姓　　名：高婉婷

院　　系：市场学系电子商务

年　　级：2018

学　　号：2018053132

课程名称：电子商务应用开发

指导老师：汤胤

项目名称	**RL Market timing**
分工	团队合作项目，高婉婷为项目负责人
日期	2021 年 12 月 12 日
分数	

Contents

一、项目简介.....3
强化学习主要模块：.....3
程序的结构图如下：.....3
Github.....3
代码结构：.....3
二、get data.....7
tushare → get_data 保存的数据表 → raw_data.sql.....7
API 接口说明：.....7
三、preprocessor.....9
raw_data → Preprocessor 部分计算指标 → processed.....9
MDPP 算法.....9
四、env.....16
五、agent.....19
六、action.....23
设计与技术路线.....23
七、reward.....23
设计与技术路线.....23
version 迭代.....23
八、evaluation.....26
设计思路：.....26
所以一个 `trained` 可以有如下内容：.....26
九、state.....28
Version 1.....28
state 部分字段.....28
OCLHV 处理成环比的作用.....28
十、VAE.....29
实现方式.....29
关键代码.....29
十一、predictor.....31
watchlist.....31
predictor.....31
模型的保存和加载.....31
十二、trainer.....33
设计思路：.....33
关键代码：.....33
十三、main.....35
十四、streamlit 前端.....36
用 streamlit 做展示；.....36
十五、项目总结.....40
1.agent 替换算法.....40
2.get data.....40
3.reward.....40

一、项目简介

该项目是一个基于强化学习（Reinforcement Learning）的股市预测问题，程序的最终目标是给出单支股票的择时策略。

强化学习主要模块：

Agent：做决策的小机器人，会依据它目前的经验和当前的环境给出一个 action

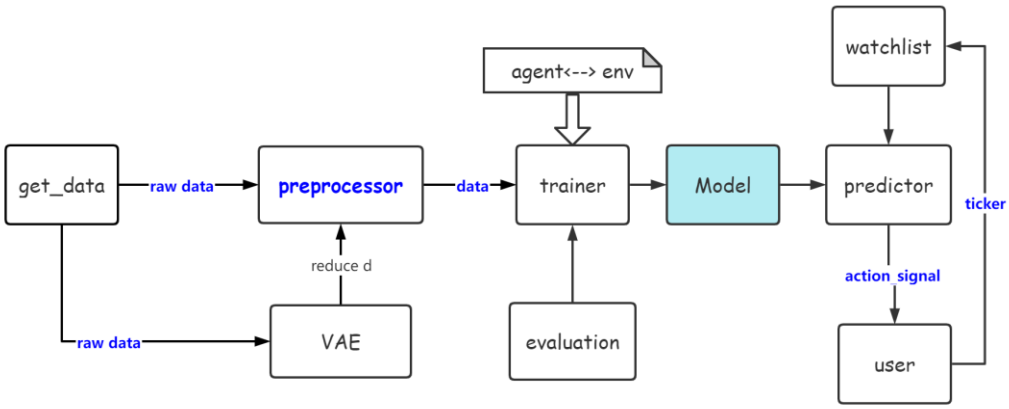
action：对于股市来说，可能是[buy, hold(neutral),sell]

