体可以选择一个在 [-1, 1] 范围内的实数作为动作，这个实数表示小车的加速度。

2. 算法

2.1 Actor - Critic 网络

Actor - Critic 是一种强化学习算法，它是策略梯度方法和值迭代方法的结合体。其名称中的 "Actor" 和 "Critic" 分别对应两个主要的部分：行为者（策略）和评价者（值函数）。

- "Actor" (行为者) : Actor 负责确定 Agent 应该执行的动作，即决定策略。它根据当前的状态和可能的动作，决定下一步应该执行的动作。
- "Critic" (评价者) : Critic 负责评估 Actor 执行的动作的好坏，即通过值函数进行评估。它根据 Actor 选择的动作以及结果状态给出反馈，评估该动作的价值。

在 Actor - Critic 网络中，Actor 和 Critic 共同学习和更新：

- **Actor** 根据 **Critic** 的反馈进行学习和调整，试图选择更好的动作。
- **Critic** 则是通过观察 **Actor** 的行为和得到的奖励进行学习，以更准确地评估 **Actor** 的动作。

由于 **Critic** 的存在，**Actor** 可以利用更准确的估计进行更好的学习，这样可以解决传统策略梯度方法中的一些问题，例如高方差问题。同时，**Actor** 的存在可以直接输出策略，避免了值迭代方法中通过贪心策略选择动作的过程。

总的来说，**Actor - Critic** 结合了策略优化和值函数估计的优点，有更好的稳定性和效率。

2.2 DDPG 算法

深度确定性策略梯度（Deep Deterministic Policy Gradient, **DDPG**）算法是一种模型自由的离线策略学习算法。它是深度强化学习（DRL）中的一种**Actor-Critic**方法，适用于连续动作空间的问题。

以下是 **DDPG** 的一些关键概念、特点和优势：

1. **Actor-Critic结构**: **DDPG** 采用**Actor-Critic**结构。其中 **Actor**负责学习并输出确定性的最优策略，**Critic**负责学习并评估这个策略的价值函数。
2. **使用深度神经网络**: **DDPG** 使用深度神经网络来参数化**Actor**和**Critic**，利用神经网络强大的函数逼近能力，可以有效地处理高维度和连续的状态空间和动作空间。
3. **经验回放**: 为了打破数据之间的相关性，并提高数据的利用效率，**DDPG** 使用了经验回放（Experience Replay）技术。在训练过程中，会先将交互的经验数据保存到回放缓冲区，然后从中随机抽取样本进行学习。
4. **目标网络**: **DDPG** 引入了目标网络（Target Network）的概念，即对于每个网络（**Actor**和**Critic**），都维护一个目标网络，用于计算目标Q值，而不直接使用在线网络计算，以保证学习的稳定性。
5. **策略延迟更新**: **DDPG** 更新策略的频率通常小于更新Q值的频率，这也是为了保证稳定性。
6. **连续动作空间**: 与许多其他强化学习算法只能处理离散动作空间不同，**DDPG** 可以直接处理连续动作空间，这是其重要的优势之一。

7. **优势:** **DDPG** 在处理具有高维度连续状态空间和动作空间的任务时表现出了很高的效率和效果。同时，通过深度学习和经验回放的技术，可以有效地利用计算资源，提高数据的利用率。

总的来说，**DDPG** 是一种有效的深度强化学习算法，特别适用于连续动作空间的强化学习任务。

2.3 Ornstein-Uhlenbeck过程

在训练强化学习模型时，我们经常需要在行动选择中引入一定程度的随机性，以增强模型的探索能力。在 **DDPG** 算法这样的确定性策略中，策略直接输出一个确定性的行动，这会使得模型可能过早地陷入局部最优，而忽略其他可能的更优解。因此，我们通常会加入噪声来增强系统的探索性，也就是所谓的探索-利用权衡。

