我现在需要对一个表面化学环境STM进行一个程序上的建模，可以理解为大概有三类对象，分子，表面和针尖。

表面可以吸附分子，

分子在表面会有规律的呈各类矩阵状的排列，

针尖可以在平面上移动，并且施加电压和隧穿电流。

分子的基本属性可以分为位置，角度，形状和状态。

表面的基本属性可以分为大小和取向；表面会有一个方法是可以将分子吸附到表面。

针尖的基本属性可以分为位置，偏压，电流；针尖有一个方法是可以用偏压，电流来操作分子，使得分子状态发生改变。

最后由一个大的类，叫env的类作为环境，将这些元素都整合在一起。由于分子的排列情况的不同，针尖对分子的操作的成功率会有影响。并且env中有方法可以把这些数学建模可视化（渲染）出来。

对此我用代码写了很小的一部分来提供给你作为参考，请你按照以上要求写一段代码，当然我很乐意看到有更加集成化的python包能够实现这些功能，因为我提到的这些功能就类似于游戏开发一样，我很好奇有没有相关的python包能够实现这些功能。

I now need to model a surface chemical environment STM on a program, which can be understood as roughly three types of objects: molecules, surfaces, and tips.

The surface can adsorb molecules,

Molecules will be arranged in various matrix shapes on the surface in a regular pattern,

The tip can move on a plane and apply voltage and tunneling current.

The basic properties of molecules can be divided into position, angle, shape, and state.

The basic properties of the surface can be divided into size and orientation; There is a way to adsorb molecules onto the surface.

The basic properties of a tip can be divided into position, bias voltage, and current; One method of using a tip is to manipulate molecules with bias voltage and current to change their state.

Finally, a large class called env is used as the environment to integrate these elements together. Due to the different arrangements of molecules, the success rate of molecular manipulation by the tip can be affected. And there are methods in env to visualize (render) these mathematical models.

I have written a small part of the code for you as a reference. Please write a code according to the above requirements. Of course, I would be happy to see more integrated Python packages that can implement these functions, because the functions I mentioned are similar to game development. I am curious if there are any related Python packages that can implement these functions.

不错，现在基础框架搭出来了。我现在需要对tip和分子的交互之间制定一些更加明确的规则：

0.分子有各种形状，有圆形的，方形的，等腰三角形的；不同形状有着不同的参数，圆形的分子的size是直径，方形的size是边长，等腰三角形size是边长。size默认为1.

1.分子的状态有3种：0（未反应），1（已反应），2（被破坏）；分子在被tip操作后会只有一定几率发生状态改变，分子状态只有0→1（成功反应），0→2（失败反应），1→2（反应后被破坏）这三种状态改变方式，且不可逆。不同的分子在不同的状态下用不一样的颜色来表示。

2.分子被tip操作改变状态的几率受到多方面影响：

1.tip操作分子的位置，

2.bias大小，

3.电流大小。

举例说明，如果操作参数太激进（过大的大电流、偏压等），分子可能会0→2（失败反应），如果操作参数太温和或者操作位置太偏，分子发生状态改变的几率就很低或者需要花费很长时间才能状态改变。

所以综上所述，我们希望能够利用合适的tip操作参数，和操作位置使得分子能够以较高的概率能够实现0→1（成功反应）的状态改变。

以下是我提供的一些参数：最佳的反应位点在是在分子中心，反应电流是1nA，反应电压是3.45V。你可以结合这个最佳参数和之前的规则（参数太激进分子会失败反应，操作参数太温和或者操作位置太偏发生状态改变的几率就很低）些随便建模，细节我们后面再商量。

我针对你的代码做出了一点小的改动，我将它发送给你。

现在我有了新的要求，分子类种应当多加几个属性，分子应该有一个属性是角度，这个角度默认为0，即分子朝上，最大为359，如果填入的值过大或者过小，则将加或减360将这个值调整带0-359之间。这个角度再后面渲染的时候会用到。

还有一个需求，我希望这些分子对象还有一个方法就是给分子添加“关键点”，这个关键点是相对于分子中心而言的。举一个简单的例子，一个三角形的分子，我们可以给这个分子添加四个关键点（分子中心已经三角形分子的三个角）。一但添加了关键点，这些关键点会随着分子角度的转动而改变。

渲染的时候把定义的关键点也显示出来。

针对于现在代码框架，我有一些想要修改的地方。

1. 分子属性不仅有key\_points，还要有render\_points，render\_points负责保存分子的渲染点，当需要渲染分子的时候，就会直接从render\_points中调取参数，分子旋转时key\_points和render\_points都需要旋转。
2. 根据形状的不同，key\_points和render\_points的部分应该在Molecule类的\_\_init\_\_里面完成。
3. Molecule类中应该将get\_rotated\_points方法改为rotate方法，这个方法负责将分子中所有的点位的旋转。
4. Molecule类应该添加一个move方法，使得改变分子位置（当然分子中的所有点位也要移动）
5. 将Molecule类的实例化操作放到Surface类的absorb方法里。