Environment：股市的环境

reward：奖励是由环境给的一个标量的反馈信号(scalar feedback signal)，这个信号显示了 agent 在某一步采取了某个策略的表现如何。

state：当前时间点环境的状态

程序的结构图如下：

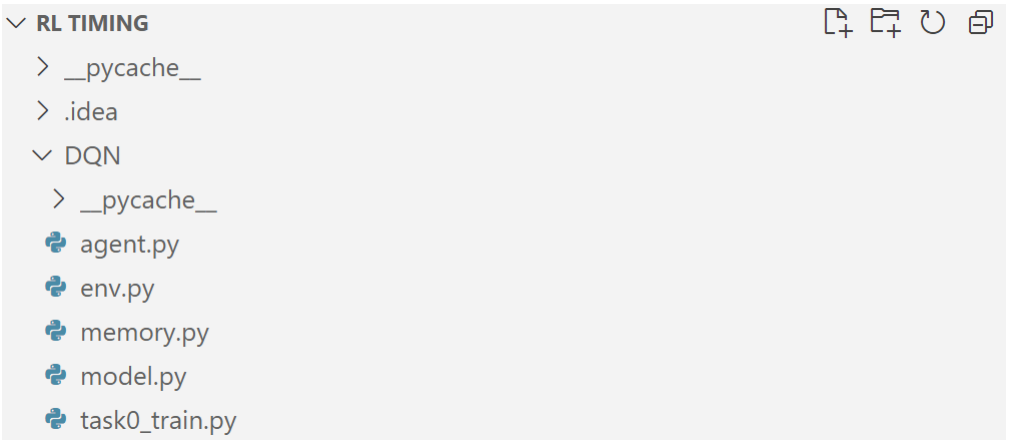


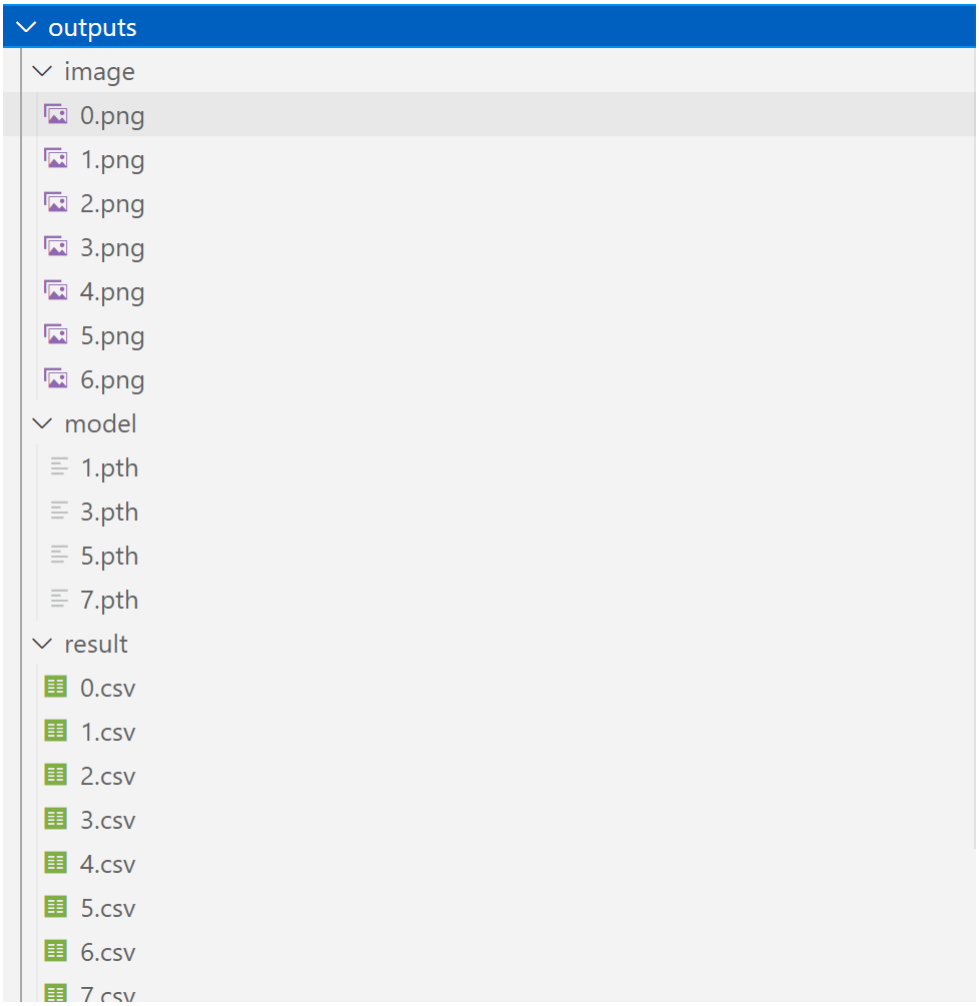
Github

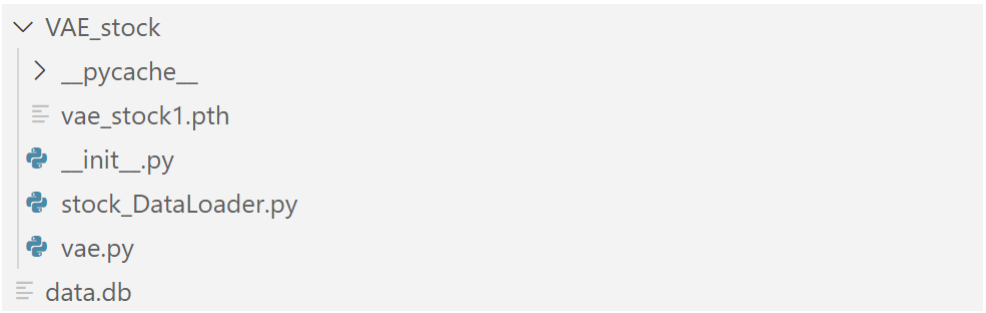
项目已同步至 github，链接如下：

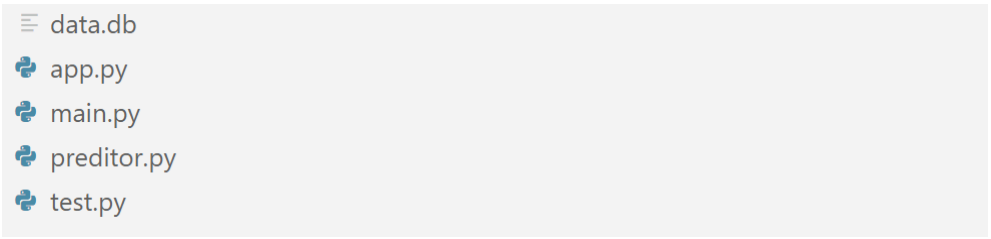
https://github.com/gaowanting/RL_stock_forecast2

代码结构：


RL TIMING
__pycache__
.idea
DQN
__pycache__
agent.py
env.py
memory.py
model.py
task0_train.py


outputs
image
0.png
1.png
2.png
3.png
4.png
5.png
6.png
model
1.pth
3.pth
5.pth
7.pth
result
0.csv
1.csv
2.csv
3.csv
4.csv
5.csv
6.csv
7.csv





RL TIMING

- Data.db 为存储数据的数据库，使用 sqlite
- app.py 为前端展示页面，使用 streamlit
- main.py 为代码总成，启动整个程序
- preditor.py 独立于以上，用于使用训练好的模型预测 action
- test.py 为项目使用过的所有测试代码

DQN

- agent.py 为 DQN 算法
- env.py 搭建整个股市预测环境
- memory.py 为 DQN 算法中使用的的 Replay Buffer
- model.py 为 MLP
- task0_train.py 实现训练过程

outputs 中保存所有训练的结果

- image 文件夹下保存每个 episode 的 action signal 图
- model 保存训练好的模型
- result 保存训练中产生的数据，如 total asset，action signal，reward 等信息

util

- evaluate.py 为评估板块
- get_data.py 获取股市数据

- preprocessor.py 为数据预处理模块
- MDPP plus.py 为改进版的标注拐点做平滑的算法
- plot.py 绘制 reward 曲线
- visualize.py 主要实现标注 action signal 可视化的过程

VAE 为实现数据降维模块

- Vae_stock1.pth 为预训练出的模型
- Stock_DataLoader.py 为预训练使用的数据加载类
- Vae.py 为 VAE 实现的算法代码

二、get data

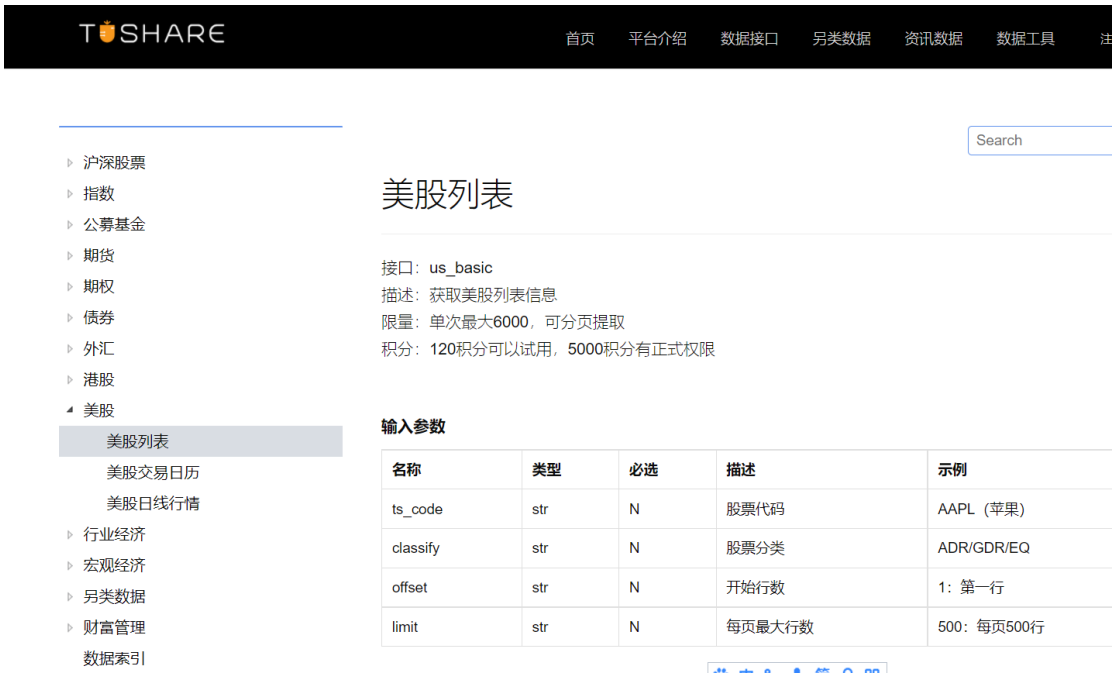
tushare → get_data 保存的数据表 → raw_data.sql

by using tushare or baostock, we retrieve including but not limited to the following fields

day	以日为单位
ticker	股票代码
open	开盘价
close	收盘价
low	最低价格
high	最高价格
volume	成交量

API 接口说明：

使用 tushare.pro 接口
Tushare 大数据社区




```
def get_random_data():
    tushare_tocken =
"c576df5b626df4f37c30bae84520d70c7945a394d7ee274ef2685444"
    ts.set_token(tushare_tocken)
```

```
    pro = ts.pro_api()

    while True:
        # 在上市公司中随机抽取股票一支股票代码
        data = pro.stock_basic(exchange='', list_status='L',
fields='ts_code, name, area,industry,list_date')
        tic = random.choice(data["ts_code"])


        # 从 2008 年到 2020 年随机选取 10 年的数据
        start_time = "2008-01-01"
        end_time = "2010-01-01"
        timestamp_s = time.mktime(time.strptime(start_time, "%Y-%m-%d"))
        timestamp_e = time.mktime(time.strptime(end_time, "%Y-%m-%d"))
        chose_time = random.randint(timestamp_s, timestamp_e)

        selected_start_time = time.strftime("%Y-%m-%d",
time.localtime(chose_time))
        selected_end_time = time.strftime("%Y-%m-%d",
time.localtime(chose_time + 315532800))

        data_tmp = ts.pro_bar(ts_code=tic, adj='qfq', freq="d",
                              start_date=selected_start_time,
end_date=selected_end_time)
        if (data_tmp is not None) and (len(data_tmp)>2000):
            print(f"the selected stock for training model is: {tic}\nHere
is the information")
            print(data[data["ts_code"] == tic])
            break
    return data_tmp
```

三、preprocessor

raw_data → Preprocessor 部分计算指标 → processed

After having read from `raw_data`, `Preprocessor` calculates several indicators (and embeddings as well if it is 5-minute ones) on pandas.DataFrame and then save in `processed`

day	**以日为单位。注如果获得的是分钟线，需要另外用算法投影成 embedding。**
ticker	股票代码
open	相比于上一日的 close 的涨跌%，例如 25，即升了 25%
close	相比于上一日的 close 的涨跌%，例如 25，即升了 25%
low	相比于上一日的 close 的涨跌%，例如 25，即升了 25%
high	相比于上一日的 close 的涨跌%，例如 25，即升了 25%
volume	相比于上一日的 volume 的涨跌%，例如 25，即升了 25%
k	来自 ta-lib 包，当然也可以增加更多其他指标
d	
j	
r	
s	
i	
...	more indicators from ta-lib
embedding	20 日以来的经过环比处理过后的 open、close、low、high、volume