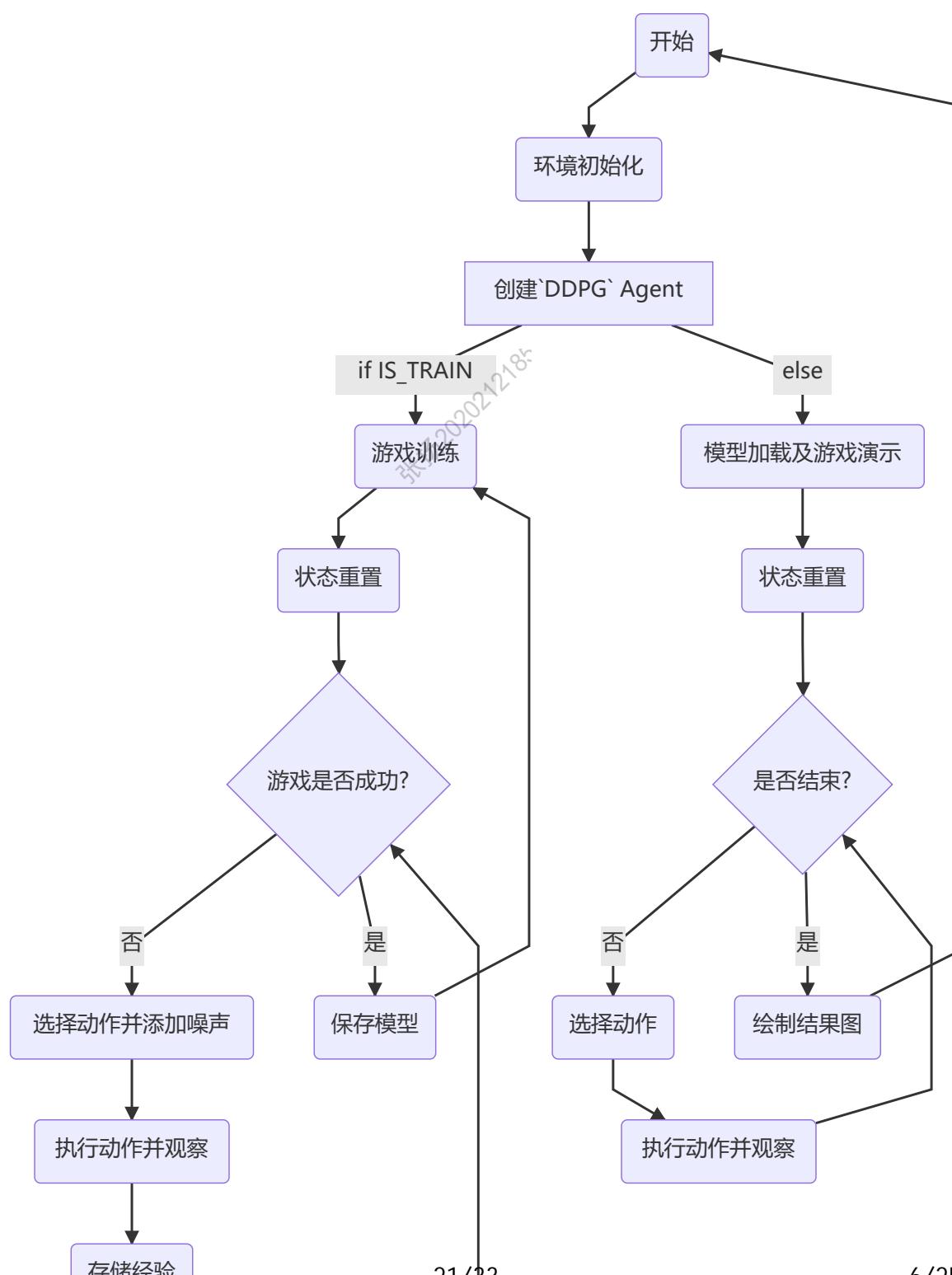
对于 **DDPG** 来说，特别是在连续动作空间中，一个常用的策略是在策略网络的输出上添加一个噪声项，使得每个行动都有微小的随机性。这个噪声通常来源于一个特定的噪声过程 (**Noise Process**)，比如Ornstein-Uhlenbeck过程。

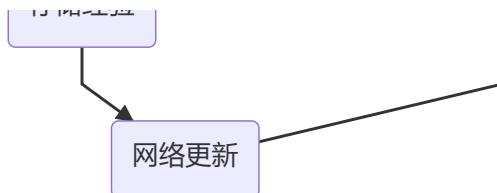
Ornstein-Uhlenbeck过程是一种常用的随机过程，它可以产生时间相关的噪声，这种噪声会自然地趋向于一个长期均值。这是一种平稳的 Gaussian过程，可以用来模拟在物理世界中的许多系统的噪声，比如空气阻力、市场力量等。在 **DDPG** 中，Ornstein-Uhlenbeck噪声过程被用来生成临时噪声，使得行动具有一定的随机性和探索性，从而增加策略的多样性，加强模型的探索能力。

在噪声过程中，当前的噪声状态是由上一时刻的噪声状态和一个随机项共同决定的，其中这个随机项通常来自于均值为 θ 的正态分布，然后与一个系数（通常称为噪声强度）相乘。这种随机项的引入使得噪声具有随机性，而上一时刻噪声状态的引入使得噪声具有时间相关性。因此，Ornstein-Uhlenbeck噪声过程可以看作是一种在确定性行动选择中引入随机性和探索性的手段。