请帮我画一下sigmod函数以及正态分布叠加的图像

现在一个模拟的表面化学环境已经被我被搭建起来了，只需要输入动作就得到新的状态（分子反应状态以及反应时间）。

现在我需要写一个SAC的强化学习模型来操作tip和我的表面化学环境上的Molecule进行互动。

Tip对于Molecule的互动参数有四个：X（横坐标），Y（纵坐标），V（偏压），A（隧穿电流）

动作空间如下：

X：-200-200

Y：-200-200

V：3-4

A：1.7-0.3

Tip对于Molecule的交互后会产生两个可观测值

1. 分子状态：0（没反应）1（成功反应） 2（被破坏）

2. 反应时间：0-8

分子的默认状态为0。我们的目标是在动作空间中找到最合适的反应参数，使得有较高的反应成功率（0→1）尽可能杜绝分子被破坏，以及较短的反应时间。

所以我定义了以下分段的奖励函数

R = 1-0.2\*t （0→1）成功反应

R =-0.2\*t （0→0）没反应

R =-5-0.2t （0→2）分子被破坏，反应失败

T是反应时间

这个框架看上去没什么大问题，现在我把我写的代码发送给你，其中有一个Env类，请帮我在其中多加一些方法使得这个Env类可以用来和SAC强化学习算法相适配。

现在我将env.py和SAC.py发送给你，我想在每次强化学习模型对环境操作的时候将环境渲染出来，我应该怎么改动代码？

现在一个模拟的表面化学环境已经被我被搭建起来了（env.py）。

现在我需要写一个SAC的强化学习模型来操作tip和我的表面化学环境上的Molecule进行互动。

Tip对于Molecule的互动参数有四个：X（横坐标），Y（纵坐标），V（偏压），A（隧穿电流）

动作空间如下：

X：-200-200

Y：-200-200

V：3-4

A：1.7-0.3

Tip每次对于Molecule的交互后会产生两个可观测值

1. 分子状态：0（没反应）1（成功反应） 2（被破坏）

2. 反应时间：0-8

分子的默认状态为0。我们的目标是在动作空间中找到最合适的反应参数，使得有较高的反应成功率（0→1）尽可能杜绝分子被破坏，以及较短的反应时间。

奖励函数我已经在env.py定义过了

请你根据我给你的env.py代码来构建SAC的强化学习模型。

我希望将训练过程中的action都用plot实时显示出来，方便我观察智能体选择action的过程。action中有4个量，x，y，v, a其中。x，y plot成一张图 。 v, a plot成另一张图。

如果action后的分子的state反应成功，则plot的点表示为半透明绿色点；反应失败表示为半透明绿红色点，分子的state没有变化，则标记为半透明灰色。

这样的代码太耦合了，后期难以维护。我想到了一个新的代码结构，请你帮我实现：

在train方法中可以选择是否要展示action，如果要展示则创建一个新的进程来专门负责plot，创建一个新的队列用来传输(state, action, reward, next\_state, done)这些信息来提供给这个新进程数据来画图，就像生产者和消费者模式一样。

关于我的原始代码，我的分析如下。

我定义三个不同的loss，根据q\_loss的定义如下

with torch.no\_grad():

next\_actions, next\_state\_log\_pi = self.select\_action(next\_states)

next\_actions = torch.tensor(next\_actions, dtype=torch.float32, device=self.device)

q\_target1 = self.target\_q\_net1(torch.cat([next\_states, next\_actions], dim=-1))

q\_target2 = self.target\_q\_net2(torch.cat([next\_states, next\_actions], dim=-1))

q\_min = torch.min(q\_target1, q\_target2) - self.alpha \* next\_state\_log\_pi

q\_target = rewards + (1 - dones) \* self.gamma \* q\_min

q1 = self.q\_net1(torch.cat([states, actions], dim=-1))

q2 = self.q\_net2(torch.cat([states, actions], dim=-1))

q\_loss1 = ((q1 - q\_target) \*\* 2).mean()

q\_loss2 = ((q2 - q\_target) \*\* 2).mean()

q\_loss = q\_loss1 + q\_loss2

而在

# update critic

self.q\_optimizer1.zero\_grad()

self.q\_optimizer2.zero\_grad()

q\_loss.backward()

self.q\_optimizer1.step()

self.q\_optimizer2.step()

这些代码的运行是没有错误的，q\_loss是可以正常经历backward的。

但是在policy\_loss的定义代码处

# policy-loss

new\_actions, log\_prob = self.select\_action(states)

new\_actions = torch.tensor(new\_actions, dtype=torch.float32, device=self.device)

# log\_prob = log\_prob.to(self.device)

q1\_new = self.q\_net1(torch.cat([states, new\_actions], dim=-1))

q2\_new = self.q\_net2(torch.cat([states, new\_actions], dim=-1))

min\_q\_pi = torch.min(q1\_new, q2\_new)

policy\_loss = (self.alpha \* log\_prob - min\_q\_pi).mean()

在对policy-loss却不能进行经历backward

# update policy

self.policy\_optimizer.zero\_grad()

policy\_loss.backward() #此行报错

self.policy\_optimizer.step()