注：`processed` 经过 predictor.py 预测后还将输出若干字段写入 `predicted`，如买卖信号 `signal`，根据买卖信号再计算 return, earning 等字段, see also [evaluation.py]

MDPP 算法

在预处理中使用了源于论文 Landmarks: A New Model for Similarity-Based Pattern Querying in Time Series Databases 中抽象出的 MDPP 算法

landmark

- n-th order landmark of a curve if the n-th order derivative is 0 on the point
- 1-th landmark: local max & local min
  2-th landmark: inflection points
  …..

restrict discussion to only “first-order landmarks”

MDPP

Minimal Distance/Percentage Principle

Given:

(1)a sequence of landmarks(x1,y1),…（xn, yn）

(2)a minimal distance D

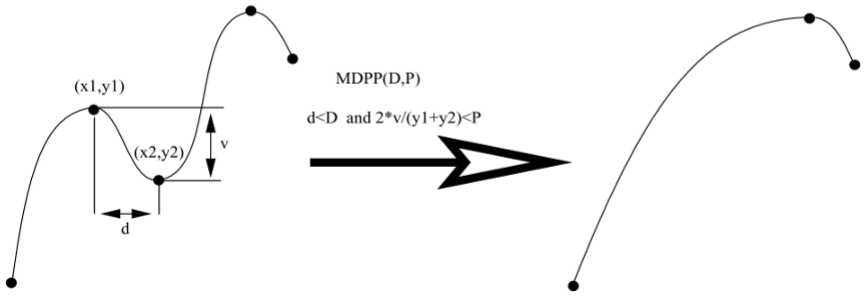
(3)a minimal percentage P

Then:

remove landmark(xi, yi) and (xi+1,yi+1)

If:

$$x_{i+1} - x_i < D \text{ and } \frac{|(y_{i+1} - y_i)|}{(|y_i| + |y_{i+1}|)/2} < P.$$

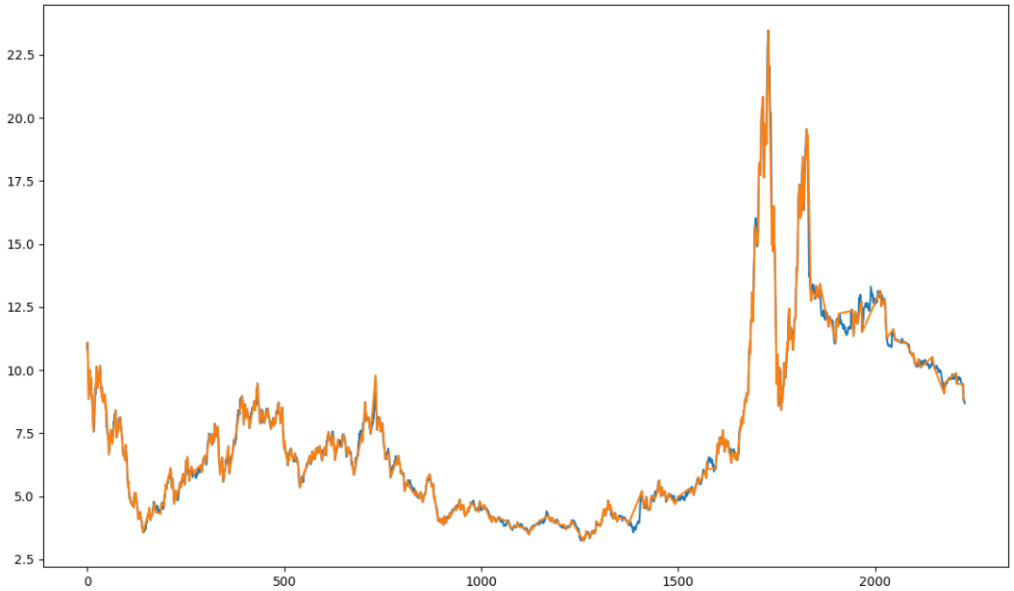


MDPP 实现代码如下：

```
def MDPP(D,P,df):
    df['landmark'] = '-'
    df.loc[(df.close.diff(1) > 0) & (df.close.diff(-1) > 0) &
(df.landmark != 1), 'landmark'] =  '^'
    df.loc[(df.close.diff(1) < 0) & (df.close.diff(-1) < 0) &
(df.landmark != 1), 'landmark'] =  'v'
    d = df[df['landmark'].isin(['^', 'v'])]
    for i in range(0,len(d)-2,2):
        P_ = abs(d.loc[d.index[i+1],'close'] - d.loc[d.index[i],'close'])
/((d.loc[d.index[i+1],'close'] + d.loc[d.index[i],'close'])/2)
        if (d.index[i+1] - d.index[i] < D) and (P_<P):
            df.loc[d.index[i+1],'landmark'] = '-'
            df.loc[d.index[i],'landmark'] = '-'
```

```
    return df
```

实现效果图：



注：源数据为 2284 天，使用 MDPP 算法后，仅用 467 个拐点，拟合出近似原曲线的效果，找出 ground truth

```
class Preprocessor:

    def __init__(self,df) -> None:
        self.df = df
        self.tech_indicator_list = ['kdjk', 'kdjd', 'kdjj', "rsi_6",
"rsi_12", "rsi_24"]
        self.normalization = 'standardization' # 'div_close'/
'standardization'
        self.windowsize = 20

    def data_cleaning(self):
        data_df = self.df

        data_df = data_df.set_index("trade_date", drop=True)  # 将

trade_date 列设为索引

        data_df = data_df.reset_index()


        # 删除一些列并更改列名
```

```
        data_df = data_df.drop(["pre_close", "change", "pct_chg",
"amount"], axis=1)
        data_df.columns = ["date", "tic", "open", "high", "low", "close",
"volume"]

        # 更改 date 列数据格式
        data_df[["date"]] = data_df[["date"]].astype(str)
        data_df["date"] = data_df.date.apply(lambda x:
datetime.strptime(x[:4] + '-' + x[4:6] + '-' + x[6:], "%Y-%m-%d"))
        data_df["date"] = data_df.date.apply(lambda x:
x.strftime("%Y-%m-%d"))

        # 删除为空的数据行
        data_df = data_df.dropna()
        data_df = data_df.sort_values(by=['date',
'tic']).reset_index(drop=True)

        # 添加 day 列,依次递增
        data_df['day'] = range(len(data_df))
        return data_df


    def add_technical_indicator(self,data_df):
        """对数据添加技术指标"""
        df = data_df
        df = df.sort_values(by=['tic', 'date'])
        # 获取 Sdf 的对象
        stock = Sdf.retype(df.copy())
        unique_ticker = stock.tic.unique()  # stock.tic 是一个
pandas.Series

        # 添加技术指标
        for indicator in self.tech_indicator_list:
            indicator_df = pd.DataFrame()
            for ticker in unique_ticker:
                tmp_df = pd.DataFrame(stock[stock.tic ==
ticker][indicator])
                tmp_df['tic'] = ticker
```

```
                tmp_df['date'] = df[df.tic == ticker]['date'].to_list()
                indicator_df = indicator_df.append(tmp_df, ignore_index =
True)
            df = df.merge(indicator_df[['tic', 'date', indicator]],
on=['tic', 'date'], how='left')
        indicator_df = df.sort_values(by=['date', 'tic'])
        return indicator_df


    def feature_engineer(self,indicator_df):
        processed_df = indicator_df
        processed_df['amount'] = processed_df.volume * processed_df.close
        processed_df['change'] = (processed_df.close - processed_df.open)
/ processed_df.close
        processed_df['daily_variance'] = (processed_df.high -
processed_df.low) / processed_df.close
        processed_df = processed_df.fillna(0)


        # print("技术指标列表: ")

        # print(self.tech_indicator_list)

        # print("技术指标数: {}个".format(len(self.tech_indicator_list)))

        # print(processed_df.head())

        # print("DataFrame 的大小: ", processed_df.shape)

        return processed_df

    def do_normalization(self,processed_df):
        df = processed_df
        norm_list = ["open", "close", "high", "low", "volume"]
        after_norm = ["open_", "close_", "high_", "low_", "volume_"]
        formater = '{0:.04f}'.format
        if self.normalization == 'div_self':

            # 将 ochlv 处理为涨跌幅

            df[after_norm] =
df[norm_list].pct_change(-1).applymap(formater)
            df = df.dropna()
        elif self.normalization == 'div_close':

            # 将 ochl 处理为相较于前一天 close 的比例

            temp = df[["open", "close", "high", "low"]].values[1:] /
df[["close"]].values[:-1]
            df = df[1:]
```

```
            # volume 单独处理
            df[["volume_"]] = df[["volume"]].pct_change(-1)
            df[["open_", "close_", "high_", "low_"]] = temp
        elif self.normalization == 'standardization':
            # do standardization in a sliding window
            norm_df = df.copy()
            for i in range(self.windowsize,len(df) - self.windowsize):
                temp = df.loc[df.index[i]:df.index[i + self.windowsize -
1], norm_list]
                scaler = preprocessing.StandardScaler().fit(temp)
                norm_df.loc[norm_df.index[i]:norm_df.index[i +
self.windowsize - 1], after_norm] = scaler.transform(temp)
            norm_df = norm_df.dropna()
            df = norm_df.copy()
        df = df.round(4)
        return df

    def landmark(self,norm_df):
        d = norm_df
        mdpp_df = MDPP(10,0.03,d)
        return mdpp_df

    def embedding(self,mdpp_df):
        final_df = mdpp_df
        embedding = []
        model = VAE()
        if torch.cuda.is_available(): model.cuda()
        model.load_state_dict(torch.load(r'./VAE_STOCK/vae_stock1.pth')
)
        for i in range(len(final_df)-self.windowsize):
            temp = final_df.iloc[i:20+i,-6:-1].to_numpy()
            data = torch.tensor(temp,dtype=torch.float32)  #
torch.Size([20, 5])
            data = data.reshape(1,100)
            data = Variable(data)
            mu, logvar = model.encode(data)
            z = model.reparametrize(mu, logvar).data.numpy()
            z = ','.join(str(i) for i in z[0])
            embedding.append(z)
        final_df = final_df.iloc[0:len(final_df)-self.windowsize,:]
        final_df['embedding'] = embedding
        return final_df
```

```
def process(self):
    data_df = self.data_cleaning()
    indicator_df = self.add_technical_indicator(data_df)
    processed_df = self.feature_engineer(indicator_df)
    norm_df = self.do_normalization(processed_df)
    mdpp_df = self.landmark(norm_df)
    mdpp_df = mdpp_df.sort_values(['date', 'tic'], ignore_index=True)
    mdpp_df.index = mdpp_df.date.factorize()[0]
    final_df = self.embedding(mdpp_df)
    return final_df
```

四、env

Env 为整个股市预测框架的关键，主要是定义了 reward，state，action，以及训练更新时的 reset，update 等关键函数。

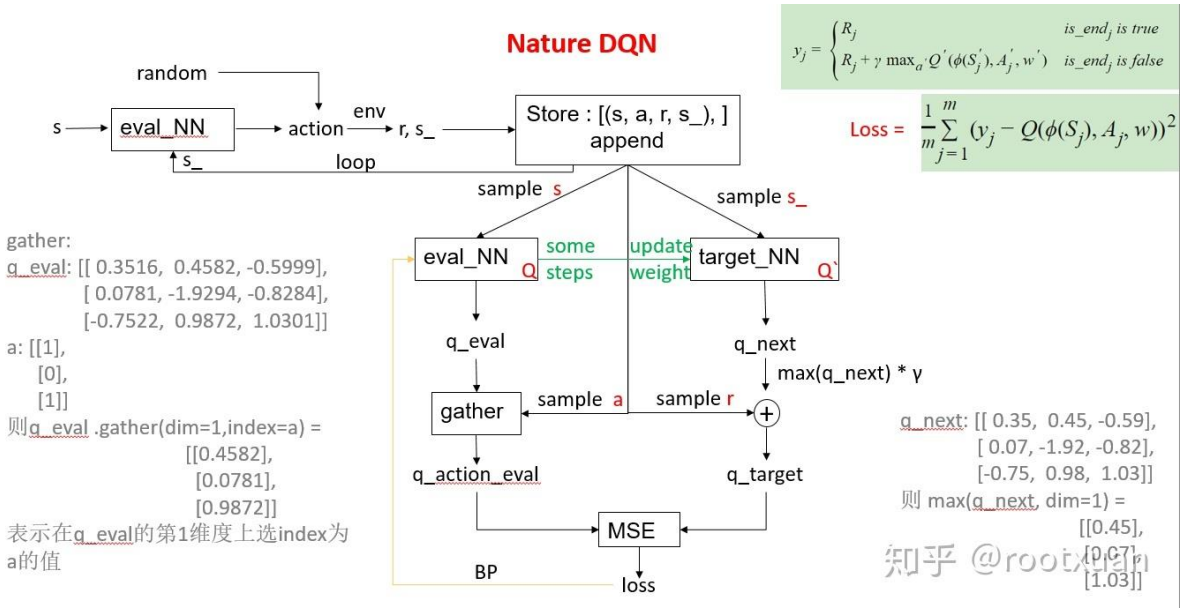
```
class StockLearningEnv(gym.Env):

    def render(self, mode="human"):
        pass


    def __init__(
            self,
            df: pd.DataFrame,
            buy_cost_pct: float = 3e-3,
            sell_cost_pct: float = 3e-3,
            print_verbosity: int = 10,
            initial_amount: int = 1e6,
            patient: bool = False,

            currency: str = "￥",

            is_train: bool = True,
    ) -> None:

        self.df = df
        self.dates = df['date']
        self.date_index = 0

        self.df = self.df.set_index('date')  # 把 date 设为 df 的索引

        self.assets = df['tic']
        self.patient = patient
        self.currency = currency
        self.is_train = is_train
        self.initial_amount = initial_amount
        self.print_verbosity = print_verbosity
        self.buy_cost_pct = buy_cost_pct
        self.sell_cost_pct = sell_cost_pct
        self.window_size = 20
        self.state_list = self.state
        self.state_space = len(self.state_list)*self.window_size
        self.action_space = spaces.Discrete(3)
        self.observation_space = spaces.Box(
            low=-np.inf, high=np.inf, shape=(self.state_space,)
        )
        self.seed()
```

```
        self.episode = -1
        self.episode_history = []
        self.printed_header = False
        self.max_total_assets = 0
        self.account_information = {
            "cash": [],
            "asset_value": [],
            "total_assets": [],
            "reward": []
        }
        self.rolling_window = True

    def reset(self) -> np.ndarray:
        self.seed()
        self.max_total_assets = self.initial_amount
        self.starting_point = 0
        self.date_index = 0
        self.episode += 1
        self.actions_memory = []
        self.transaction_memory = []
        self.state_memory = []
        self.coh_memory = [1e+6]
        self.holdings_memory = [0]
        self.account_information = {
            "cash": [],
            "asset_value": [],
            "total_assets": [],
            "reward": []
        }
        init_state = np.zeros(self.state_space)
        self.state_memory.append(init_state)
        return np.array([init_state])

    def step(
            self, actions: np.ndarray
    ) -> Tuple[list, float, bool, dict]:
        self.log_header()
        if (self.current_step + 1) % self.print_verbosity == 0:
            self.log_step(reason="update")
        # save evaluate information on last step for each episode
        if self.date_index == len(self.dates) - 1:
            if self.is_train:
                save_path =
f"train_record/train_action{self.episode}.csv"
```

```
                self.save_transaction_information().to_csv(save_path)
            return self.return_terminal(reward=self.reward)
        else:
            self.action = actions - 1
            transactions = self.action * 1000
            begin_cash = self.cash_on_hand

            assert_value = np.dot(self.holdings, self.closings)  # 目前总
持股金额
            reward = self.reward
            # save account_information
            self.account_information["cash"].append(begin_cash)
            self.account_information["asset_value"].append(assert_value
)
            self.account_information["total_assets"].append(begin_cash +
assert_value)
            self.account_information["reward"].append(reward)
            self.actions_memory.append(self.action)
            self.transaction_memory.append(transactions)

            sells = -np.clip(transactions, -np.inf, 0)
            proceeds = np.dot(sells, self.closings)
            costs = proceeds * self.sell_cost_pct

            coh = begin_cash + proceeds  # 计算现金的数量

            buys = np.clip(transactions, 0, np.inf)
            spend = np.dot(buys, self.closings)
            costs += spend * self.buy_cost_pct
            coh = coh - spend - costs
            holdings_updated = self.holdings + transactions
            self.date_index += 1
            state =
self.df.loc[self.df.index[self.date_index]:self.df.index[self.date_inde
x + self.window_size - 1],
                    self.state_list]
            state = state.values.reshape(1,240)
            self.state_memory.append(state)
            self.coh_memory.append(coh)
            self.holdings_memory.append(holdings_updated)
            return state, reward, False, {}
```

五、agent

version1 — RL 经典算法系列（DQN）



1、将环境信息s输入到eval网络， 输出为action的Q值，选择最大Q值对应的action， 或者随机生成一个action(所谓的探索)，将action输入环境， 得到奖励r和下一个环境信息s_， 存储当前环境s, 选择的action, 得到的奖励r这4个信息, 然后再将s_输入到eval网络， 循环步骤1， 直到存储了一定量的[(s, a, r, s_), ...]， 注意这个过程中eval网络没有更新。

2、从存储的[(s, a, r, s_), ...]中sample出一部分(s, a, r, s_) ，将sample出来的多个s做为batch输入到eval 网络， 得到batch的action的Q值q_eval， 然后根据sample出来的action选出其在q_eval中对应的Q值q_action_eval(对应gather过程)。

3、将sample出来的多个s_做为batch输入到target网络(eval与target有相同的网络结构)， 得到batch的action的Q值q_next, 取q_next中最大的Q值(dim=1)并乘以一个系数gamma， 然后再加上sample出来的奖励r， 得到q_target。

4、计算loss：loss 为第2步的结果q_action_eval与第3步的结果q_target的MSE， 然后根据loss反向传播更新eval网络。

5、循环1到4步骤， 每过一定的步数后，将eval网络的权重更新到target网络上。

注：在 Value-based 系列中，第三步是 target_net 唯一使用到的地方：用来计算(延迟的) q_{target}q_{target} = r + \gamma \argmax Q'(s')

```
class DQN:
    def __init__(self, state_dim, action_dim, **cfg):


        self.action_dim = action_dim  # 总的动作个数

        self.device = cfg["device"]  # 设备，cpu 或 gpu 等

        self.gamma = cfg["gamma"]  # 奖励的折扣因子

        # e-greedy 策略相关参数

        self.frame_idx = 0  # 用于 epsilon 的衰减计数

        self.epsilon = lambda frame_idx: cfg["epsilon_end"] + \
            (cfg["epsilon_start"] - cfg["epsilon_end"]) * \
            math.exp(-1. * frame_idx / cfg["epsilon_decay"])
        self.batch_size = cfg["batch_size"]
        self.policy_net = MLP(state_dim, action_dim,
hidden_dim=cfg["hidden_dim"]).to(self.device)
        self.target_net = MLP(state_dim, action_dim,
hidden_dim=cfg["hidden_dim"]).to(self.device)
        for target_param, param in
zip(self.target_net.parameters(),self.policy_net.parameters()): # copy
params from policy net
            target_param.data.copy_(param.data)
        self.optimizer = optim.Adam(self.policy_net.parameters(),
lr=cfg["lr"])
        self.memory = ReplayBuffer(cfg["memory_capacity"])

    def choose_action(self, state):

        '''选择动作
        '''
        self.frame_idx += 1
        if random.random() > self.epsilon(self.frame_idx):
            action = self.predict(state)
        else:
            action = random.randrange(self.action_dim)
        return action

    def predict(self, state):

        # 不求导

        with torch.no_grad():
```

```
            state = torch.tensor([state], device=self.device,
dtype=torch.float32)
            q_values = self.policy_net(state)
            action = q_values[0][0].tolist().index(q_values[0][0].max())
        return action


    def update(self):
        if len(self.memory) < self.batch_size:
            return

        # 从 memory 中随机采样 transition

        state_batch, action_batch, reward_batch, next_state_batch,
done_batch = self.memory.sample(
            self.batch_size)
        state_batch = [np.squeeze(i.T) for i in state_batch]

        '''转为张量

        例如 tensor([[-4.5543e-02,
-2.3910e-01,  1.8344e-02,  2.3158e-01],...,[-1.8615e-02, -2.3921e-01,
-1.1791e-02,  2.3400e-01]])'''
        state_batch = torch.tensor(
            state_batch, device=self.device, dtype=torch.float)
        action_batch = torch.tensor(action_batch,
device=self.device).unsqueeze(

            1)  # 例如 tensor([[1],...,[0]])

        reward_batch = torch.tensor(
            reward_batch, device=self.device, dtype=torch.float)  #
tensor([1., 1.,...,1])
        next_state_batch = torch.tensor(
            next_state_batch, device=self.device, dtype=torch.float)
        done_batch = torch.tensor(np.float32(
            done_batch), device=self.device)


        '''计算当前(s_t,a)对应的 Q(s_t, a)'''

        '''torch.gather:对于 a=torch.Tensor([[1,2],[3,4]]),那么
a.gather(1,torch.Tensor([[0],[1]]))=torch.Tensor([[1],[3]])'''
        q_values = self.policy_net(state_batch).gather(

            dim=1, index=action_batch)  # 等价于 self.forward
```

```
        # 计算所有 next states 的 V(s_{t+1})，即通过 target_net 中选取 reward
最大的对应 states
        next_q_values = self.target_net(next_state_batch)[0].max(
            1)[0].detach()  # 比如 tensor([ 0.0060, -0.0171,...,])
        # 计算 expected_q_value
        # 对于终止状态，此时 done_batch[0]=1，对应的 expected_q_value 等于
reward
        expected_q_values = reward_batch + \
            self.gamma * next_q_values * (1-done_batch)
        # self.loss =
F.smooth_l1_loss(q_values,expected_q_values.unsqueeze(1)) # 计算 Huber
loss
        loss = nn.MSELoss()(q_values, expected_q_values.unsqueeze(1))  #
计算 均方误差 loss

        # 优化模型
        self.optimizer.zero_grad()  # zero_grad 清除上一步所有旧的
gradients from the last step

        # loss.backward()使用 backpropagation 计算 loss 相对于所有
parameters(需要 gradients)的微分
        loss.backward()
        # for param in self.policy_net.parameters():  # clip 防止梯度爆炸
        #     param.grad.data.clamp_(-1, 1)
        self.optimizer.step()  # 更新模型
```

六、action

设计与技术路线

1. action = [buy, neutral, sell] 三个动作的概率，用 DQN/Reinforce 即可实现离散动作的选择

七、reward

设计与技术路线

reward 的设计是重中之重，有如下几种方案

- 直接用收益率，但这个是不准的，如果股价普遍上扬，agent 的认识是有 bias 的
- 识别未来若干日的波峰波谷（MDPP）后，计算 reward，可以客观地判断 agent 给出决策的好坏

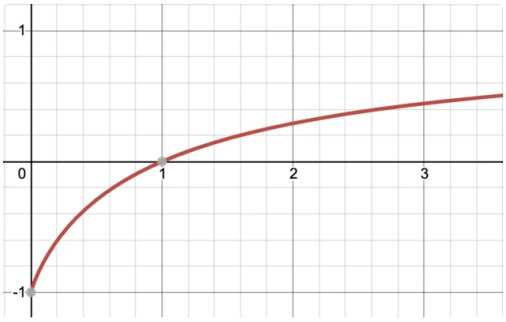
version 迭代

- 当前状态下总资产（cash_on_hand + hold_stock_num*price）
- 直接使用 evaluation 计算出的若干指标
- 考虑回撤率

注：经过以上方案尝试，agent 均未学习到有效的信息，因此最终 reward 设计方案如下：

- [ ] 利用 landmark 得到波峰波谷，从而客观地计算 action 的回报。Without loss of generality, 设 signal 附近（“附近”的定义在算法里）的 landmark 为 y_1 ， y_2 。那么三个点对 y 轴投影分别为 y_1, y_s, y_2 。理论上 s 应当离相同类型（同为买或同为卖）的 landmark 越近越好，离不同类型的 landmark 越远越好。尝试三种方案：

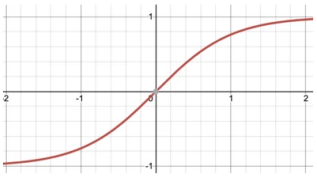
  - $r = tanh(log(\frac{|y_2 - y_s|}{|y_s - y_1| + \epsilon}))$ or $r = tanh(\frac{y_2 - y_s}{y_s - y_1 + \epsilon})$ where $\epsilon = 0.005$



注： $tanh(log(\cdot))$ 这个曲线对错误惩罚很重，对正确奖励比较少。作为家长，对 agent 是不是太严厉了点?

我认为是合理的，相比于赚钱，agent 先学会不亏钱更为重要。

  - $r = |y_2 - y_s| - |y_s - y_1|$
  - $r = tanh(|y_2 - y_s| - |y_s - y_1|)$



问题：这里价格为环比还是绝对价格？ 环比价格

```
    @property
    def reward(self) -> float :
        # init
        epsilon = 0.005
        signal_dict = {0:'-',-1:"v",1:"^"}
        action_signal = signal_dict.get(self.action)
        ground_truth = self.df.loc[self.df.index[self.date_index +
self.window_size],'landmark']
        current_day = self.df.day[self.date_index + self.window_size]
        landmark_series = self.df[self.df['landmark'].isin(["v","^"])]
        y_current = self.closings

        if action_signal == '-':
            reward = 0.5 if ground_truth == action_signal else -0.5
        else:
            # get y_previous
```

```
            y_previous = landmark_series[landmark_series.day <=
current_day].iloc[-1,5]
            # get y_same, y_diff
            previous_signal = landmark_series[landmark_series.day <=
current_day].iloc[-1,-6]
            y_next = landmark_series[landmark_series.day >=
current_day].iloc[0,5]
            y_next2 = landmark_series[landmark_series.day >=
current_day].iloc[1,5]
            y_same, y_diff = (y_previous,y_next) if previous_signal ==
action_signal else (y_next, y_next2)
            # calculate reward
            reward = np.tanh(np.log(abs(y_diff - y_current) / abs(y_same
- y_current + epsilon)))
        return  reward
```

八、evaluation

设计思路：

这个机器人的绩效评估，盈利能力，相比于“完全不操作”的比例，某个时间段的买卖信号和真正的信号对比的准确率，召回率等。

1. 机器人给出的信号，本质上是个分类问题，这个分类是不是与实际情况相符呢？可参考分类问题的若干指标，如**Accuracy, Precision, Recall 和 F1-score ，还可以绘制 ROC 曲线，以及 AUC 值（滑动计算）。**并判断指标的均值和方差（代表不稳定性）；

https://zhuanlan.zhihu.com/p/147663370

1. 机器人给出信号，盈利情况如何呢？从它买入到卖出，涨/跌了多少（比例）呢？如 80%，这个需要在卖出的节点位置记录下来；
2. 机器人给出信号，同时能否给出置信度（confidence），这代表了它对此次决策的信心。这个信心和前面的“不稳定性”对比，就可以判断它是否“过于自信”或“过于自卑”，并在强化学习框架下逐步调整到位
3. 假设一开头资产为 1，按照第 2 点全仓买入/卖出，资产增值情况会是怎样呢？这个也需要给出来，显示成曲线增长的形势
4. 买卖点，应当显示在 K 线图上，反而是 KDJ/RSI 这些隐藏在后台即可。

具体地说，需要在 pandas 中增加的字段有可能是：

- Flex (实际拐点，"本应"买入、卖出或不操作)，包括[1,0,-1]，这个需要手工打标签，或者编程寻找拐点
- action（机器人给出的买卖信号 ）
- confidence（机器人对此次决策的信心）
- earn_percentage (这一轮操作的盈利/亏损比例)

可能还有其他，代码上，可以有另外的进程，另外读取这个 pandas，显示在界面上。

所以一个 `trained` 可以有如下内容：

Evaluation Metrics	说明
day	以日为单位存储
ticker	股票代码
flex	MDPP 算法标记出的有意义的拐点 ref:LANDMARK/模型 MDPP 算法 这是计算分类 AUC 指标的依据——与信号距离 d<δd<\deltad<δ

Evaluation Metrics	说明
action	[1,0,-1]RL Agent 给出的决策
confidence	置信度 =1-σ2\sigma ^2σ2, 这里 σ2\sigma ^2σ2 是 RL 决策的方差
# accuracy	胜率 = 预测成功次数/预测总次数（由 Agent 估算）
# odds	赔率 = 成功时的盈利/失败时的亏损（由 Agent 估算）
AUC	滑动窗口末端的 AUC，用于评估 Agent 当前绩效
Accuracy	滑动窗口末端的 Accuracy，用于评估 Agent 当前绩效
F1-Score	滑动窗口末端的 F1-Score，用于评估 Agent 当前绩效
Precision	滑动窗口末端的 Precision，用于评估 Agent 当前绩效
Recall	滑动窗口末端的 Recall，用于评估 Agent 当前绩效
earning	依据 flex 计算距离上次买入/卖出的收益%
returns	收益率%，根据 earning 序列迭代计算。用于绘制一个单位资产（eg. 1 万）根据策略全仓买卖的收益曲线

```
class Evaluation:

    # 评估类，  auc()，绘制滑动 auc 曲线

    #         accuracy()，  返回准确率

    #         precision()，  返回三种多分类精准度

    #         recall()，  返回三种多分类召回率

    #         F1_score()，  返回三种多分类 F1 分数

    def __init__(self, cfg=None, env=None, agent=None, MDPP_D=5,
MDPP_P=0.03, single_true=True, windows_size=50):

        # 评估训练集动作集时可不输入配置，环境，代理这三个参数。

        self.cfg = cfg
        self.env = env
        self.agent = agent
        self.MDPP_D = MDPP_D
        self.MDPP_P = MDPP_P

        self.single_true = single_true  # auc c(1,n)默认单个元素为真值

        self.windows_size = windows_size  # auc 滑动窗口大小

        # self.accuracy_show = []
```

九、state

Version 1

- rsi,kdj,SAR,MACD,DMI,CCI,WR,BOLL,EMA,OBV 若干指标
- 实现 20 日滑动窗口
- + 股票的 oclhv（环比）（）处理成相较于前一天的涨跌幅
- + 对 20 日滑动窗口数据做 AutoEncoder 降维 vae.py

state 部分字段

open	相比于上一日的 open 的涨跌%，例如 25，即升了 25%
close	相比于上一日的 close 的涨跌%，例如 25，即升了 25%
low	相比于上一日的 low 的涨跌%，例如 25，即升了 25%
high	相比于上一日的 high 的涨跌%，例如 25，即升了 25%
volume	相比于上一日的 volume 的涨跌%，例如 25，即升了 25%
k	来自 ta-lib 包，当然也可以增加更多其他指标
d	
j	
r	
s	
i	
embedding hidden 用 AE 方法对上述指标进行降维得到的表示（v2）	

注：计算 open/close/low/high 的环比有两种路线，注意细微区别。两种都可以尝试一下，就修改几行代码而已。

- 相比 open/close/low/high 各自上一天的涨跌百分比
- 相比上一天 close 的涨跌百分比，编码时候略微麻烦一些，但它有它的道理。
- 对 open/close/low/high/volume 直接做 standardization 处理？

```
Standardize features by removing the mean and scaling to unit
variance
```

OCLHV 处理成环比的作用

对数据进行类似 normalization(归一化)的操作，由于 batch_normlization 和 layer_normlization 在 DRL 中较难处理，因此用环比代替，目前代码中使用 pct_change(-1)处理为相较于前一天的涨跌幅。

十、VAE

VAE 属于数据预处理的一部分，目前将过去 20 日的 OCHLV（100 维）降维至 1*2 的 tensor，作为 state 的一部分。

实现方式

alternatively： 和 agent 交替训练

获取一支股票数据后，先训练 vae 获得 state 的 embedding，然后使用 embeding 作为 state 的一部分训练 agent。

关键代码

```
class StockDataset(Dataset):
    def __init__(self,df):
        self.df = df
        stock_data = []
        for i in range(len(self.df)-20):
            temp = self.df.iloc[i:20+i,-6:-1].to_numpy()
            stock_data.append(temp)
        self.stock_data = stock_data


    def __getitem__(self,index):
        data = self.stock_data[index]
        data = torch.tensor(data,dtype=torch.float32)
        return data

    def __len__(self):
        return len(self.stock_data)



class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
        self.fc1 = nn.Linear(100, 80)
        self.fc21 = nn.Linear(80, 2)
        self.fc22 = nn.Linear(80, 2)
        self.fc3 = nn.Linear(2, 80)
        self.fc4 = nn.Linear(80, 100)

    def encode(self, x):
        h1 = F.relu(self.fc1(x))
```

```
        return self.fc21(h1), self.fc22(h1)

    def reparametrize(self, mu, logvar):
        # standardization
        std = logvar.mul(1.0e-5).exp_()
        if torch.cuda.is_available():
            eps = torch.cuda.FloatTensor(std.size()).normal_()
        else:
            eps = torch.FloatTensor(std.size()).normal_()
        eps = Variable(eps)
        return eps.mul(std).add_(mu)

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        # return F.sigmoid()
        # return torch.sigmoid(self.fc4(h3))
        return self.fc4(h3)

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparametrize(mu, logvar)
        # print('x',x)
        # print('self.decode(z)',self.decode(z))
        return self.decode(z), mu, logvar


def loss_function(recon_x, x, mu, logvar):
    """
    recon_x: generating images
    x: origin images
    mu: latent mean
    logvar: latent log variance
    """

    BCE = reconstruction_function(recon_x, x)  # mse loss
    KLD_element =
mu.pow(2).add_(logvar.exp()).mul_(-1).add_(1).add_(logvar)
    KLD = torch.sum(KLD_element).mul_(-0.5)
    return BCE + KLD
```

十一、predictor

watchlist

watchlist 用于

- class WatchList() 用于用户加入心仪的股票代码，维护一个 watchlist，持久化保存在数据库
- inference() 对 watchlist 的股票做 inference，若得到[1, -1]，发出邮件

注：watchlist 写成固定的 list 直接放在 predictor 中。

predictor

- predictor 的作用是加载一个 state，让 agent 选出当前模型给出的最优动作
- 不要直接继承 DQN 算法中的 def predict()，虽然目前的效果是一样的，但是 predict.py 中的 predict 功能要更加自由，可以实现 model 的替换，使它做出任意时刻保存模型的预测。
- 调通其他算法后，可以考虑让其他算法中的 def predict()直接继承 predict.py

模型的保存和加载

因为该项目使用的 DQN 网络较小，因此选择直接保存整个网络的方式保存模型：

```
# 保存整个网络
torch.save(model,PATH)

# 对应的加载方式
model = torch.load(PATH)
model.predict()
```

```
class Predictor:

    def __init__(self,model_path):
        self.model = DQN(240, 3, **config).target_net
        self.model.load_state_dict(torch.load(model_path))
        # self.model = torch.load(model_path)

    def get_state(self,ticker):
        start_time =
(datetime.datetime.now()-datetime.timedelta(days=100)).strftime("%Y-%m-
%d")
        end_time = datetime.datetime.now().strftime('%Y-%m-%d')
```

```
        # 获取最近 100 天数据，处理后截取 20 天作为 state
        data_tmp = ts.pro_bar(ts_code=ticker, adj='qfq', freq="d",
                start_date=start_time, end_date=end_time)
        pre = util.preprocessor.Preprocessor(df=data_tmp)
        data = pre.process()
        data['embedding'] = [float(i) for i in data['embedding']]
        state = data.loc[data.index[-20:], ['kdjk', 'kdjd', 'kdjj',
'rsi_6', 'rsi_12', 'rsi_24', "open_", "close_", "high_", "low_",
"volume_",'embedding']]
        state = torch.Tensor(state.values).flatten()
        return state

    def predict(self, state):
        q_values = self.model(state)
        action_map = {-1:'sell',0:'hold',1:'buy'}
        action = q_values.tolist().index(q_values.max()) - 1
        action = action_map[action]
        return action
```

十二、trainer

设计思路:

processed → trainer → trained

关键代码：

```
    def train(self):
        print('Start to train !')
        print(f'Env:{self.env}, Algorithm:{self.config["algo"]},
Device:{self.config["device"]}')
        rewards = []
        ma_rewards = []  # moving average reward
        for i_ep in range(self.config["train_eps"]):
            reward_detail = []
            state = self.env.reset()
            done = False
            ep_reward = 0
            ep_step = 0
            while ep_step < (len(self.env.df) - 2*self.env.window_size):
                ep_step += 1
                action = self.agent.choose_action(state)
                next_state, reward, done, _ = self.env.step(action)
                ep_reward += reward
                self.agent.memory.push(state, action, reward, next_state,
done)
                reward_detail.append(reward)
                state = next_state
                self.agent.update()
                if done:
                    break
            # save ma rewards
            if ma_rewards:
                ma_rewards.append(0.9 * ma_rewards[-1] + 0.1 * ep_reward)
            else:
                ma_rewards.append(ep_reward)
            if (i_ep + 1) % self.config["target_update"] == 0:
                self.agent.target_net.load_state_dict(self.agent.policy
_net.state_dict())
            if (i_ep + 1) % 10 == 0:
```

```
                print('Episode:{}/{}, Reward:{}'.format(i_ep + 1,
self.config["train_eps"], ep_reward))
            if (i_ep + 1) % self.config["save_model"] == 0:
                torch.save(self.agent.target_net.state_dict(),
self.model_dir + str(i_ep) + ".pth")
            if (i_ep + 1) % self.config["save_result"] == 0:
                result = self.env.save_transaction_information()
                # visualize signal
                plot_signal(result)
                result.to_csv(self.result_dir+ str(i_ep) + ".csv")
            rewards.append(ep_reward)

        print('Complete training! ')

        return rewards, ma_rewards
```

十三、main

main.py 用于驱动整个 trainer 的工作，与 predictor 相互独立。大致的流程是：获取随机一条股票的数据→保存→预处理（转换 oclhv 为环比、增加技术指标、拐点等）→ 训练 → 保存模型和预测信号→绩效计算→显示 reward、收益率、sharpe ratio、资产的增长，显示此次训练的股票曲线叠加买卖信号。

而 portfolio 的作用则是加载训练好的模型→加载 watchlist 的股票代码→获取最新数据并预测出未来信号→保存数据和信号→延迟的绩效评估

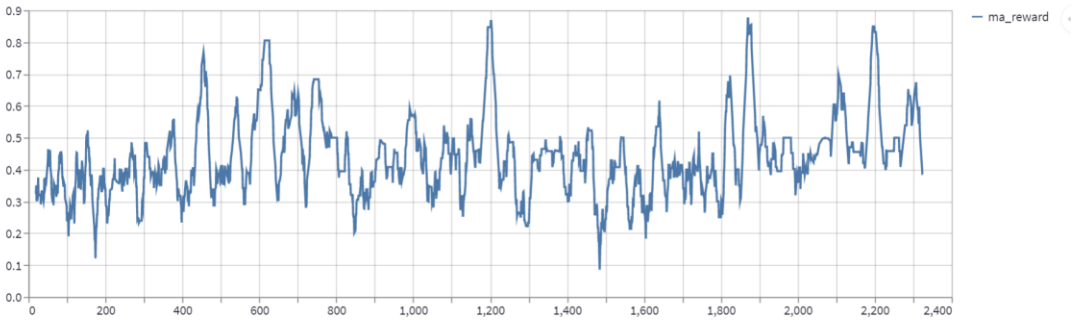
```
if __name__ == "__main__":
    start_time = datetime.datetime.now()
    '''get data'''
    raw_data = get_data.get_random_data()
    '''preprocess'''
    pre = preprocessor.Preprocessor(df=raw_data)
    train_data = pre.process()
    train_data['embedding'] = [float(i) for i in train_data['embedding']]
    '''train'''
    env = StockLearningEnv(train_data)
    agent = DQN(env.state_space, env.action_space.n, **config)
    trainer = Trainer(config, agent, env)
    breakpoint()
    trainer.create_data_dir()
    trainer.train()
    end_time = datetime.datetime.now()
    print("training finished, time {}".format(end_time - start_time))
```

十四、streamlit 前端

用 **streamlit** 做展示；

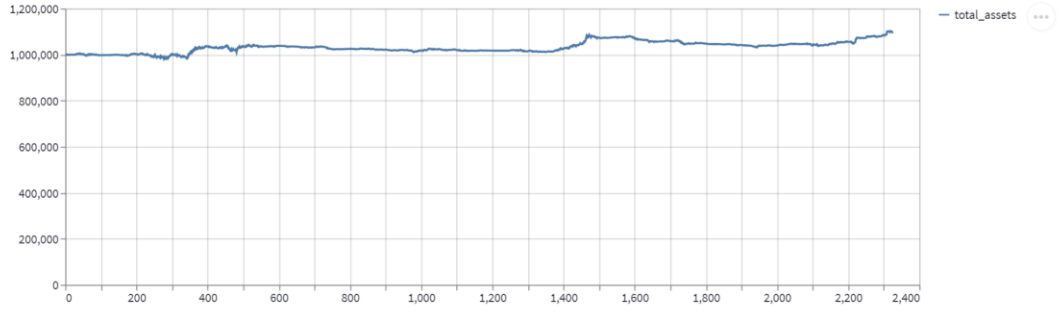
url: www.jnuecthg.top

ma reward

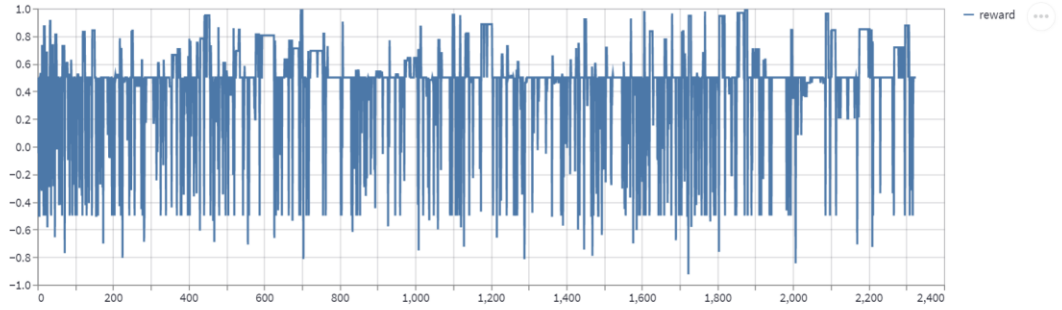


本次训练轮数:12

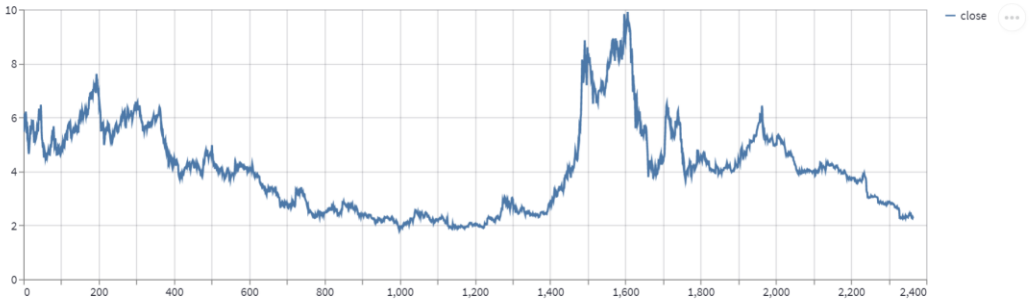
total assets



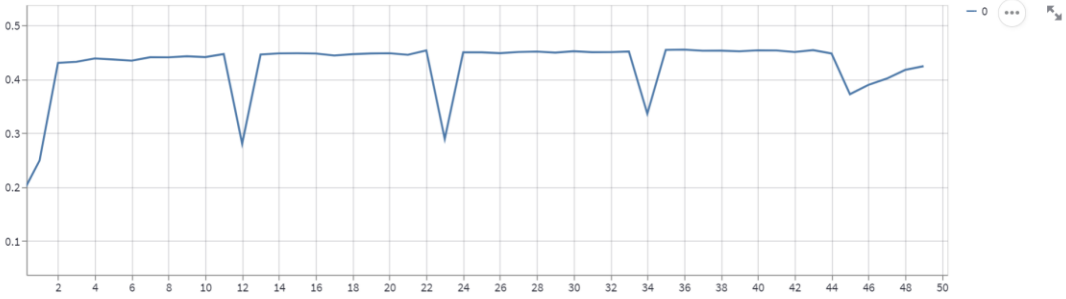
reward



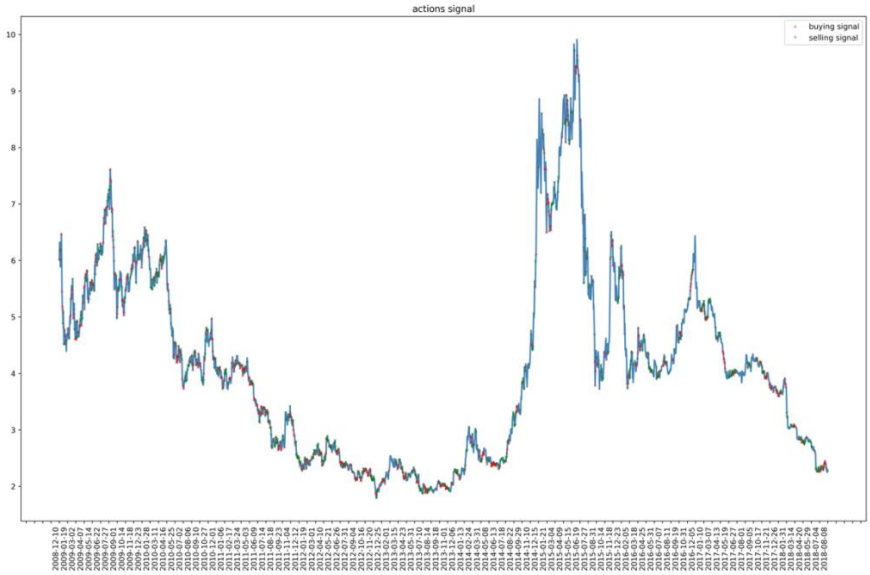
close price for 601099.SH



mean reward for every episode



singal



predict action for 000555.SZ is hold

predict action for 601099.SH is hold

```
def run(self):
    # close price
```

```
        con = sqlite3.connect(os.path.join(os.getcwd(), 'data.db'))
        df = pd.read_sql('SELECT * FROM train_data',con,index_col='index')
        ticker = df['tic'].values[0]
        st.markdown(f"## close price for {ticker}")
        close_data = pd.DataFrame(df['close'])
        st.line_chart(data=close_data, width=0, height=0,
use_container_width=True)

        # mean reward for every episode
        episode_ma_reward = []
        for root, dirs, files in os.walk("./outputs/result"):
            for i in files:
                df = pd.read_csv('./outputs/result/' + i)
                episode_ma_reward.append(df['reward'].mean())
        st.markdown(f"### mean reward for every episode")
        st.line_chart(episode_ma_reward)

        # read result data
        for root, dirs, files in os.walk("./outputs/result"):
            result = files
        selected_result_file = st.sidebar.selectbox("choose a result file",
result)
        df = pd.read_csv('./outputs/result/' + selected_result_file)
        i_ep = str(df['episode'][0])

        st.markdown("### 本次训练轮数:" + i_ep)



        # total assets
        st.markdown(f"### total assets")
        st.line_chart(df['total_assets'])

        # reward
        st.markdown(f"### reward")
        st.line_chart(df['reward'])

        # ma_reward
        st.markdown(f"### ma reward")
        df['ma_reward'] = df['reward'].rolling(20).mean()
        st.line_chart(df['ma_reward'])

        # show singal
        st.markdown(f"### singal")
        image = Image.open('./outputs/image/' + i_ep + '.png')
        st.image(image)
```

```
        # prediction
        input_code = watchlist
        predictor = Predictor(r'./outputs/model/3.pth')
        for ticker in watchlist:
            state = predictor.get_state(ticker)
            action = predictor.predict(state)
            st.markdown(f'#### predict action for {input_ ticker} is
{action}')
        return None
```

十五、项目总结

该项目现阶段工作已经搭建起 RL Market Timing 的基本框架，未来的改进版本可能会从以下几个部分进行：

1.agent 替换算法

Version2 — RL 其他经典算法系列，如 A2C, SAC, PPO 等

Version3 — 基于 Transformer 做序列预测

Temporal Fusion Transformers for Interpretable Multi-horizon Time Series Forecasting（Google Cloud AI, USA）

version3 — Transformer+RL 做序列预测

此版本已经进入科研阶段

2.get data

获取更加丰富的股票数据，如将日线替换成 5 分线，然后使用 vae 降维

3.reward

目前的 reward 设计思路较为直接，主要考虑因素为买卖点于拐点的距离，后续可思考其他方案。