3. 代码

3.1 代码整体结构





3.2 Actor 网络和 Critic 网络

```

qhxx - ddpg.py

1  class Actor(nn.Module):
2      def __init__(self, input_size, hidden_size, output_size):
3          super(Actor, self).__init__()
4          self.fc1 = nn.Linear(input_size, hidden_size)
5          self.fc2 = nn.Linear(hidden_size, output_size)
6
7      def forward(self, x):
8          x = F.relu(self.fc1(x))
9          return torch.tanh(self.fc2(x))
10
11
12 class Critic(nn.Module):
13     def __init__(self, input_size, hidden_size):
14         super(Critic, self).__init__()
15         self.fc1 = nn.Linear(input_size, hidden_size)
16         self.fc2 = nn.Linear(hidden_size, 1)
17
18     def forward(self, x):
19         x = F.relu(self.fc1(x))
20         return self.fc2(x)
  
```

这两个类是深度确定性策略梯度（DDPG）算法的核心组成部分，它们分别定义了 **Actor** 和 **Critic** 两个网络。

Actor 类定义了策略函数，也称为**行动者模型**，它将环境状态映射到特定的动作。它包含**两个全连接层**。输入的大小是**环境的状态空间**大小，输出的大小是**动作空间**的大小。在这个网络中，ReLU被用作第一个全连接层之后的激活函数，**tanh** 被用作最后一层的激活函数，以便输出在-1到1之间的动作。

Critic 类定义了价值函数，也称为**评论者模型**，它预测给定状态动作对的期望回报。这也是一个**两层的全连接网络**，但是它的输入是**环境的状态和动作的连接**，输出是一个标量，代表**给定状态-动作对的预期回报**。在这个网络中，**ReLU** 被用作第一个全连接层之后的激活函数。

这两个网络都使用了线性层 (`torch.nn.Linear`) 和非线性激活函数 (ReLU和Tanh)。这种组合为网络提供了足够的复杂性，使其能够学习复杂的行为策略和价值函数。

3.3 OUNoise 类

```
● ● ● qhxx - ddpg.py
1 class OUNoise(object):
2     def __init__(self, action_space, mu=0.0, theta=0.15, max_sigma=0.3, min_sigma=0.3, decay_period=100000):
3         self.mu      = mu                      # 均值
4         self.theta   = theta                    # 系数
5         self.sigma   = max_sigma               # 标准差
6         self.max_sigma = max_sigma              # 最大标准差
7         self.min_sigma = min_sigma              # 最小标准差
8         self.decay_period = decay_period       # 衰减周期
9         self.action_dim = action_space.shape[0]  # action的维度
10        self.low     = action_space.low        # action的最小值
11        self.high    = action_space.high       # action的最大值
12        self.reset()
13
14    def reset(self):
15        """
16        重置噪声
17        """
18        self.state = np.ones(self.action_dim) * self.mu
19
20    def evolve_state(self):
21        """
22        计算噪声
23        """
24        x = self.state
25        dx = self.theta * (self.mu - x) + self.sigma * np.random.randn(self.action_dim)
26        self.state = x + dx
27        return self.state
28
29    def get_action(self, action, t=0):
30        """
31        根据当前的action和时间t, 计算噪声并返回
32        """
33        ou_state = self.evolve_state()
34        self.sigma = self.max_sigma - (self.max_sigma - self.min_sigma) * min(1.0, t / self.decay_period)
35        return np.clip(action + ou_state, self.low, self.high)
```