原因在哪？

如果按照你说的是select\_action的错误，那么q\_loss应该也是不能backward的

所以我不认为是select\_action的定义有问题。请你仔细分析其他修改的可能性。

我有一个简单的环境，定义如下，环境的state一共有三种情况，初始状态是0，与环境的互动的动作是一个4维的动作。互动之后，环境会变成以下状态分别是0（没变化），1（成功互动），2（失败互动）。每当state改变时，一个episode结束。如果state一直没变，则继续互动。所以一个典型episode中state的改变为0→0→…→0→1,或者0→0→…→0→2.。

据我理解，如果我采用HER来在这些经验中重新采样，那么过程是不是如下：假设我有一个episode：s和a的过程如下：s1(0)→a1→s2(0)→a2→s3(0)→a3→s4(1)成功互动。如果对这个episode加入HER采样，我是不是能采出下列buffer：

s1(0)→a1→s2(0)→a2→s3(1) 成功互动，

s1(0)→a1→s2(1) 成功互动.

我的理解有错吗？

还是说，我如果将原来的0状态看作是新的目标所以应该采出以下buffer：

s1(1) 成功互动→a1→s2(1) 成功互动→a2→s3(1) 成功互动

HER那种情况是对的？如果不对，那么针对我的例子，什么样的采样方式才是真正的HER？

我需要用python写一个函数。找出分子取向。首先，我已知分子的中心点位坐标，其次我已利用语义分割得到了分子的mask，且我已知分子的大小与形状（近似一个正方形）。我希望有一个函数可以通过分子中心点，分子的mask，分子的大小与形状等信息来拟合出正方形的取向。

假设我有一个环境，这个环境是达到最终目标的一共有两条路径，这两条路径都是正确的，属于殊途同归。我用这个环境训练了SAC，我希望在这个智能体在训练完成后能让我指定按照哪一条路径达到终点，我应该如何训练？

在我的强化学习环境中state, action, reward, next\_state, done, info这些信息会存放到buffer当中。但是我的环境事实上是一个空间上多重对称的环境,(对象是个正方形)

State的格式是array([a, b, c, d, True/False])；abcd对应于正方形对象的四个反应点位，最后一位是反应路径path的某个参数，不用在意。

Action的格式是array([ x, y , V, I])，分别代表对的环境对象的动作参数，x，y是位置坐标，v是电压，I是电流。

Reward由reward\_culculate (state, next\_state, 0)函数计算得出

next\_state格式同State格式一致

done控制是否结束当前episode

info也来源于reward\_culculate的返回值（函数reward\_culculate返回格式是Reward，info）

分子的初始状态是(0,0,0,0,path), 我希望反应过后会变成(1,0,0,0,path)，即第一个点位发生了反应。在训练初期，反应状态常常无法按预期反应比如会反应成(0,1,0,0,path)。按照强化学习以往的经验来说，这是错误的路径所以需要给出负Reward。但由于环境对象的高度对称，我们可以通过旋转和对称操作联系Action中的x和y将原来的失败案例进行数据增强从而得到更多的经验（数据增强）。我举例如下：

真实发生的环境变化：

(0,0,0,0,path) →(-1,-3,v,i)→ (0,0,1,0,path)→-1

由于旋转和对称数据增强得到的虚拟的环境变化：

(0,0,0,0,path) →(-1,3,v,i)→ (0,0,1,0,path) →-1 (环境以y轴对称)

(0,0,0,0,path) →(-3,1,v,i)→ (0,0,0,1,path) → -1 (环境顺时针旋转90度)

(0,0,0,0,path) →(-3,-1,v,i)→ (0,0,0,1,path) → -1 (环境以y=x对称)

(0,0,0,0,path) →(1,3,v,i)→ (1,0,0,0,path) → +2 (环境顺时针旋转180度)

(0,0,0,0,path) →(-1,3,v,i)→ (1,0,0,0,path) → +2 (环境以x轴对称)

(0,0,0,0,path) →(3,1,v,i)→ (0,0,0,1,path) → -1 (环境以y=-x对称)

(0,0,0,0,path) →(3,-1,v,i)→ (0,0,0,1,path) → -1 (环境顺时针旋转270度)

所以，以上述情况来看，与环境的一次互动就可以或的更多exp，并且可以将失败的经历转化为成功的经历。

下面我们接着来看，我想在假设的成功的反应基础上，继续反应，首先我会将真实环境的

(0,0,1,0,path) →(环境顺时针旋转180度) → (1,0,0,0,path) (环境顺时针旋转180度)

这个作为成功案例继续往下反应，（我将记录好我对环境的对称操作，方便最后追溯真正的action的x和y）

(环境顺时针旋转180度)发生的环境变化：

(1,0,0,0,path) →(-4,-2,v,i)→ (1,0,0,1,path)→-3

真实情况：(1,0,0,0,path) →(-4,-2,v,i)→ (1,0,0,1,path)

由于对称数据增强得到的虚拟的环境变化：

(1,0,0,0,path) →(4,-2,v,i)→ (1,1,0,0,path)→+4 (环境以y轴对称)

由于(1,0,0,0,path)状态的对称轴只有这一条所以只能得到一条对称数据增强exp

需要提醒的是action (-4,-2,v,i) action在作用于环境之前，由于先前的(环境顺时针旋转180度)的记录在案的操作，所以在将这个action真正作用于环境时需要先逆时针旋转180度，即变为action (4,2,v,i)

让我们更进一步把，首先我依然选择成功案例

(1,0,0,0,path) →(4,-2,v,i)→ (1,1,0,0,path)→+4 (环境以y轴对称)

让我们继续往下反应，不要忘记对环境做过的操作(环境顺时针旋转180度，环境以y轴对称)

真实发生的环境变化：

(1,1,0,0,path) →(-5,-1,v,i)→ (1,1,0,1,path)→-4

由于对称数据增强得到的虚拟的环境变化：

(1,1,0,0,path) →(-1,-5,v,i)→ (1,1,1,0,path)→+6 (环境以y=x轴对称)

由于(1,1,0,0,path)状态的对称轴只有y=x这一条所以只能得到一条对称数据增强exp

以此类推

(1,1,1,0,path) →(-5,2,v,i)→ (1,1,1,1,path)→+6

数据增强

(1,1,1,0,path) →(-5,-2,v,i)→ (1,1,1,1,path)→+6 (环境以x轴对称)

反应完成

通过以上案例可以总结出

总结来看目前合法的状态为5种，即：

(0,0,0,0,path)，(1,0,0,0,path)，(1,1,0,0,path)，(1,1,1,0,path)，(1,1,1,1,path)

每当环境状态不在这5种之中时，首先将当前环境旋转或对称操作将其变为合法状态

比如 现在输入的状态是(1,0,0,1,path)那么可以利用环境旋转90度将状态变成输入到(1,1,0,0,path)强化学习模型中，当强化学习模型输出action后，将action 中的xy部分逆时针旋转90度再与环境交互。

当旋转90度，状态的前4位，向后轮换即(a,b,c,d,path)→(d,a,b,c, path), 逆时针则反向轮换

当x轴翻转时，(a,b,c,d,path)→(d,c,b,a, path)

当y轴翻转时，(a,b,c,d,path) →(b,a,d,c, path)

当沿 y=x轴翻转时 (a,b,c,d,path)→(a,d,c,b,path)

当沿 y=-x轴翻转时(a,b,c,d,path)→(a,c,b,d,path)

综上所述，我需要几个功能，1.输入一个exp，帮我把exp数据增强。返回exp以及所有exp数据增强

2.输入一个状态，返回合法状态，以及旋转角度

我的代码如下：

# Replay Buffer

class ReplayBuffer:

    def \_\_init\_\_(self, max\_size, env):

        self.buffer = deque(maxlen=max\_size)

        self.env = env

        self.now = time.strftime('%Y-%m-%d\_%H-%M-%S', time.localtime(time.time()))

    def add(self, state, action, reward, next\_state, done, info):

        self.buffer.append((state, action, reward, next\_state, done, info))

    def sample(self, batch\_size):

        success\_ratio = 0.3

        latest\_ratio = 0.2

        random\_ratio = 0.5

        num\_success = round(success\_ratio \* batch\_size)

        num\_latest = round(latest\_ratio \* batch\_size)

        # -------------------------------

        # 筛选所有认为是 success 的数据，即反应不在 ["bad", "wrong", "non", "skip"] 中

        success\_all\_idx = [idx for idx, experience in enumerate(self.buffer)

                            if experience[-1].get("reaction") not in ["bad", "wrong", "non", "skip"]]

        # 将 reaction 为 "mol 4→5" 的数据单独筛选出来

        mol45\_idx = [idx for idx in success\_all\_idx if self.buffer[idx][-1].get("reaction") == "mol 4→5"]

        actual\_success = min(num\_success, len(success\_all\_idx))

        sampled\_success\_experiences = []

        if actual\_success > 0:

            # 至少要求三分之一的 success 数据是 "mol 4→5"

            req\_mol45 = round(actual\_success / 3) + 1

            if len(mol45\_idx) >= req\_mol45:

                sampled\_mol45 = random.sample(mol45\_idx, req\_mol45)

            else:

                # 如果数量不足，则全部采样

                sampled\_mol45 = mol45\_idx

            remaining\_needed = actual\_success - len(sampled\_mol45)

            remaining\_candidates = list(set(success\_all\_idx) - set(sampled\_mol45))

            if remaining\_needed > 0 and len(remaining\_candidates) >= remaining\_needed:

                sampled\_rest = random.sample(remaining\_candidates, remaining\_needed)

            else:

                # 如果剩余候选不足，则直接取 success\_all\_idx 内未采样的部分（可能少于要求）

                sampled\_rest = list(set(success\_all\_idx) - set(sampled\_mol45))

                sampled\_rest = sampled\_rest[:remaining\_needed]

            sampled\_success = sampled\_mol45 + sampled\_rest

            sampled\_success\_experiences = [self.buffer[idx] for idx in sampled\_success]

        else:

            sampled\_success\_experiences = []