OUNoise 类实现的是Ornstein-Uhlenbeck噪声，这是一种用于增加探索性的噪声，特别适合于有物理限制的问题和连续控制任务。

它是一种带有噪声的随机过程，其中下一步状态与前一步状态相关，这对于生成连续的动作非常有用。

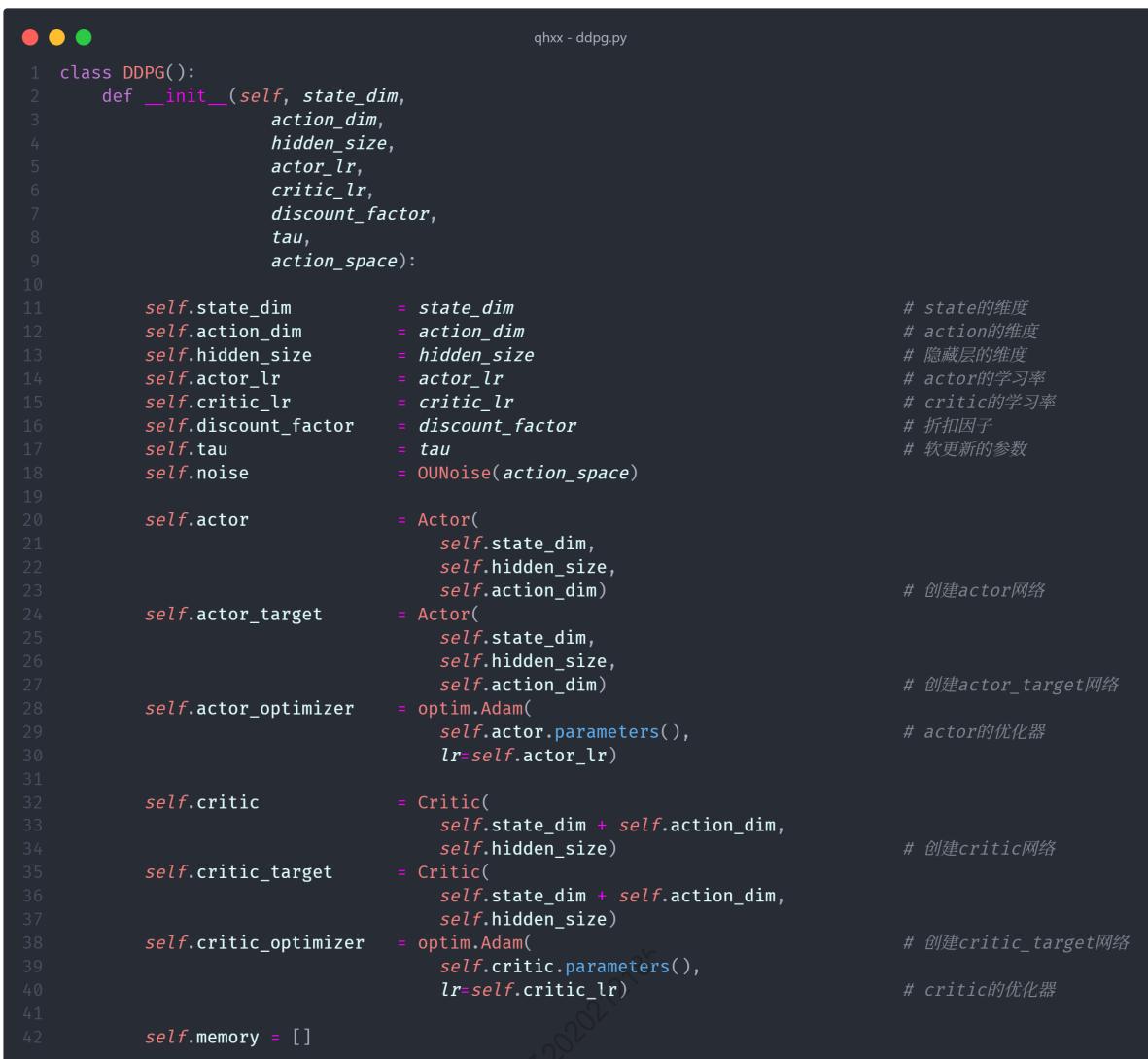
1. **`__init__`**：初始化方法，设置了一些参数，包括：

- **mu**：均值，表示需要围绕什么值产生噪声。
- **theta**：表示向着均值回归的速度，这是Ornstein-Uhlenbeck噪声的特点。

- **sigma** : 表示噪声的标准差。
 - **action_dim** : 动作的维度。
 - **low** 和 **high** : 动作的最小值和最大值, 用于限制噪声添加后的动作在合理范围内。
2. **reset** : 重置方法, 主要是把状态重置为均值。
3. **evolve_state** : 演变状态, 计算下一个状态的值, 实现了 Ornstein-Uhlenbeck 过程。具体地, 它根据当前状态计算下一个状态, 使用的公式如下:
- $dx = \theta * (\mu - x) + \sigma * \epsilon$
 - $x_{t+1} = x_t + dx$
- 其中, x_t 表示当前的状态, ϵ 是从标准正态分布中抽取的随机数。
4. **get_action** : 获得动作, 主要是根据当前的动作和时间步, 计算噪声并添加到当前动作上, 然后根据动作的取值范围对结果进行裁剪。同时, 随着时间的推移, 噪声的标准差会从最大值线性减少到最小值, 以降低探索性。