        # -------------------------------

        sampled\_latest\_experiences = []

        if num\_latest > 0:

            sampled\_latest\_experiences = list(self.buffer)[-num\_latest:]

        remaining = batch\_size - actual\_success - len(sampled\_latest\_experiences)

        if remaining > 0:

            exclude\_indices = set()

            try:

                exclude\_indices.update(sampled\_success)

            except:

                pass

            exclude\_indices.update(range(len(self.buffer) - num\_latest, len(self.buffer)))

            available\_indices = list(set(range(len(self.buffer))) - exclude\_indices)

            if len(available\_indices) < remaining:

                sampled\_random\_experiences = random.choices(self.buffer, k=remaining)

            else:

                sampled\_random = random.sample(available\_indices, remaining)

                sampled\_random\_experiences = [self.buffer[idx] for idx in sampled\_random]

        else:

            sampled\_random\_experiences = []

        batch = sampled\_success\_experiences + sampled\_latest\_experiences + sampled\_random\_experiences

        if len(batch) < batch\_size:

            additional = batch\_size - len(batch)

            additional\_samples = random.choices(self.buffer, k=additional)

            batch += additional\_samples

        # 最后随机打乱顺序

        batch = random.sample(self.buffer, batch\_size)

        random.shuffle(batch)

        states, actions, rewards, next\_states, dones, info = zip(\*batch)

        return (np.array(states), np.array(actions), np.array(rewards, dtype=np.float32),

                np.array(next\_states), np.array(dones, dtype=np.float32))

    def size(self):

        return len(self.buffer)

    def plot\_train(self, hyperparameters):

        # plot the action and plot the reward

        fig = plt.figure(figsize=(20, 16))

        gs = gridspec.GridSpec(2, 2, height\_ratios=[5, 3], width\_ratios=[5, 5])

        ax1 = fig.add\_subplot(gs[0, 0])

        ax2 = fig.add\_subplot(gs[0, 1])

        ax3 = fig.add\_subplot(gs[1, :])

        action\_bound = [self.env.action\_space.low, self.env.action\_space.high]

        ax1.set\_title("x, y Distribution")

        ax1.set\_xlim(action\_bound[0][0],action\_bound[1][0])

        ax1.set\_ylim(action\_bound[0][1],action\_bound[1][1])

        ax1.set\_xlabel("x")

        ax1.set\_ylabel("y")

        ax2.set\_title("v, a Distribution")

        ax2.set\_xlim(action\_bound[0][2],action\_bound[1][2])

        ax2.set\_ylim(action\_bound[0][3],action\_bound[1][3])

        ax2.set\_xlabel("v")

        ax2.set\_ylabel("a")

        data\_x, data\_y, colors\_xy = [], [], []

        data\_v, data\_a, colors\_va = [], [], []

        reward\_list = []

        for state, action, reward, next\_state, done, info in self.buffer:

            if self.env.polar\_space:

                l, theta, v, a = action

                x = l \* np.cos(np.radians(theta))

                y = l \* np.sin(np.radians(theta))

            else:

                x, y, v, a = action

                (x,y) =  find\_nearest\_grid\_point(self.env.xy\_grid\_points, (x,y))

                (v,a) =  find\_nearest\_grid\_point(self.env.vi\_grid\_points, (v,a))

            # 根据反应结果设置颜色

            if info["reaction"] == "mol 0→1":

                # 半透明橙色

                color = (1, 0.5, 0, 0.5)

            elif info["reaction"] == "mol 1→2":

                # 半透明黄色

                color = (1, 1, 0, 0.5)

            elif info["reaction"] == "mol 1→3":

                # 半透明绿色

                color = (0, 1, 0, 0.5)

            elif info["reaction"] == "mol 2→4":

                # 半透明青色

                color = (0, 1, 1, 0.5)

            elif info["reaction"] == "mol 3→4":

                # 半透明蓝色

                color = (0, 0.5, 1, 0.5)

            elif info["reaction"] == "mol 4→5":

                # 半透明紫色

                color = (0.5, 0, 1, 0.5)

            elif info["reaction"] in ["failure", "bad", "wrong", "skip"]:  # 失败:

                color = (1, 0, 0, 0.5)  # 半透明红色

            elif info["reaction"] == "non":  # 无变化

                color = (0.5, 0.5, 0.5, 0.5)  # 半透明灰色

            data\_x.append(x)

            data\_y.append(y)

            colors\_xy.append(color)

            data\_v.append(v)

            data\_a.append(a)

            colors\_va.append(color)

            reward\_list.append(reward)

        ax1.scatter(data\_x, data\_y, c=colors\_xy, alpha=0.5)

        ax2.scatter(data\_v, data\_a, c=colors\_va, alpha=0.5)

        ax3.plot(reward\_list, color='blue')

        plt.tight\_layout()

        if not os.path.exists("./train\_result\_plot"):

            os.makedirs("./train\_result\_plot")