3.3 DDPG 类

3.3.1 **__init__** 方法



```

1  class DDPG():
2      def __init__(self, state_dim,
3                   action_dim,
4                   hidden_size,
5                   actor_lr,
6                   critic_lr,
7                   discount_factor,
8                   tau,
9                   action_space):
10
11         self.state_dim      = state_dim           # state的维度
12         self.action_dim     = action_dim          # action的维度
13         self.hidden_size    = hidden_size         # 隐藏层的维度
14         self.actor_lr       = actor_lr            # actor的学习率
15         self.critic_lr      = critic_lr           # critic的学习率
16         self.discount_factor= discount_factor    # 折扣因子
17         self.tau             = tau                 # 软更新的参数
18         self.noise           = OUNoise(action_space)
19
20         self.actor           = Actor(
21             self.state_dim,
22             self.hidden_size,
23             self.action_dim)                      # 创建actor网络
24         self.actor_target     = Actor(
25             self.state_dim,
26             self.hidden_size,
27             self.action_dim)                      # 创建actor_target网络
28         self.actor_optimizer  = optim.Adam(
29             self.actor.parameters(),
30             lr=self.actor_lr)                    # actor的优化器
31
32         self.critic           = Critic(
33             self.state_dim + self.action_dim,
34             self.hidden_size)                  # 创建critic网络
35         self.critic_target     = Critic(
36             self.state_dim + self.action_dim,
37             self.hidden_size)
38         self.critic_optimizer = optim.Adam(
39             self.critic.parameters(),
40             lr=self.critic_lr)                # 创建critic_target网络
41
42         self.memory = []

```

在 `__init__` 方法中，`DDPG` 类的初始化主要包括一些超参数的设定，以及相关的网络模型和优化器的构建。

以下是各个初始化的变量的含义和作用：

1. `state_dim`：状态空间的维度，也就是输入到 `Actor` 网络和 `Critic` 网络的状态的大小。
2. `action_dim`：动作空间的维度，也就是 `Actor` 网络的输出大小。
3. `hidden_size`：`Actor` 网络和 `Critic` 网络中隐藏层的大小。
4. `actor_lr` 和 critic_lr`：`Actor` 网络和 `Critic` 网络的学习率，这是优化器的超参数。
5. `discount_factor`：折扣因子，用于计算未来奖励的折扣累计值。
6. `tau`：用于网络参数的软更新，软更新是指将目标网络（`target network`）的参数向估计网络（`main network`）的参数平滑移动。

7. **action_space** : 动作空间, 它用于初始化Ornstein-Uhlenbeck噪声。

在 **__init__** 方法中, 也创建了 **Actor** 网络和 **Critic** 网络及其对应的优化器, 以及它们的目标网络。

在 **DDPG** 算法中, 目标网络是用于稳定学习过程的关键部分, 它是通过将网络参数逐步、平滑地向最新学习到的网络参数移动来更新的。

这里也创建了一个名为 **self.memory** 的空列表, 作为经验回放缓冲区用于存储和采样经验, 这是强化学习中常用的一种技术, 用于打破数据之间的关联性, 稳定训练过程。

最后, 还初始化了一个Ornstein-Uhlenbeck噪声实例, 用于在策略执行过程中增加一些探索性。

3.3.2 update 方法

```
● ● ● qhxx - ddpg.py
1 def update(self, batch_size):
2     # 检查经验池是否足够大, 能否提供足够的样本进行训练
3     if len(self.memory) < batch_size:
4         return
5
6     # 从经验池中随机取样
7     batch = random.sample(self.memory, batch_size)
8
9     # 将取样的数据转换成numpy数组的形式
10    state_batch, action_batch, reward_batch, next_state_batch, done_batch = map(np.array, zip(*batch))
11
12    # 将numpy数组转换成PyTorch张量的形式
13    state_batch = torch.FloatTensor(state_batch)
14    action_batch = torch.FloatTensor(action_batch)
15    reward_batch = torch.FloatTensor(reward_batch)
16    next_state_batch = torch.FloatTensor(next_state_batch)
17    done_batch = torch.FloatTensor(done_batch)
18
19    # 计算目标Q值
20    target_q = self.critic_target(torch.cat([next_state_batch, self.actor_target(next_state_batch)], 1))
21    target_q = reward_batch + (1 - done_batch) * self.discount_factor * target_q
22
23    # 计算当前的Q值
24    q = self.critic(torch.cat([state_batch, action_batch], 1))
25
26    # 更新critic网络, 最小化预测的Q值和目标Q值的均方误差
27    critic_loss = F.mse_loss(target_q, q)
28    self.critic_optimizer.zero_grad()
29    critic_loss.backward()
30    self.critic_optimizer.step()
31
32    # 更新actor网络, 最大化Q值
33    actor_loss = -self.critic(torch.cat([state_batch, self.actor(state_batch)], 1)).mean()
34    self.actor_optimizer.zero_grad()
35    actor_loss.backward()
36    self.actor_optimizer.step()
37
38    # 软更新target网络的参数, 让target网络慢慢接近实际网络
39    for target_param, param in zip(self.actor_target.parameters(), self.actor.parameters()):
40        target_param.data.copy_(self.tau * param.data + (1 - self.tau) * target_param.data)
41
42    for target_param, param in zip(self.critic_target.parameters(), self.critic.parameters()):
43        target_param.data.copy_(self.tau * param.data + (1 - self.tau) * target_param.data)
```

update 方法是实现 **DDPG** 算法的关键部分。它包含了算法的主要步骤：从经验池中采样，计算目标Q值和当前Q值，更新 **critic** 网络，然后更新 **actor** 网络，最后还进行了网络的软更新。