        # add the hyperparameters to the figure name

        str\_hyperparameters = '\_'.join([f"{key}={value}" for key, value in hyperparameters.items()])

        plt.savefig(f"train\_result\_plot/action\_{self.now}\_{str\_hyperparameters}.png")

    # save the buffer data as a txt file

    def save(self):

        # now = time.strftime('%Y-%m-%d\_%H-%M-%S', time.localtime(time.time()))

        with open(f"train\_result\_plot/buffer\_data\_{self.now}.txt", "w") as f:

            for state, action, reward, next\_state, done, info in self.buffer:

                f.write(f"state: {state}, action: {action}, reward: {reward}, next\_state: {next\_state}, done: {done}, info: {info}\n")

        print("Save the buffer data successfully!")

    def read(self, data\_file):

        """

        读取之前保存的 buffer\_data 文件，并将数据还原到 self.buffer 中。

        state, action, reward, next\_state, done, info

        对应 np.array, np.array, float, np.array, bool, dict。

        """

        if not os.path.exists(data\_file):

            print(f"No data file found at {data\_file}.")

            return

        self.buffer.clear()

        with open(data\_file, "r") as f:

            for line in f:

                line = line.strip()

                if not line:

                    continue

                # 根据写入时的格式进行切分

                parts = line.split(", ")

                # 依次解析 state、action、reward、next\_state、done、info

                st\_str = parts[0].replace("state: ", "")

                ac\_str = parts[1].replace("action: ", "")

                re\_str = parts[2].replace("reward: ", "")

                ns\_str = parts[3].replace("next\_state: ", "")

                do\_str = parts[4].replace("done: ", "")

                in\_str = ""

                for parts in parts[5:]:

                    in\_str = in\_str + ',' + parts

                in\_str = in\_str[1:].replace("info: ", "")

                # 解析 state

                st\_nums = st\_str.strip("[]").split()

                state\_arr = np.array(list(map(float, st\_nums))) if st\_nums else np.array([])

                # 解析 action

                ac\_nums = ac\_str.strip("[]").split()

                action\_arr = np.array(list(map(float, ac\_nums))) if ac\_nums else np.array([])

                # 解析 reward

                reward\_val = float(re\_str)

                # 解析 next\_state

                ns\_nums = ns\_str.strip("[]").split()

                next\_state\_arr = np.array(list(map(float, ns\_nums))) if ns\_nums else np.array([])

                # 解析 done

                done\_val = (do\_str == "True")

                # 解析 info 为字典

                info\_dict = ast.literal\_eval(in\_str)

                # 将还原的数据加入缓冲区

                self.buffer.append(

                    (state\_arr, action\_arr, reward\_val, next\_state\_arr, done\_val, info\_dict)

                )

        print("Read the buffer data successfully!")

# SAC Agent

class SACAgent:

    def \_\_init\_\_(self, env: Env

                 , buffer\_size=10000,

                 batch\_size=64,

                 gamma=0.99,

                 tau=0.005,

                 alpha=0.3,

                 lr=0.0003):

        self.hyperparameters = {

            "buffer\_size": buffer\_size,

            "batch\_size": batch\_size,

            "gamma": gamma,

            "tau": tau,

            "alpha": alpha,

            "lr": lr

        }

        self.env = env

        # TODO: redefine the observation space in the env\_Br.py

        self.state\_dim = 5 #env.observation\_space.shape[0]                         # input dimension

        self.action\_dim = env.action\_space.shape[0]                             # output dimension

        self.action\_bound = [env.action\_space.low, env.action\_space.high]

        self.device = torch.device("cuda:0" if torch.cuda.is\_available() else "cpu")

        self.target\_entropy = -torch.tensor([self.action\_dim], dtype=torch.float32, device=self.device)

        self.replay\_buffer = ReplayBuffer(buffer\_size,env)

        self.batch\_size = batch\_size

        self.gamma = gamma

        self.tau = tau

        self.alpha = alpha

        self.epsilon = 1e-6

        low, high = self.action\_bound

        self.action\_scale = torch.tensor((high - low) / 2, dtype=torch.float32, device=self.device)

        self.action\_bias = torch.tensor((high + low) / 2, dtype=torch.float32, device=self.device)

        # self.action\_scale = (high - low) / 2

        # self.action\_bias = (high + low) / 2

        # Networks

        # self.policy\_net = MLP(self.state\_dim, self.action\_dim \* 2).to(self.device)

        self.policy\_net = PolicyNet(self.state\_dim, self.action\_dim).to(self.device)

        self.q\_net1 = CriticNet(self.state\_dim + self.action\_dim, 1).to(self.device)

        self.q\_net2 = CriticNet(self.state\_dim + self.action\_dim, 1).to(self.device)

        self.target\_q\_net1 = CriticNet(self.state\_dim + self.action\_dim, 1).to(self.device)

        self.target\_q\_net2 = CriticNet(self.state\_dim + self.action\_dim, 1).to(self.device)

        self.log\_alpha = torch.tensor([np.log(alpha)], requires\_grad=True, device=self.device, dtype=torch.float32)

        self.alpha = self.log\_alpha.detach().exp()

        self.alpha\_optim = optim.Adam([self.log\_alpha], lr=lr)

        # Optimizers

        self.policy\_optimizer = optim.Adam(self.policy\_net.parameters(), lr=lr, weight\_decay=1e-4)

        self.q\_optimizer1 = optim.Adam(self.q\_net1.parameters(), lr=lr, weight\_decay=1e-4)

        self.q\_optimizer2 = optim.Adam(self.q\_net2.parameters(), lr=lr, weight\_decay=1e-4)

        # Copy weights to target networks

        self.target\_q\_net1.load\_state\_dict(self.q\_net1.state\_dict())

        self.target\_q\_net2.load\_state\_dict(self.q\_net2.state\_dict())

    # TODO: redefine the one\_hot\_encode function

    def one\_hot\_encode(self, state):

        """根据当前分子状态和反应时间生成独热编码"""

        molecule\_state = int(state) + 1  # 分子状态

        # reaction\_time = state[1]  # 反应时间

        one\_hot = np.zeros(self.state\_dim)

        one\_hot[0] = molecule\_state  # 对分子状态进行独热编码

        return np.array(one\_hot)

    def remap\_action(self, action):

        """将 [-1, 1] 范围的动作重新映射到实际动作空间"""

        return self.action\_bias + self.action\_scale \* action

    def select\_action(self, states, deterministic=False, detach\_action=True):

        if isinstance(states, np.ndarray):

            states = torch.from\_numpy(states).float().to(self.device)

        elif isinstance(states, torch.Tensor):

            states = states.to(self.device)

        mean, log\_std = torch.chunk(self.policy\_net(states), 2, dim=-1)

        log\_std = torch.clamp(log\_std, -20, 2)

        std = log\_std.exp()

        dist = Normal(mean, std)

        if deterministic:

            pre\_tanh\_action = mean

        else:

            pre\_tanh\_action = dist.rsample()

        action = torch.tanh(pre\_tanh\_action)  # 将动作限制在 [-1, 1]

        log\_prob = dist.log\_prob(pre\_tanh\_action) - torch.log(self.action\_scale \* (1 - action.pow(2)) + self.epsilon)

        log\_prob = log\_prob.sum(dim=-1, keepdim=True)

        remapped\_action = self.action\_bias + self.action\_scale \* action

        if detach\_action:

            return remapped\_action.detach().cpu().numpy(), log\_prob.detach()

        return remapped\_action, log\_prob

    def train(self, num\_steps, render=False, visualize=False, load\_buffer=True):

        state = self.env.reset()

        # state = self.one\_hot\_encode(state)  # 转换状态为独热编码并添加反应时间

        # plot the action

        if visualize:

            queue = multiprocessing.Queue(50)

            plot\_process = multiprocessing.Process(target=self.plot\_actions, args=(queue,self.env))

            plot\_process.start()

        if load\_buffer:

            data\_file = './train\_result\_plot/pre\_buffer/Br\_buffer\_data.txt'

            self.replay\_buffer.read(data\_file)

        for step in range(num\_steps):

            # create a random 0 or 1 to decide the reaction path

            # self.reaction\_path\_num = random.randint(0,1)

            action,\_ = self.select\_action(state, detach\_action=True)

            # connect to the env

            next\_state, reward, done, info = self.env.step(action)

            print(f"Reward: {reward}", f"Info: {info}")

            # next\_state = self.one\_hot\_encode(next\_state)  # 转换为独热编码并添加反应时间

            # exp augmentation

            self.replay\_buffer.add(state, action, reward, next\_state, done, info)

            if render:

                self.env.render()

                self.env.clock.tick(60)

            if visualize:

                if queue.full():  # 如果队列满了，则等待一段时间

                    queue.get()

                queue.put((state.copy(), action.copy(), reward, next\_state.copy(), done, info.copy()))

            if self.replay\_buffer.size() > self.batch\_size:

                self.update()

            if step % 1000 == 0:

                self.save\_model(step)

            if done:

                state = self.env.reset()  # 重置并编码

            else:

                state = next\_state

        if visualize:

            queue.put(None)

            plot\_process.join()

        self.replay\_buffer.plot\_train(self.hyperparameters)

        self.replay\_buffer.save()

    def update(self):

        states, actions, rewards, next\_states, dones = self.replay\_buffer.sample(self.batch\_size)

        states = torch.tensor(states, dtype=torch.float32, device=self.device)

        actions = torch.tensor(actions, dtype=torch.float32, device=self.device)

        #reverse the action to the range of [-1,1]

        actions = (actions - self.action\_bias) / self.action\_scale

        rewards = torch.tensor(rewards, dtype=torch.float32, device=self.device).unsqueeze(1)

        next\_states = torch.tensor(next\_states, dtype=torch.float32, device=self.device)

        dones = torch.tensor(dones, dtype=torch.float32, device=self.device).unsqueeze(1)

        # Q-Loss

        with torch.no\_grad():

            next\_actions, next\_state\_log\_pi = self.select\_action(next\_states, detach\_action=True)

            next\_actions = torch.tensor(next\_actions, dtype=torch.float32, device=self.device)

            next\_actions = (next\_actions - self.action\_bias) / self.action\_scale

            q\_target1 = self.target\_q\_net1(torch.cat([next\_states, next\_actions], dim=-1))

            q\_target2 = self.target\_q\_net2(torch.cat([next\_states, next\_actions], dim=-1))

            q\_min = torch.min(q\_target1, q\_target2) - self.alpha \* next\_state\_log\_pi

            q\_target = rewards + (1 - dones) \* self.gamma \* q\_min

        q1 = self.q\_net1(torch.cat([states, actions], dim=-1))

        q2 = self.q\_net2(torch.cat([states, actions], dim=-1))

        q\_loss1 = ((q1 - q\_target) \*\* 2).mean()

        q\_loss2 = ((q2 - q\_target) \*\* 2).mean()

        q\_loss = q\_loss1 + q\_loss2

        # Policy-loss

        new\_actions, log\_prob = self.select\_action(states, detach\_action=False)  # 保持与计算图的连接

        new\_actions = (new\_actions - self.action\_bias) / self.action\_scale

        # new\_actions 已经是张量，与计算图相连

        q1\_new = self.q\_net1(torch.cat([states, new\_actions], dim=-1))

        q2\_new = self.q\_net2(torch.cat([states, new\_actions], dim=-1))

        min\_q\_pi = torch.min(q1\_new, q2\_new)

        policy\_loss = (self.alpha \* log\_prob - min\_q\_pi).mean()

        # policy\_loss = (self.alpha\* log\_prob - states  ).mean()

        # Alpha-loss

        alpha\_loss = -(self.log\_alpha \* (log\_prob + self.target\_entropy).detach()).mean()

        # Update critic

        self.q\_optimizer1.zero\_grad()

        self.q\_optimizer2.zero\_grad()

        q\_loss.backward()

        # torch.nn.utils.clip\_grad\_norm\_(self.q\_net1.parameters(), 2, norm\_type=2.0)    # gradient clipping

        # torch.nn.utils.clip\_grad\_norm\_(self.q\_net2.parameters(), 2, norm\_type=2.0)

        # Update policy

        self.policy\_optimizer.zero\_grad()

        policy\_loss.backward()

        torch.nn.utils.clip\_grad\_norm\_(self.policy\_net.parameters(), 2, norm\_type=2.0)      # gradient clipping

        # Update alpha

        self.alpha\_optim.zero\_grad()

        alpha\_loss.backward()

        # torch.nn.utils.clip\_grad\_norm\_(self.log\_alpha.parameters(), 2, norm\_type=2.0)      # gradient clipping

        # Update all networks

        self.q\_optimizer1.step()

        self.q\_optimizer2.step()

        self.policy\_optimizer.step()

        self.alpha\_optim.step()

        self.alpha = self.log\_alpha.detach().exp()

        # Soft update target networks

        for target\_param, param in zip(self.target\_q\_net1.parameters(), self.q\_net1.parameters()):

            target\_param.data.copy\_(self.tau \* param.data + (1 - self.tau) \* target\_param.data)

        for target\_param, param in zip(self.target\_q\_net2.parameters(), self.q\_net2.parameters()):

            target\_param.data.copy\_(self.tau \* param.data + (1 - self.tau) \* target\_param.data)

    def save\_model(self, step):

        # get the current date and time

        now = time.strftime('%Y-%m-%d\_%H-%M-%S', time.localtime(time.time()))

        if not os.path.exists(f"models/{now}"):

            os.makedirs(f"models/{now}")

        torch.save(self.policy\_net.state\_dict(), f"models/{now}/policy\_net\_{step}.pth")

        torch.save(self.q\_net1.state\_dict(), f"models/{now}/q\_net1\_{step}.pth")

        torch.save(self.q\_net2.state\_dict(), f"models/{now}/q\_net2\_{step}.pth")

需要注意的是

当初始状态为(0,0,0,0,path)的经验可以由1条拓展为8条

(1,0,0,0,path) 由1条拓展为2条

(1,1,0,0,path) 由1条拓展为2条

(1,1,1,0,path) 由1条拓展为2条

我大致总结出了如下改动

1.我将我的SAC的代码发送给你，我希望在class ReplayBuffer中添加一个aug\_exp的方法，使得我输入一个exp，他可以根据输入的exp来所有数据增强，并且将所有数据存储到buffer中。数据增强中的reward计算可以调用class ReplayBuffer中的self.env.reward\_culculate()来计算。

这个方法将替代SAC训练中

self.replay\_buffer.add(state, action, reward, next\_state, done, info)代码

转为

self.replay\_buffer. aug\_exp (state, action, reward, next\_state, done, info)

2. 在如下代码之前，将state转变为合法状态，并记录对环境操作

action,\_ = self.select\_action(state, detach\_action=True)

然后在如下代码之前，将action中的x y分量逆转化环境操作

next\_state, reward, done, info = self.env.step(action)