以下是具体的实现步骤：

1. 检查经验池是否有足够的样本可以采样：

如果经验池中的样本数少于设定的batch size，就不进行更新操作。

2. 从经验池中随机采样：

这个操作对于打破样本间的相关性非常重要。

3. 整理采样得到的批数据：

使用 **map** 函数和 **zip** 函数将样本按照状态、动作、奖励、下一状态、完成标志 (done) 的顺序整理，然后转为张量形式。

4. 计算目标Q值 (**Target Q value**) :

目标Q值的计算依据是贝尔曼方程，由即时奖励和折扣后的未来奖励两部分组成。未来奖励的计算使用了目标 **actor** 网络和目标 **critic** 网络。

5. 计算当前Q值 (**Current Q value**) :

当前的Q值是由 **critic** 网络基于当前的状态和动作计算得到的。

6. 更新 **Critic** 网络：

Critic 网络的更新是一个回归问题，即让当前的Q值尽可能接近目标Q值，这里使用均方误差 (Mean Squared Error) 作为损失函数，通过梯度下降方法更新网络参数。

7. 更新 **Actor** 网络：

Actor 网络的更新是一个策略优化问题，即要最大化期望的累积奖励。这里通过取负值将其转化为最小化问题，同时使用了 **Critic** 网络作为评估手段，具体表现为对 **Critic** 网络的输出求负均值。

8. 进行软更新 (**Soft Update**) :

这个步骤是为了让目标网络的参数向估计网络的参数平滑移动，从而提供稳定的目标Q值。

通过这些步骤，就可以完成 **DDPG** 算法的一个更新周期。

3.3.3 get_action 方法

```
qhxx - ddpg.py
1 def select_action(self, state):
2     state = torch.from_numpy(state).float()
3     action = self.actor(state).detach().numpy()
4     return self.noise.get_action(action)
# 将numpy数组转化为PyTorch张量
# 通过actor网络生成动作，并将PyTorch张量转化为numpy数组
# 通过噪声处理生成最终的动作
```

在 **DDPG** 类中，**select_action** 方法用于生成给定状态下的动作。具体的操作流程是：

首先，通过 **actor** 网络对给定状态进行处理，生成原始的动作。

然后，通过添加噪声，将原始的动作稍微进行扰动，生成最终的动作。

在这个函数中，噪声的添加是为了增强探索性，让agent不仅能利用已有的知识（exploitation），也能探索新的可能性（exploration）。

3.3.4 save_model 方法和 load_model 方法

save_model 和 **load_model** 这两个方法分别用于保存和加载模型。

save_model 方法：

在 **save_model** 方法中，将使用PyTorch的 **torch.save** 函数，将模型的参数（**state_dict**）以及优化器的状态保存到指定的文件中。这样，以后可以通过加载这些参数和优化器的状态，恢复模型的训练状态。

load_model 方法：

在 **load_model** 方法中，将使用PyTorch的 **torch.load** 函数，从指定的文件中加载模型的参数（**state_dict**）以及优化器的状态。并且，将这些参数和状态设置给当前的模型和优化器。这样，就能恢复之前的训练状态，继续进行训练。

3.4 训练模型

```

1 IS_TRAIN = False
2 if IS_TRAIN:
3     episodes = 500
4     for episode in range(episodes):
5         state, _ = env.reset()
6         done = False
7         actions = []
8         while 1:
9             action = agent.select_action(state)
10            actions.append(action.item())
11            next_state, reward, terminated, truncated, __ = env.step(action)
12            done = terminated or truncated
13            agent.memory.append((state, action, reward, next_state, done))
14            state = next_state
15            agent.update(32)
16            if terminated:
17                print(f'第{episode}局游戏成功, reward为{reward}')
18                os.makedirs(f'./models/episode{episode}', exist_ok=True)
19                agent.save_model(f'./models/episode{episode}/ddpg')
20                break
21            if truncated:
22                print(f'第{episode}局游戏失败, reward为{reward}')
23                break

```

当 `IS_TRAIN` 设为 `True` 时，代表模式为训练模式。此时程序会进行如下步骤：

1. 设置游戏回合数：在这段代码中，设定的游戏回合数为 `500`，也就是说 `agent` 将会在环境中进行 `500` 次尝试。
2. 对于每一回合，首先调用 `env.reset()` 函数初始化环境状态。
3. 然后，`agent` 在每个时间步进行以下操作：
 - 通过调用 `agent.select_action(state)` 函数，基于当前状态选择一个动作。
 - 调用 `env.step(action)` 函数，执行选择的动作，并获取返回的下一状态、奖励、以及游戏是否结束等信息。
 - 将经验（当前状态、动作、奖励、下一状态、游戏是否结束）添加到经验池。
 - 更新当前状态为下一状态。
 - 调用 `agent.update(32)` 函数进行学习，其中 `32` 是批量更新的大小。
 - 如果游戏结束（无论是成功还是失败），则结束当前回合，并输出结果信息。
 - 如果游戏被截断，那么结束当前回合并打印输出信息。

- 在每回合成功后，模型都会被保存，以便于在后续可以从这些保存点恢复训练或进行测试。

3.5 测试模型

当 `IS_TRAIN` 设为 `False` 时，代表模式为测试模式。此时程序会进行如下步骤：

- 首先，调用 `agent.load_model("./models/ ddpg ")` 函数，加载之前保存的模型。
- 然后，调用 `env.reset()` 函数初始化环境状态。
- 接着，`agent` 在每个时间步进行以下操作：
 - 通过调用 `agent.select_action(state)` 函数，基于当前状态选择一个动作。
 - 调用 `env.step(action)` 函数，执行选择的动作，并获取返回的下一状态、奖励、以及游戏是否结束等信息。
 - 将当前状态添加到列表中。
 - 更新当前状态为下一状态。
 - 如果游戏结束（无论是成功还是失败），则结束当前回合。
- 在回合成功后，将列表中的数据绘制成图表，以便于观察`agent`的行为。

3.6 超参数设置和调整

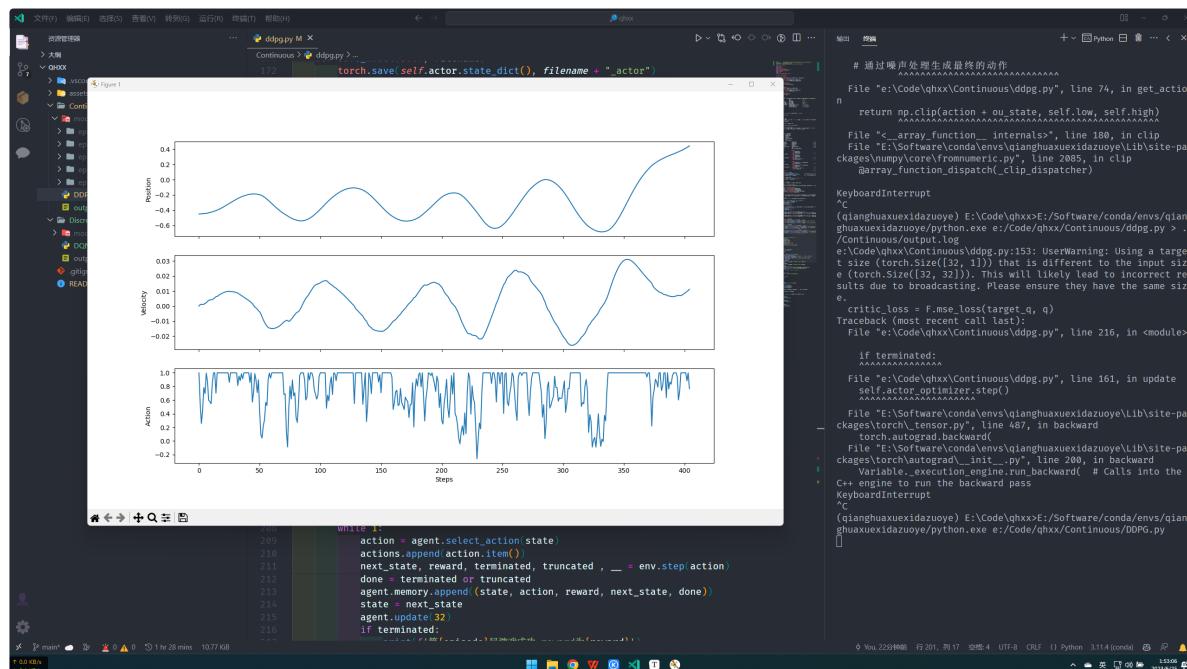
在这个 `DDPG` 实现中，以下是一些重要的超参数及其在代码中的设置：

- `state_dim`：状态维度。在`MountainCarContinuous-v0`环境中，每个状态由两个值（小车的位置和速度）组成，所以这个值被设置为2。
- `action_dim`：动作维度。在`MountainCarContinuous-v0`环境中，`agent`可以选择的动作是一个(-1, 1)连续的值，代表小车的推力，所以这个值被设置为1。
- `hidden_size`：隐藏层大小。这个参数决定了神经网络隐藏层的神经元数量。这个值的设置依赖于问题的复杂性以及计算资源。它被设置为64。

4. **actor_lr** 和 **critic_lr**：分别是 **actor** 网络和 **critic** 网络的学习率。学习率决定了模型参数在每次更新时变化的幅度。这里，**actor** 的学习率被设置为 $1e-4$ ，**critic** 的学习率被设置为 $1e-3$ 。通常，**critic** 的学习率被设置得比 **actor** 的学习率大，因为 **critic** 的学习通常更难，需要更大的学习率。
5. **discount_factor**：折扣因子，也被称为伽马因子（Gamma）。它决定了未来奖励在计算当前的预期奖励时的权重。这个值越接近1，agent考虑的未来奖励就越多。这个值被设置为 **0.99**，表示agent在做决策时会较多地考虑未来的奖励。
6. **tau**：软更新的参数，用于更新目标网络的参数。它的值通常接近于0，使得目标网络的更新过程更平滑。它被设置为 $1e-2$ 。
7. **episodes**：训练的回合数。这个值决定了agent在环境中尝试的次数。它被设置为 **500**。
8. **batch_size**：每次更新网络时从经验池中抽取的经验数量。这个值越大，每次更新时使用的经验就越多，但同时也需要更多的计算资源。它被设置为 **32**。
9. **max_sigma** 和 **min_sigma**：用于OUNoise类的参数，决定了噪声的变化范围。它们都被设置为 **0.3**。

4. 效果展示

选取了200个回合的训练结果进行展示，可以看到，agent已经学会了如何在环境中移动，最终成功到达了目标位置。



讨论与思考

关于离散版本

1. **探索和利用的平衡**: 在强化学习中, 我们需要找到一种有效的方式来平衡探索和利用。 ϵ 贪心策略是一种可能的方法, 但也可能有其他更有效的策略, 例如使用Upper Confidence Bound (UCB) 或者基于信息论的方法。
2. **经验回放的使用**: DQN中使用经验回放来打破样本之间的关联性, 增强算法的稳定性。但是经验回放需要大量的存储空间, 并且可能遇到样本利用率低、重要性采样偏差等问题。可以思考如何更好地利用和管理这些经验。
3. **性能评估**: 强化学习的性能评估可能会比监督学习更加复杂, 因为我们不仅关心最后的结果, 也关心到达这个结果的过程。例如, 在 MountainCar-v0环境中, 我们可能希望车辆能够尽快到达目的地, 而不仅仅是最终能够到达。

以上只是对这个问题的一些初步思考, 实际上还有许多其他的问题和方向值得深入探索。

关于连续版本

1. **关于Actor-Critic方法**: Actor-Critic方法结合了基于值的方法和基于策略的方法, 每种方法都有其优势和劣势。基于策略的方法直接优化策略性能, 但可能有高的方差; 基于值的方法有更低的方差, 但可能受到值函数近似的影响。Actor-Critic方法尝试从这两种方法中获得最佳效果, 使用Critic来降低基于策略方法的方差, 同时使用Actor直接优化策略性能。
2. **关于DDPG的应用**: DDPG是一种强大的算法, 可以处理连续动作空间问题。这使得DDPG在很多实际问题中有广泛的应用, 例如机器人控制、自动驾驶等。然而, DDPG的训练可能需要大量的环境交互, 并且可能受到不稳定和敏感的超参数设置的影响。
3. **关于噪声过程的使用**: 在确定性策略中引入噪声是一种重要的策略, 可以增加策略的探索能力。然而, 噪声的类型、大小以及如何引入噪声等问题都可能影响策略的性能。例如, DDPG中常用的Ornstein-Uhlenbeck过程生成的噪声具有时间相关性, 这可能适合模拟一些自

然现象，但在其他问题中可能并不适用。因此，如何选择和设计噪声过程是一个值得进一步研究的问题。

4. **关于超参数的设置：** DDPG的训练可能受到超参数设置的影响。例如，学习率、折扣因子、 τ 等超参数都会影响训练的稳定性和性能。尽管有一些常用的设置，但在特定问题上可能需要进行详细的超参数调整。因此，自动超参数调整可能是一个有价值的研究方向。
5. **关于训练和测试的策略选择：** 在训练阶段，我们希望策略具有足够的探索能力，但在测试阶段，我们可能更希望策略能够实现更好的性能。因此，训练和测试阶段的策略选择可能需要不同。例如，在测试阶段，我们可能会去掉噪声过程，以得到更好的性能。如何有效地在探索和利用之间进行权衡，以及如何设计训练和测试阶段的策略，都是重要的研究问题。

张扬20201218: