Notes on Portfolio Optimization with FinRL and Stable Baselines 3 - v1.0

2022/09/14, AJ Zerouali

Updated: 22/10/17

A set of notes for the implementation of portfolio optimization using the FinRL library. Ideally, this notebook should be viewed as a user guide for future modifications of FinRL.

The contents of this document grew from an effort to clarify the following tutorial on the library's GitHub page: https://github.com/AI4Finance-Foundation/FinRL/blob/master/tutorials/2-Advance/FinRL_PortfolioAllocation_Explainable_DRL.ipynb

Contents:

---

1 - The FinRL algorithmic trading pipeline

We start with the template of FinRL trading algorithm. The idea is to summarize the main function calls in one place, to have a roadmap for the upcoming sections where each task is discussed in more detail. The references for this section are the following tutorials on GitHub:

- Basic algo trading: https://github.com/AI4Finance-Foundation/FinRL/blob/master/tutorials/1-Introduction/Stock_NeurIPS2018_SB3.ipynb
- Portfolio optimization: https://github.com/AI4Finance-Foundation/FinRL/blob/master/tutorials/2-Advance/FinRL_PortfolioAllocation_Explainable_DRL.ipynb
- Paper trading with Alpaca: https://github.com/AI4Finance-Foundation/FinRL/blob/master/tutorials/3-Practical/FinRL_PaperTrading_Demo.ipynb We go into more detail in the next subsections.

a) Stock trading pipeline:

This is the first tutorial one consults for getting a basic understanding of FinRL: https://github.com/AI4Finance-Foundation/FinRL/blob/master/tutorials/1-Introduction/Stock_NeurIPS2018_SB3.ipynb.

The main function calls are as follows:

1) **Download data from Yahoo Finance**

In [ ]:
```
df = YahooDownloader(start_date = TRAIN_START_DATE,
                     end_date = TRADE_END_DATE,
                     ticker_list = config_tickers.DOW_30_TICKER).fetch_data()
```

This creates a dataframe with the list of tickers used for our portfolio, with the usual prices and volume. The *YahooDownloader()* function is imported from the *preprocessor* submodule in finrl/meta, which wraps the main functions of Yahoo Finance's API (yfinance).

2) **Data pre-processing**

Feature engineering - Clean data, compute technical indicators and other user-defined quantities:

In [ ]:
```
fe = FeatureEngineer(
        use_technical_indicator=True,
        tech_indicator_list = INDICATORS,
        use_vix=True,
        use_turbulence=True,
        user_defined_feature = False)
processed = fe.preprocess_data(df)
```

Training/Trading data split:

In [ ]:
```
train = data_split(processed_full, TRAIN_START_DATE,TRAIN_END_DATE)
trade = data_split(processed_full, TRADE_START_DATE,TRADE_END_DATE)
```

The *FeatureEngineer()* and *data_split()* functions are imported from the *preprocessor/preprocessors* submodule in finrl/meta: https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/meta/preprocessor/preprocessors.py.

3) **Create the Gym/SB3 environment**

In [ ]:
```
# Initialize args
stock_dimension = len(train.tic.unique()) # No. of stocks
state_space = 1 + 2*stock_dimension + len(INDICATORS)*stock_dimension # State space dim
env_kwargs = {
    "hmax": 100,
    "initial_amount": 1000000,
    "num_stock_shares": num_stock_shares,
    "buy_cost_pct": buy_cost_list,
    "sell_cost_pct": sell_cost_list,
```

```
    "state_space": state_space,
    "stock_dim": stock_dimension,
    "tech_indicator_list": INDICATORS,
    "action_space": stock_dimension,
    "reward_scaling": 1e-4
}

# Instantiate FinRL StockTradingEnv()
e_train_gym = StockTradingEnv(df = train, **env_kwargs)

# Convert to stable_baselines3 environment
env_train, _ = e_train_gym.get_sb_env()
```

The implementation of *StockTradingEnv()* is at the following link:

https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/meta/env_stock_trading/env_stocktrading.py.

This environment is in principle implemented in 3 libraries: ElegantRL, RLLib and SB3. For the sake of stability, we only look at the SB3 implementation, which is discussed in detail in section 2 of this notebook.

4) **Create and train trading agent**

Taking a DDPG agent for instance:

In [ ]:
```
# Instantiate agent
agent = DRLAgent(env = env_train)

# Get stable_baselines3 DDPG model
model_ddpg = agent.get_model("ddpg")

# Train model
trained_ddpg = agent.train_model(model=model_ddpg,
                                 tb_log_name='ddpg',
                                 total_timesteps=50000)
```

The *DRLAgent()* class is discussed in detail in section 3 below. The implementation is here: https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/agents/stablebaselines3/models.py. The training part is discussed in section 5 below.

5) **Backtest with trained agent**

Create the trading environment with test data:

In [ ]:
```
e_trade_gym = StockTradingEnv(df = trade, turbulence_threshold = 70,
                              risk_indicator_col='vix', **env_kwargs)
```

Run the backtest:

In [ ]:
```
df_account_value, df_actions = DRLAgent.DRL_prediction(model=trained_ddpg,
                                                       environment = e_trade_gym)
```

The simplicity of this last instruction is one of the main attractive features of the FinRL library. The *DRL_prediction()* method is discussed in section 6 of this notebook.

6) **Plot backtest results using pyfolio**

In [ ]:
```
backtest_plot(df_account_value,
              baseline_ticker = '^DJI',
              baseline_start = df_account_value.loc[0,'date'],
              baseline_end = df_account_value.loc[len(df_account_value)-1,'date'])
```

This last instruction produces several performance plots along with statistics of our strategy. The implementation is at the following link: https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/plot.py.

It's worth noting that this function imports the *timeseries* submodule of PyFolio. At the time of writing, this file has a bug which is fixed here: https://stackoverflow.com/questions/63554616/attributeerror-numpy-int64-object-has-no-attribute-to-pydatetime.

b) Portfolio optimization pipeline:

In this subsection, our reference is the portfolio optimization tutorial here:

https://github.com/AI4Finance-Foundation/FinRL/blob/master/tutorials/2-Advance/FinRL_PortfolioAllocation_Explainable_DRL.ipynb.

Setting aside some key differences that we will discuss later, the pipeline is rather similar to the one discussed in subsection (1.a) above.

Before getting into the main differences, here are the main calls:

1) **Download data with yfinance**

In [ ]:
```
df = YahooDownloader(start_date = '2008-01-01',
                     end_date = '2021-09-02',
                     ticker_list = config_tickers.DOW_30_TICKER).fetch_data()
```

2) **Feature engineering**

In [ ]:
```
fe = FeatureEngineer(
                    use_technical_indicator=True,
                    use_turbulence=False,
                    user_defined_feature = False)
df = fe.preprocess_data(df)

### Add covariance matrix to states
df=df.sort_values(['date','tic'],ignore_index=True)
df.index = df.date.factorize()[0]
cov_list = []
return_list = []
lookback=252 # look back is one year

for i in range(lookback,len(df.index.unique())):
  data_lookback = df.loc[i-lookback:i,:]
  price_lookback=data_lookback.pivot_table(index = 'date',columns = 'tic', values = 'close')
  return_lookback = price_lookback.pct_change().dropna()
  return_list.append(return_lookback)
  covs = return_lookback.cov().values
  cov_list.append(covs)

df_cov = pd.DataFrame({'date':df.date.unique()lookback:],
                    'cov_list':cov_list,'return_list':return_list})
df = df.merge(df_cov, on='date')
df = df.sort_values(['date','tic']).reset_index(drop=True)

### Train/test data split
train = data_split(df, '2009-01-01','2020-06-30')
trade = data_split(df,'2020-07-01', '2021-09-02')
```

3) **Create Portfolio Optimization environment**

In [ ]:
```
### Init. env. arguments
stock_dimension = len(train.tic.unique())
state_space = stock_dimension
print(f"Stock Dimension: {stock_dimension}, State Space: {state_space}")
tech_indicator_list = ['macd', 'rsi_30', 'cci_30', 'dx_30']
feature_dimension = len(tech_indicator_list)
```

```
env_kwargs = {
    "hmax": 100,
    "initial_amount": 1000000,
    "transaction_cost_pct": 0,
    "state_space": state_space,
    "stock_dim": stock_dimension,
    "tech_indicator_list": tech_indicator_list,
    "action_space": stock_dimension,
    "reward_scaling": 1e-1

}

### Instantiate portfolio alloc. environment
e_train_gym = StockPortfolioEnv(df = train, **env_kwargs)
env_train, _ = e_train_gym.get_sb_env()
```

4) **Instantiate and train model (PPO)**

In [ ]:
```
agent = DRLAgent(env = env_train)
PPO_PARAMS = {"n_steps": 2048,"ent_coef": 0.005,"learning_rate": 0.001,"batch_size": 128,}
model_ppo = agent.get_model("ppo",model_kwargs = PPO_PARAMS)
trained_ppo = agent.train_model(model=model_ppo,
                                tb_log_name='ppo',
                                total_timesteps=40000)
```

5) **Backtest trained agent**

In [ ]:
```
e_trade_gym = StockPortfolioEnv(df = trade, **env_kwargs)
df_daily_return_ppo, df_actions_ppo = DRLAgent.DRL_prediction(model=trained_ppo,
                                                              environment = e_trade_gym)
```

Note (22/09/15): To be continued. A functional version of the script is given in the *FinRL_PortfolioOptimization_Reduced.ipynb*.

c) Paper trading with Alpaca:

Reference: https://github.com/AI4Finance-Foundation/FinRL/blob/master/tutorials/3-Practical/FinRL_PaperTrading_Demo.ipynb

Note (22/09/15): Will write-up the details later. The tutorial above has several critical issues, as it was initially written for ElegantRL which is completely broken at the time of writing. I modified that notebook to run with SB3, but the paper-trading algorithm also has critical problems.

---

2 - Stable Baselines 3

FinRL is mainly a library of wrappers for financial applications, and when it comes to RL implementations, this section is maybe the most important part of these notes. We address the following topics.

a) Environment classes

- a.1 - The *VecEnv* and *DummyVecEnv* classes
- a.2 - The *gym.Env* class

b) Models I - Algorithm classes

- b.1 - Basics
- b.2 - Abstract classes: *BaseAlgorithm* and *On/OffPolicyAlgorithm*

- b.3 - The *learn()* and *train()* methods
- b.4 - The *collect_rollouts* method
- b.5 - Saving and loading models

c) Models II - Neural nets for deterministic policies

- c.1 - Basics
- c.2 - The *common.torch_layers* submodule
- c.3 - Example: The *TD3.td3* and *TD3.policy* submodules

d) Models III - Neural nets for stochastic policies

- d.1 - The *ActorCriticPolicy* class
- d.2 - The *common.distributions* submodule
- d.3 - Example: The *PPO.ppo* and *PPO.policy* submodules

e) Loggers

f) TensorBoard and Callbacks

The main references for the discussion below are the following pages:

- SB3 official docs: https://stable-baselines3.readthedocs.io/en/master/index.html
- SB3 GitHub: https://github.com/DLR-RM/stable-baselines3
- Tutorial repo: https://github.com/araffin/rl-tutorial-jnrr19

a) Environment classes

a.1 - The *VecEnv* and *DummyVecEnv* classes

The main abstract environment class of SB3 is the vectorized environment *VecEnv*.

- A description of vectorized envs is given here: https://stable-baselines3.readthedocs.io/en/master/guide/vec_envs.html;
- The class is implemented at: https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/vec_env/base_vec_env.py.

The idea is to allow the possibility of having multiple agents evolving in stacked environments (particularly important when dealing with Atari frames for states), and thus get vectorized actions, rewards and state spaces with various shapes. Just as with Gym, there is a plethora of wrappers that the library uses, which are briefly described at the first link above.

There are two subclasses of *VecEnv* that are called in practice. From the docs:

- ***SubprocEnv***: Creates a multiprocess vectorized wrapper for multiple environments, distributing each environment to its own process, allowing significant speed up when the environment is computationally complex.
- ***DummyVecEnv***: Creates a simple vectorized wrapper for multiple environments, calling each environment in sequence on the current Python process. This is useful for computationally simple environment such as cartpole-v1, as the overhead of multiprocess or multithread outweighs the environment computation time. This can also be used for RL methods that require a vectorized environment, but that you want **a single environment** to train with.

The latter is the one used by FinRL for SB3 environments. The implementation is here:

- https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/vec_env/dummy_vec_env.py

Comments and facts on *DummyVecEnv* and *VecEnv*:

- *VecEnv* is quite similar to the *VectorEnv* class of Gym (https://github.com/openai/gym/blob/master/gym/vector/vector_env.py), but in contrast to the latter, *VecEnv* is an **abstract base class** (https://docs.python.org/3/library/abc.html) in SB3.
- Concerning the classes representing state and action spaces, *VecEnv* declares these as *gym.spaces.Space* objects in the constructor.
- Here is an important note from SB3's Vectorized Envs. docs: When using vectorized environments, the environments are automatically reset at the end of each episode. Thus, the observation returned for the i-th environment when *done[i]* is true will in fact be the first observation of the next episode, **not** the last observation of the episode that has just terminated. You can access the "real" final observation of the terminated episode—that is, the one that accompanied the *done* event provided by the underlying environment—using the *terminal_observation* keys in the info dicts returned by the *VecEnv*.
- The *step()* method itself is not implemented in *DummyVecEnv*, it is only inherited from *VecEnv*, where it is implemented in a very abstract manner using the *typing* module:

In [ ]:
```
# Typing imports
from typing import Any, Dict, Iterable, List, Optional, Sequence, Tuple, Type, Union

# VecEnvStepReturn is what is returned by the step() method
# it contains the observation, reward, done, info for each env
VecEnvStepReturn = Tuple[VecEnvObs, np.ndarray, np.ndarray, List[Dict]]

# Class def
class VecEnv(ABC):
    ....
     def step(self, actions: np.ndarray) -> VecEnvStepReturn:
        """
        Step the environments with the given action
        :param actions: the action
        :return: observation, reward, done, information
        """
        self.step_async(actions)
        return self.step_wait()
```

Similarly, the usual *reset()*, *close()*, and *render()* are all abstract methods of *VecEnv* that wrap helper methods implemented in *DummyVecEnv*. For instance, *DummyVecEnv* class implements the *step_wait()* method called by *VecEnv*'s *step()* method.

- In the end, a *DummyVecEnv* instance is constructed from stacked *gym.Env* objects and other environment functions, as one can see from its constructor:

In [ ]:
```
def __init__(self, env_fns: List[Callable[[], gym.Env]]):
    self.envs = [fn() for fn in env_fns]
    env = self.envs[0]
    VecEnv.__init__(self, len(env_fns), env.observation_space, env.action_space)
    obs_space = env.observation_space
    self.keys, shapes, dtypes = obs_space_info(obs_space)
    ...
```

In summary, and in relation to our use in FinRL, the *DummyVecEnv* class is simply a wrapper for the portfolio optimization environment that allows us to use SB3 agents. Concretely, this means that the state/action spaces and rewards constituting our MDP are specified in the *StockPortfolioEnv* class, which subclasses the foundational *gym.Env* class. In the next part we give some reminders on the main moving parts of this class.

To do (22/09/30): Add a comment on how **custom** *step()*/*reset()*/*init()* methods from a gym environment are adopted by the *VecEnv*/*DummyVecEnv* classes. This is particularly important for the discussion of FinRL's

StockPortFolioEnv below, since:

- At first glance, this is not obvious from the code or the docs. What's the mechanism? Check:
  - https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/vec_env/dummy_vec_env.py
  - https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/vec_env/base_vec_env.py
- Unclear at the moment on what the argument

  ```
  env_fns: List[Callable[[], gym.Env]]
  ```
  means.

- Comment on the *DummyVecEnv.env_method(method_name,* method_args) *method. For instance, it is used for non-gym methods proper to* StockPortfolioEnv*.
- Might be useful to give a sequence of method calls here.
- There's more to say about environments in the *BaseAlgorithm* section. Algorithms have an *env* parameter that is wrapped by SB3 before being assigned to *BaseAlgorithm.env*.
- In FinRL, *StockPortfolioEnv* is converted to a *DummyVecEnv* before being passed to the SB3 algorithm. This closes the loop between FinRL's *DRLAgent* class and SB3 models.

Custom environments

Note (22/09/19): See: https://stable-baselines3.readthedocs.io/en/master/guide/custom_env.html

a.2 - The *gym.Env* class

To summarize, this class consists of:

- ***action_space:*** This attribute encodes the actions in the environment, and is typically a subclass of *Space* discussed below.
- ***observation_space:*** Same as above, but this attribute encodes the state space.
- ***reset():*** This method puts resets the agent back to the initial state in the environment. It should always be called after instantiating the *Env* object to obtain the initial observation of the episode.
- ***step():*** This method is used as follows:

In [ ]:
```
next_state, reward, done, info = env.step(action)
```

The input is the action of the agent, and the output consists of the new state (NumPy array), the corresponding reward (float), and the boolean *done* saying whether or not *next_state* is terminal. The *info* output contains information relevant for debugging and learning.

- ***render():*** This method allows to visualize the evolution of the agent. It calls the PyGame package which in turn calls upon some lower-level C functions (one of the prerequisite packages of Gym is *SWIG*, which calls a C++ compiler). One of the args of *render()* is *mode*, which equals '*human*' by default and displays the game screen. Once done with rendering, one must call ***Env.close()*** to shutdown PyGame.

The action space and the state space are represented by the abstract Gym class entitled ***Space***, whose two most relevant methods are:

- ***sample()***, which returns a random sample from the space. This method is typically called when performing actions.
- ***contains(x)***, which checks whether the state x belongs to the space's domain.

These abstract methods are reimplemented in the 3 usual subclasses of *Space*:

- ***Discrete:*** This space class models finite spaces, with elements labelled from 0 to n. The values assigned to each label are described in the environment subclass (see example below).

- ***Box:*** Boxes represent n-dimensional tensor of rational numbers in intervals $[x_{\min}, x_{\max}]$, and have a *dtype* parameter in their constructor. The first use of this class is to define the bounds of a rectangular region that will be discretized in the background, and the *dtype* specifies the desired accuracy. There is also a *shape* argument in the constructor, which for example is used when the states are represented by screenshots of a game (think of the Atari environments). In the case of images of size 210x160 pixels, one calls

In [ ]:
```
            Box(low = 0, high=255, size = (210, 160, 3)),
```

where 3 stands for the RGB channels.

- ***Tuple:*** Some spaces could be of various complexity, such as having discrete and continuous components. The *Tuple* class allows to define such spaces in a nested way, by combining the previous classes for instance (see example of car controls).

In terms of RL algorithms, the *step()* method is the crucial one, as it implements the environment dynamics: This method is typically where the state transitions and rewards are computed. This remark will be of particular importance when returning to FinRL.

Some remarks on *gym.Env*:

- When calling *gym.make('Env_name')*, Gym actually calls a wrapper to create the environment, and not exactly the class itself.
- Gym's pre-built environments do not readily give access to the possible actions in a given state or the rewards corresponding to a new state. To access these, one can use the ***env.unwrapped*** attribute, such as for instance:

In [ ]:
```
env = gym.make('CartPole-v1')
# Get action space
cartpole_action_space = env.unwrapped.action_space
# Get no. of possible actions
cartpole_n_actions = env.unwrapped.action_space.n
```

b) Models I - Algorithm classes

b.1 - Basics

The main documentation can be found at the following links:

- Getting started: https://stable-baselines3.readthedocs.io/en/master/guide/quickstart.html
- Developer guide: https://stable-baselines3.readthedocs.io/en/master/guide/developer.html#
- Algos: https://stable-baselines3.readthedocs.io/en/master/guide/algos.html
- Base RL class: https://stable-baselines3.readthedocs.io/en/master/modules/base.html
- Saving and loading: https://stable-baselines3.readthedocs.io/en/master/guide/save_format.html
- More algorithms: https://github.com/Stable-Baselines-Team/stable-baselines3-contrib

One of the most interesting features of SB3 is the stable implementation of policy gradient and actor-critic methods, along with the intuitive interface for their training and deployment.

Before delving into details, let's start with the algorithms structure discussed in the developer guide above:

- The DRL algorithms that train agents are stored in individual submodules of stable_baselines3.

- Each of these submodules contains two files: policy.py and algo_name.py. The former implements the policy, while the latter implements the training algorithm for the agent.
- In SB3, "policy" refers to all neural nets involved in the algorithm, and not only the learned policy network.
- Each algorithm has **two main methods**. First is *collect_rollouts()* that defines how the samples are collected, and stores them in a *RolloutBuffer* (discarded after grad. update) or a *ReplayBuffer*. Second is the *train()* method that updates the parameters using samples from the buffer.
- All the environments handled by agents are inherited from *VecEnv*.
- At present, the algorithms that SB3 supports (on- and off-policy versions) are A2C, DDPG, DQN, HER, PPO, SAC and TD3.

On top of the algorithms implemented in the main SB3 library, there is a second library called *stable_baselines3-contrib* (sb3-contrib), where there are more experimental algorithms/agents implemented. Among the presently supported algorithms are ARS (Aug. Rand. Search), TRPO, and Recurrent PPO. The GitHub repo is here:

https://github.com/Stable-Baselines-Team/stable-baselines3-contrib

At this point, it's helpful to give an example of how to use SB3 models. Say we would like to train and deploy a PPO agent in the cartpole environment. The code will look like this:

In [ ]:
```
import gym
from stable_baselines3 import PPO
from stable_baselines3.common.env_util import make_vec_env

# Create 4 parallel environments
env = make_vec_env("CartPole-v1", n_envs=4)

# Instantiate agent
model = PPO("MlpPolicy", env, verbose=1)

# Train
model.learn(total_timesteps=25000)

# Save
model.save("ppo_cartpole")

obs = env.reset()
while True:
    action, _states = model.predict(obs)
    obs, rewards, dones, info = env.step(action)
    env.render()
```

Loading a trained agent is done as follows:

In [ ]:
```
from stable_baselines3 import PPO
model = PPO.load("ppo_cartpole")
```

As mentioned above, any DRL algorithm consists of two parts:

- (A) An algorithm class inherited from **BaseAlgorithm(ABC)**, which implements the sample collection (*collect_rollouts()*) and the corresponding learning algorithm (*train()* and *learn()*).
- (B) A policy class inherited from **BasePolicy(torch.nn.Module)**, which gathers all neural nets used in the *collect_rollouts()* and *learn()* methods of the algorithm class. Any instance of a *BaseAlgorithm* subclass has a *policy* attribute that is a *BasePolicy* object.

The *BasePolicy* class is discussed in Section (2.c). For the remainder of Section (2.b) we focus on the specifics of the subclasses of *BaseAlgorithm*.

b.2 - Abstract classes - *BaseAlgorithm*, *On/OffPolicyAlgorithm*

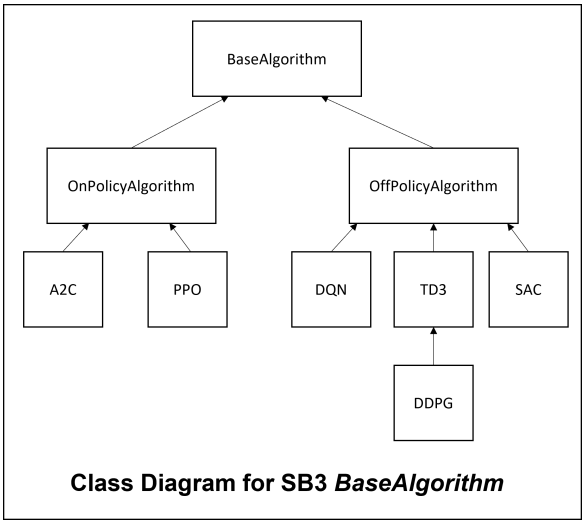
The abstract base class underlying all SB3 algorithms is the *BaseAlgorithm* class in *stable_baselines3.common.base_class*. The implementation can be found here:

https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/base_class.py

Stable Baselines 3 separates its DRL implementations into on-policy algorithms and off-policy ones. As such, any algorithm implemented in this library is a subclass of either *OnPolicyAlgorithm* or *OffPolicyAlgorithm*, (inherited from *BaseAlgorithm*). These are also in the submodule *stable_baselines3.common*, and their implementations are at:

- On: https://github.com/DLR-RM/stable-baselines3/blob/88e1be9ff5e04b7688efa44951f845b7daf5717f/stable_baselines3/common/on_policy_algorithm.py
- Off: https://github.com/DLR-RM/stable-baselines3/blob/88e1be9ff5e04b7688efa44951f845b7daf5717f/stable_baselines3/common/off_policy_algorithm.py

We have the following class diagram for the algorithms implemented at the time of writing:



Class Diagram for SB3 *BaseAlgorithm*

For the sake of completeness and for future reference, we close this part by listing some of the interface components of SB3 on/off-policy classes.

Attributes:

- *policy*: *Policy* object, discussed in more detail below. Can be built at instantiation if *_init_setup_model*=True, and customized by passing the attribute *policy_kwargs* to the constructor.
- *env*: The *gym.Env* environment to learn from. **(Requires comments)**
- *learning_rate*: Learning rate for the optimizer, can be a function of the current progress remaining.
- *gamma*: Discount factor.
- *action_noise*: Action noise type (*None* by default) for hard exploration problems. See *common.noise* for the different action noise types.
- *seed*: Seed for the pseudo-random generators.
- *use_sde*: Whether to use generalized State Dependent Exploration (gSDE) instead of action noise exploration (default: False). The sample frequency can be customized with *sde_sample_freq*.
- *device*: Specify CPU or GPU. Will try to use a Cuda compatible device by default and fallback to CPU otherwise.
- For *OnPolicyAlgorithm*, one has attributes *ent_coef* (entropy coeff.), *vf_coef* (value f'n coeff.), and *max_grad_norm*.
- For *OffPolicyAlgorithm*, one has attributes *batch_size*, *buffer_size*, *learning_starts*, *tau* (soft update coeff.), *train_freq*, *gradient_steps* etc.
- *supported_action_spaces*: The action spaces supported by the algorithm.
- In addition to the above, we also have the following attributes:
  - For training monitoring: *monitor_wrapper*, *tensorboard_log*, and *verbose*.
  - More environment attributes: e.g. *support_multi_env*, *create_eval_env*.

The policies are handled using the *stable_baselines3.common.policies* submodule, and the replay buffer classes are in the *stable_baselines3.common.buffers* submodule.

Methods:

- *__init__()*: The specific parameters of the constructor depend on the algorithm type (on/off-policy).
- ***collect_rollouts()***: Collects experiences using the current policy and fills a *RolloutBuffer*. "Rollout" here refers to the model-free RL notion and **should not be confused** with the rollout concept of model-based RL or planning. This method is implemented in the *On/OffPolicyAlgorithm* classes (not in the Base), and calls the *step()* method of the environment, .
- ***train()***: This is an abstract method of *On/OffPolicyAlgorithm*, and is **implemented for each deep RL algorithm individually**. In particular, this is the method containing the usual PyTorch training loop with *loss.backward()* and *optimizer.step()* (for a concrete example, see implementation at https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/dqn/dqn.py).
- ***learn()***: Returns a trained model, included in the Base class and implemented differently in *On/OffPolicyAlgorithm*. The purpose of this method is to manage the *logger* and *callback* objects during training. It first calls *_setup_learn* to initialize training parameters, and next loops over time steps to collect rollouts using the method above. Once all the callback/logger instructions are done, the method *train* is called to update network weights.
- ***predict()***: This is a method of *BaseAlgorithm* that wraps the *predict()* method of the *policy* attribute (see below).
- *get_parameters()*: Returns the parameters of the agent. This includes parameters from different networks, e.g. critics (value functions) and policies (pi functions).
- The remaining methods manage several other aspects of the interface. For instance:
  - More deep learning functions: *_setup_model*, *_setup_lr_schedule*, *_update_current_progress_remaining*, *_update_learning_rate*, *_get_torch_save_params*, *_setup_learn*, *set_random_seed*, (loads files),
  - Saving and loading: *set_parameters*, *load*, *save*
  - Environment methods: *_wrap_env*, *_get_eval_env*, *get_env*, *set_env*, *get_vec_normalize_env*.
  - Memory buffer methods: *_update_info_buffer*,
  - Callback methods: *_init_callback*,
  - Logger methods: *set_logger*, *logger*, *_dump_logs*.

b.3 - The *learn()* and *train()* methods

Regarding these two methods, here are the points to keep in mind:

- The first notable difference between the *On/OffPolicyAlgorithm* classes is their *learn()* methods. Both call the methods *_setup_learn()*, *callback.on_training_start()*, *callback.on_training_end()*, *collect_rollouts()*, and both contain the main training loop where the *train()* method is called. The main difference is that *OnPolicyAlgorithm.learn()* records the training process in the *logger* attribute.

- The *train()* method is where the deep reinforcement learning algorithms are effectively implemented. Indeed, *train()* is an abstract method of *On/OffPolicyAlgorithm.learn()*, and it is only implemented in their subclasses (A2C, PPO, TD3, DDPG, SAC and DQN).

- The *collect_rollouts()* method is discussed in Sec. 2.b.4 below.

(Requires more comments)

b.4 - The *collect_rollouts()* method

This method is implemented in the *On/OffPolicyAlgorithm* classes. **(how different are they)**

b.5 - Saving and loading models

- When loading a trained model, you do not need to have an environment argument.
- SB3 saves a model into a zip file, containing the PyTorch weights and a text file with training params.
- More explanations found in the docs: https://stable-baselines3.readthedocs.io/en/master/guide/save_format.html.

(Finish this)

Our next topic is the handling of neural networks and policies in *stable_baselines3*.

c) Models II - Neural nets for deterministic policies

c.1 - Basics

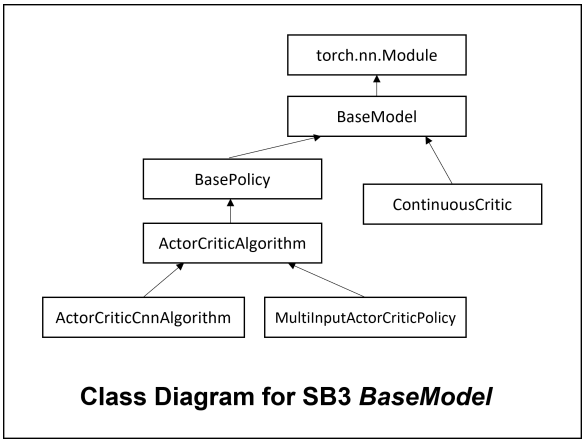
The submodule of interest here is *stable_baselines3.common.policies*, which can be found at:

https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/policies.py.

More information on the structure of SB3's neural nets can be found at the article on customization of policies:

https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html

Here we look at the *BaseModule*, *BasePolicy*, and their subclasses that constitute the *policy* attribute of *BaseAlgorithm*. As previously stated, this *BaseAlgorithm.policy* stores and manages **all** of the neural networks involved in a given algorithm, not only the policy network. Here is the class diagram of *BaseModule/BasePolicy*:



For the remainder Section 2.(c), we describe some implementation details of the neural nets used for deterministic gradient algorithms, by focusing on TD3 for the sake of concreteness. The stochastic policies will be discussed in Section 2.(d).

c.2 - The *common.torch_layers* submodule

Neural networks in SB3 are separated into two main parts:

- A **feature extractor** network. This could be a CNN for instance when dealing with images, or a "combined" feature extractor when using Dict environments. This part of the model is encoded in the **features_extractor** attribute of an algorithm's policy class.
- A **fully-connected network** mapping features to the policy network (actions) or to the Q-function (values). The various SB3 policy classes (*ActorCriticPolicy*, *Actor*, *ContinuousCritic*, etc.) implement the actor and critic networks differently, depending on whether the policy is deterministic or stochastic.

The feature extractor of an SB3 "policy" is implemented as a separate class in the *common.torch_layers* submodule. The code and the documentation can be found at the following links:

- Code: https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/torch_layers.py.
- Custom policies: https://stable-baselines3.readthedocs.io/en/master/guide/custom_policy.html.

The main facts to keep in mind about the feature extraction layers are the following:

- The feature extractor of any SB3 policy is made of the level of the abstract *BaseModel* class. In the constructor of *BaseModel*, the relevant parameters are ***observation_space***, ***features_extractor_class***, ***features_extractor_kwargs***, and ***features_extractor***. By default, this contructor instantiates the *features_extractor* attribute using the *BaseModel.make_features_extractor()* method.
- For **off-policy algorithms**, SB3 uses the classes *Actor[BasePolicy]* and *ContinuousCritic[BasePolicy]* to encode the actor and critic networks respectively. These have *net_arch* and *activation_fn* parameters to specify the layers of these networks on top of the feature extractor, and are built using the *torch_layers.create_mlp()* helper function.
- For **on-policy algorithms**, the actor and critic networks are built quite differently, as they are encoded by the *common.policies.ActorCriticPolicy* class, which uses the *torch_layers.MlpExtractor* class instead of *Actor* and *ContinuousCritic*. We discuss this point in more detail in Section 2.(d) below, when dealing with stochastic policies.

The next subsection gives a concrete example of the considerations here. For the sake of completeness, we close this part with a summary of the contents of the *torch_layers* submodule

Classes:

- *BaseFeaturesExtractor(nn.Module)*: Base class for feature extractors.
- *FlattenExtractor(BaseFeaturesExtractor)*: Feature extractor flattening input. Used as placeholder when feature extraction not needed.
- *NatureCNN(BaseFeaturesExtractor)*: The CNN from Mnih et al 2015 DQN paper.
- *CombinedExtractor(BaseFeaturesExtractor)*: Used with *Dict* observation spaces. Builds a feature extractor for each key of the dictionary, so that inputs are fed through separate submodules, and then concatenated through an additional "combined" MLP net.
- *MlpExtractor(nn.Module)*: Constructs an MLP whose inputs are outputs of a previous feature extractor (e.g. CNN) or environment observations. Outputs a latent representation for the policy and value networks. Has a *net_arch* parameter that specifies the no. and size of hidden layers, as well as the no. of shared layers between the policy and value nets.

Helper functions:

- *create_mlp(input_dim, output_dim, net_arch, activation_fn, squash_output)*: Creates a multi layer perceptron (MLP), which is a collection of fully-connected layers each followed by an activation function. Output is a list of *nn.Module* objects.
- *get_actor_critic_arch(net_arch)*: Gets the policy and value network architectures for the off-policy actor-critic algorithms (SAC, DDPG, TD3). Has a *net_arch* argument that specifies the no. and size of hidden layers. Note: Other than the feature extraction layers, no other layers can be shared by the actor and critic networks. This is to avoid evaluation issues with the target networks.

c.3 - Example: The *TD3.td3* and *TD3.policies* submodules

The source code of this part is at the following links:

- TD3 algorithm: https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/td3/td3.py
- TD3 policy: https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/td3/policies.py

- Off-policy algorithms: https://github.com/DLR-RM/stable-baselines3/blob/88e1be9ff5e04b7688efa44951f845b7daf5717f/stable_baselines3/common/off_policy_algorithm.py

We note that our goal here is to give an idea of how the library is organized, and to give the readers a starting point to implement their own policies. As such, the present description of SB3 is in no way exhaustive, as there are several features that are not discussed (for example, the use of state-dependent exploration noise, or the structure of the replay buffers in *common.buffers*).

In Section 2.(b.1), we gave a template of how an SB3 algorithm is instantiated, trained and deployed. We revisit this template below with comments on each step, where we give more details on the objects, functions and submodules involved.

In [ ]:
```
# Create the environment
env = make_vec_env("CartPole-v1", n_envs=1)
'''
    Some comments on environments:
    1) The instruction above is the same as gym.make("env_name"),
    except that we are wrapping the gym.Env object with a SB3 VecEnv class.
    More generally, any SB3 algorithm is instatiated
    with an env parameter.

    2) If an SB3 model is created with a gym.Env object
    for the env argument, the BaseAlgorithm constructor
    wraps it with a VecEnv object using:
        env = self._wrap_env(env, self.verbose, monitor_wrapper)
        (Line 159 of common.base_class)
    This method looks for appropriate environment and monitor
    wrappers and re-orders image channels.
'''

# Instantiate TD3 algorithm object
model = TD3(policy = "MlpPolicy", env=env, verbose=1)
'''
    This instruction creates a TD3 object, with arguments specifying those of
    the corresponding TD3Policy object (TD3.policy attribute).

    I - TD3Policy Constructor arguments:
    ------------------------------------
    1) For BaseAlgorithm, the policy argument is either "MlpPolicy", "CnnPolicy",
    or "MultiInputPolicy". These 3 aliases are always defined in the corresponding
    algorithm submodule. In the case of TD3, MlpPolicy = TD3Policy, and the feature
    extractor class is FlattenExtractor. "CnnPolicy" is a subclass of TD3Policy for
    which the feature extractor is a CNN (NatureCNN by default), typically used for
    image data. "MultiInputPolicy" is a subclass of MlpPolicy for which the
    environment has a Dict observation space, and whose feature extractor class
    is CombinedFeatureExtractor.

    2) The env parameter could be a gymEnv, VecEnv or DummyVecEnv object, see comments
    above. This parameter is not needed to load a trained SB3 model.

    3) To further customize the network architecture, one specifies the following
    parameters: net_arch, activation_fn, features_extractor_class,
    features_extractor_kwargs, and share_features_extractor.

    4) The optimizer parameters are optimizer_class and optimizer_kwargs. By
    default, the optimizer class is torch.optim.Adam with eps = 1e-5.

    5) The training parameters are lr_schedule, optimizer_class, optimizer_kwargs,
    and normalize_images.

    II - TD3Policy Neural nets:
    ---------------------------
    1) The TD3 constructor calls TD3Policy._build() to initialize 4 neural nets,
    each of which is an attribute of the class TD3Policy. The helper functions used
    in _build() are make_features_extractor(), make_actor(), and make_critic().
    As opposed to the last 2 functions, make_features_extractor() is inherited from
    BaseModel (see line 114 of the common.policies submodule).
```

```
    2) Actor networks: The TD3Policy.actor and TD3Policy.actor_target attributes are
    instances of the Actor[BasePolicy] class from the TD3.policies submodule.
    In detail, the constructor of Actor calls common.torch_layers.create_mlp() to
    combine the feature extractor layers and the activation function, then compiles
    this module with:
        (Line 60) self.mu = nn.Sequential(*actor_net)
    The function TD3Policy.actor.forward() wraps mu.forward().

    3) Critic: The TD3Policy.critic and TD3Policy.critic_target attributes are
    instances of the ContinuousCritic[BaseModel] class from the common.policies submodule.
    This class allows to have several critic networks, and to decide whether or not
    to share the feature extraction layers with the actor.
    The main attribute of ContinuousCritic is the list of q_networks, which ss for the actor,
    are created using common.torch_layers.create_mlp() and by setting:
        (Line 875) q_net = nn.Sequential(*q_net)
    for each critic in (see common.policies).
    The ContinuousCritic.forward() function returns a tuple containing the output
    of each of these networks, while the ContinuousCritic.q1_forward() function
    returns only the output of the first entry in the ContinuousCritic.q_networks list.

    4) Aliases: The TD3 model constructor calls a helper method called _setup_model(),
    which initializes the TD3.policy object as a TD3Policy instance, and then calls
    _make_aliases() that initialized the TD3.actor, TD3.actor_target, TD3.critic, and
    TD3.critic_target attributes. These are equated to the attributes of the same name
    of TD3.policy, and used in the TD3.train() method discussed below.

'''

# Train TD3 model
model.learn(total_timesteps=25000)
'''
    1) OffPolicyAlgorithm.learn() as an abstract method calls:
        a) OffPolicyAlgorithm.collect_rollouts()
        b) TD3.train()

    2) For (a) above, collect_rollouts() calls the OffPolicyAlgorithm.predict()
    method. The details on the predict() method are given at the deployment
    step below.

    3) For (b) above, TD3.train() implements the TD3 algorithm. It uses all the
    neural networks built by the constructor with the following
    calls in the main loop:
        (Line 168) next_actions = (self.actor_target(replay_data.next_observations)
                                   + noise).clamp(-1, 1)
        (Line 171) next_q_values = th.cat(self.critic_target(replay_data.next_observations,
                                    next_actions), dim=1)
        (Line 176)  current_q_values = self.critic(replay_data.observations,
                                        replay_data.actions)
        (Line 190) actor_loss = -self.critic.q1_forward(replay_data.observations,
                                self.actor(replay_data.observations)).mean()
    All of these are calls to the forward() method of the corresponding network.
    (The line numbers refer to the TD3.td3 submodule.)

    4) The critic loss computation and its gradient step are performed in
    lines 178 to 185 of the TD3.td3 submodule.

    5) The actor loss computation and its gradient step are performed in
    lines 189 to 196 of the Td3.td3 submodule.
'''

# Deploy model
obs = env.reset()
while True:
    # Predict actions
    action, _states = model.predict(obs)
    '''
        1) At level of abstract classes, the above is BaseAlgorithm.predict(),
        which wraps BaseAlgorithm.policy.predict(), which in turn wraps the abstract
        method BasePolicy._predict() of the policy class.
```

```
        2) In the case of TD3: TD3Policy._predict() wraps the method:
            TD3Policy.actor.forward()
        (which is the same here as TD3Policy.forward().)

        3) The above wraps the method :
            Actor.mu.forward()
    '''
    # One time step
    obs, rewards, dones, info = env.step(action)
```

Our summary of SB3 policies started with the case of deterministic policies, whose architecture is less involved than that of stochastic policies. In particular, the example above explains SB3's implementations of both TD3 and DDPG, and also gives an idea of how the Q-network is implemented for DDQN (see: https://github.com/DLR-RM/stable-baselines3/tree/master/stable_baselines3/dqn).

d) Models III - Neural nets for stochastic policies

d.1 - The *ActorCriticPolicy* class

In this section we discuss the second design pattern used by Stable Baselines 3 for policies. We will mostly focus on the stochastic policies of on-policy algorithms such as A2C and PPO, by looking into the details of the *ActorCriticPolicy* class in the *common.policies* submodule.

In contrast to the previous section, where the actor and critic networks were represented by distinct classes, the *ActorCriticPolicy* class uses several common layers between the actor and critic networks. In more detail:

- On top of the *features_extractor* layers, *ActorCriticPolicy* has a *mlp_extractor* attribute that computes auxilary tensors *latent_vf* and *latent_pi*. These layers are shared by the attributes *value_net*, *action_net* and *log_std*.
- The *features_extractor* and *mlp_extractor* layers are specified by the *features_extractor_class*, *features_extractor_kwargs*,*net_arch*, and *activation_fn* parameters of the constructor.
- The stochastic policy is encoded by the *action_dist* attribute, which is a *Distribution* object from the *common.distributions* submodule.
- The *action_net* attribute represents the neural net computing the mean of *action_dist*, while *log_std* is the net computing the log-standard deviation of this distribution.

Section 2.(d.2) gives more details on the *Distribution* class of which *action_dist* is an instance. This is particularly important here, as it solves the additional layer of complexity coming from encoding a probability distribution and appropriately evaluating random actions during training.

When instantiating an *ActorCriticPolicy* object, the attributes discussed above are initalized in the following order: 1) *ActorCriticPolicy.**init**()* first initializes *features_extractor*. 2) *action_dist* is initalized using *common.distributions.make_proba_distribution()*, according to the class of the action space (see next subsection for more details). 3) The constructor calls the *_build()* method that calls *_build_mlp_extractor()*, which in turn initializes *mlp_extractor* (a *MlpExtractor* object, implemented in *common.toch_layers*). 4) Within *_build()*, the attributes *action_net* and *log_std* are initialized using *action_dist.proba_distribution_net()*, depending on the distribution class used. 5) *_build()* initializes *value_net* as a *nn.Linear* object with one output feature.

The precise outputs *value_net*, *action_net*, and *log_std* networks are specified in the *ActorCriticPolicy.forward()* method, reproduced in the next cell with comments.

In [ ]:
```
    def forward(self, obs: th.Tensor, deterministic: bool = False) -> Tuple[th.Tensor, th.Tensor, th.Ten
        """
        Forward pass in all the networks (actor and critic)
        :param obs: Observation
        :param deterministic: Whether to sample or use deterministic actions
```

```
        :return: action, value and log probability of the action
        """
        ### This part calls the common.torch_layers submodule
        # Preprocess the observation if needed
        features = self.extract_features(obs)
        '''
            This is a method inherited from BasePolicy that
            performs the following sequence of calls:
            BasePolicy.extract_features()->BaseFeaturesExtractor.features_extractor.forward().
        '''

        # Compute latent policy and latent value function
        latent_pi, latent_vf = self.mlp_extractor(features)
        '''
            1) Here, "latent" refers to the shared features
            by the actor and critic networks, which are
            computed by the feature extraction layers.
            2) The line above is in fact a call to
            MlpExtractor.forward(), which returns the outputs
            of:
            - MlpExtractor.policy_net.forward()
            - MlpExtractor.value_net.forward()
        '''

        # Evaluate the values for the given observations
        values = self.value_net(latent_vf)
        '''
            This line calls ActorCriticPolicy.value_net, whic is
            initialized in ActorCriticPolicy._build() as:
               self.value_net = nn.Linear(self.mlp_extractor.latent_dim_vf, 1)
        '''

        ### This part calls the common.distributions submodule
        # Get action_distribution from latent features
        distribution = self._get_action_dist_from_latent(latent_pi)
        '''
            1) This line involves three attributes:
            - ActorCriticPolicy.action_dist
            - ActorCriticPolicy.action_net
            - ActorCriticPolicy.log_std
            and wraps the function:
                ActorCriticPolicy.action_dist.proba_distribution(),
            which returns the latent probability distribution.
            2) For continuous actions distributed according to a Gaussian,
            action_dist.proba_distribution_net() sets up the mean and std
            via:
                action_net =  nn.Linear(latent_dim, self.action_dim)
                log_std = nn.Parameter(th.ones(self.action_dim) * log_std_init,
                                       requires_grad=True)
                (see common.distributions submodule).
        '''

        # Compute actions from the action distribution and their log probabilities
        actions = distribution.get_actions(deterministic=deterministic)
        '''
            This calls the Distribution.get_actions() method that
            calls one of the following 2 abstract methods:
            - sample() if deterministic = False
            - mode() if deterministic = True.
        '''
        log_prob = distribution.log_prob(actions)
        actions = actions.reshape((-1,) + self.action_space.shape)


        # return output
        return actions, values, log_prob
```

One of the confusing points about *ActorCriticPolicy* are the various methods used to compute the outputs of the neural nets involved. These are as follows:

- *forward()*: Forward pass in all networks: feature extractor, MLP extractor, value_net, and action_net. Used when collecting rollouts (see ***OnPolicyAlgorithm.collect_rollouts()***).
- *evaluate_actions()*: Given an observation and actions, get corresponding values and log probabilities. Used during training (see ***PPO.train()*** for a concrete example).
- *extract_features()*: Inherited from *BaseModel*. Preprocess observation if needed and call *forward()* method of *feature_extractor*.
- *get_distribution()*: Get current policy distribution corresponding to observation.
- *_get_action_dist_from_latent()*: Returns the action distribution from the latent values of features (*latent_pi* output by *mlp_extractor*)
- *_predict()*: Given an observation, get action from the policy distribution. Wraps the instruction self.get_distribution(observation).get_actions(deterministic=deterministic), and is wrapped by *BasePolicy.predict()*. Used when deploying a trained model for instance.
- *predict_values()*: Get estimated Q-value according to current policy, and given an observation. (See ***OnPolicyAlgorithm.collect_rollouts()***)

d.2 - The *common.distributions* submodule

As one would expect, the *distributions* submodule is central to the implementation of stochastic policies:

https://github.com/DLR-RM/stable-baselines3/blob/master/stable_baselines3/common/torch_layers.py

It might be important to note that *common.distributions* imports PyTorch's distributions submodule *torch.distributions*, which is itself inspired by TensorFlow's distributions package. Some useful links:

- Torch distributions docs: https://pytorch.org/docs/stable/distributions.html.
- TensorFlow distributions paper: https://arxiv.org/pdf/1711.10604.pdf.
- PyTorch abstract *Ditribution* class: https://github.com/pytorch/pytorch/blob/master/torch/distributions/normal.py.
- PyTorch *Normal* class: https://github.com/pytorch/pytorch/blob/master/torch/distributions/normal.py.
- PyTorch Kullback-Liebler submodule: https://github.com/pytorch/pytorch/blob/master/torch/distributions/kl.py

In particular, SB3's distributions submodule imports the *Bernoulli*, *Categorical*, and *Normal* classes from PyTorch.

For reference, this submodule contains the following:

Classes:

- *Distribution(ABC)*: Base abstract class for SB3 distributions. Has one attribute *distribution*, and declares most of the abstract **FINISH THIS**.
- *DiagGaussianDistribution(Distribution)*: To model Gaussian distributions with diagonal covariance matrix on continuous action spaces. Superclass of *SquashedDiagGaussianDistribution*.
- *CategoricalDistribution(Distribution)*: To model discrete distributions on discrete action spaces. Superclass of *MultiCategoricalDisrtibution*.
- *BernoulliDistribution(Distribution)*: Bernoulli distribution for MultiBinary action spaces.
- *StateDependentNoiseDistribution(Distribution)*: Used for state-dependent noise exploration, as used elsewhere in SB3.
- *TanhBijector*: Bijective transformation of a probability distribution. Used in implementation of SAC.

Helper functions:

- *sum_independent_dims()*: Function summing log probabilities when computing the entropy of a Gaussian.
- *make_proba_distribution(action_space, use_sde, dist_kwargs)*: Takes a *gym.spaces.Space* argument and distribution arguments to return a *Distribution* instance adapted to the action space class. Notably, if the action space class is

gym.spaces.Box, then the output is a *DiagGaussianDistribution* object of the appropriate action space dimension. If the action space if of *gym.spaces.Discrete* class, then the output is of *CategoricalDistribution* class.

- *kl_divergence(dist_true, dist_pred)*: Wrapper for PyTorch's *torch.distributions.kl_divergence(P,Q)*. Inputs are SB3 distributions, and output is a *torch.Tensor*. **Note:** Comment on how SB3's distributions wrap PyTorch's.

Finish this. Discuss sampling and mode().

d.3 - Example: The *PPO.ppo* and *PPO.policy* submodules

Finish this...

In [ ]:
```
# Create the environment
env = make_vec_env("CartPole-v1", n_envs=1)

# Instantiate TD3 algorithm object
model = PPO(policy = "MlpPolicy", env=env, verbose=1)
'''
    1) Differences between "MlpPolicy", "CnnPolicy", and "MultiInputPolicy" for PPO

    2) Action distribution

    3) Actor network:

    4) Critic:

    ?) Aliases?

'''

# Train PPO model
model.learn(total_timesteps=25000)
'''
    1) OnPolicyAlgorithm.learn() as an abstract method calls:
        a) OnPolicyAlgorithm.collect_rollouts()
        b) PPO.train()

    2) For (a) above, collect_rollouts() calls
        (Line 166) actions, values, log_probs = self.policy(obs_tensor)
        (Line 210) values = self.policy.predict_values(obs_as_tensor(new_obs, self.device))
        (in common.OnPolicyAlgorithm submodule)

    3) For (b) above, PPO.train() implements the PPO algorithm.
        (Line 208) values, log_prob, entropy = self.policy.evaluate_actions(rollout_data.observations, a

    4) Computation of loss functions and gradient step.
'''

# Deploy model
obs = env.reset()
while True:
    # Predict actions
    action, _states = model.predict(obs)
    '''
        This line wraps the ActorCriticAlgorithm._predict() method, which
        returns:
            (Line 613) ActorCriticAlgorithm.get_distribution(observation).get_actions(deterministic=dete
            (see common.policies and common.distributions)
    '''
    # One time step
    obs, rewards, dones, info = env.step(action)
```

d) Loggers

Note (22/09/19): This is about the information logged during model training. In *BaseAlgorithm*, can find the methods *logger()* and *set_logger()* for instance See: https://stable-baselines3.readthedocs.io/en/master/common/logger.html#logger

e) TensorBoard and Callbacks

Note (22/09/19): This is mostly about Tensorboard integration. See:

- https://stable-baselines3.readthedocs.io/en/master/guide/callbacks.html
- https://stable-baselines3.readthedocs.io/en/master/guide/tensorboard.html

---

3 - Portfolio optimization environment

a) FinRL's updated environment

This class definition is from: https://github.com/AI4Finance-Foundation/FinRL/blob/master/tutorials/2-Advance/FinRL_PortfolioAllocation_Explainable_DRL.ipynb. "Updated" here refers to the fact that the implementation below isn't the one in the library files at the time of writing (Sept. 2022).

Below is the code with my comments.

In [ ]:
```
########################################
##### STOCK PORTFOLIO ENVIRONMENT #####
########################################

'''
### Imports
# Other than the usual: np, pd, gym, plt, matplotlib.

from gym.utils import seeding
from gym import spaces
from stable_baselines3.common.vec_env import DummyVecEnv
matplotlib.use('Agg')

'''

class StockPortfolioEnv(gym.Env):
    """A portfolio allocation environment for OpenAI gym

    Attributes
    ----------
        df: DataFrame
            input data
        stock_dim : int
            number of unique stocks
        hmax : int
            maximum number of shares to trade
        initial_amount : int
            start money
        transaction_cost_pct: float
            transaction cost percentage per trade
        reward_scaling: float
            scaling factor for reward, good for training
        state_space: int
            the dimension of input features
        action_space: int
            equals stock dimension
        tech_indicator_list: list
```

```
        a list of technical indicator names
    turbulence_threshold: int
        a threshold to control risk aversion
    day: int
        an increment number to control date

    ### Addendas (22/09/28, AJ Zerouali)
    # Several other attributes are used below
    reward: float
        Current portfolio value. Only computed in step().
    lookback: int
        No. of lookback days for computation of covariances of asset returns
    data: pd.DataFrame
        Data on self.day, i.e. stock prices, tech. indicators, cov and returns
    terminal: bool
        Attribute signaling last day of input data df
    portfolio_return_memory: list
        List of  what?
    portfolio_value: ?
    covs: np.ndarray, shape = (stock_dim, stock_dim)
        Covariance of portfolio asset returns on self.day.
    date_memory: list of str
        List of dates corresponding to self.day attribute.


Methods
-------
_sell_stock() (NOT IMPLEMENTED)
    perform sell action based on the sign of the action
_buy_stock() (NOT IMPLEMENTED)
    perform buy action based on the sign of the action
step()
    at each step the agent will return actions, then
    we will calculate the reward, and return the next observation.
reset()
    reset the environment
render()
    use render to return other functions
save_asset_memory()
    return history of portfolio returns over dates in df
save_action_memory()
    return history of portfolio weights


"""
metadata = {'render.modes': ['human']}

####################
### Constructor ###
####################
def __init__(self,
             df,
             stock_dim,
             hmax,
             initial_amount,
             transaction_cost_pct,
             reward_scaling,
             state_space,
             action_space,
             tech_indicator_list,
             turbulence_threshold=None,
             lookback=252,
             day = 0):

    self.day = day
    self.lookback=lookback # Clarify
    self.df = df
    self.stock_dim = stock_dim
    self.hmax = hmax # Max no. of stocks to trade
    self.initial_amount = initial_amount
```

```
        self.transaction_cost_pct =transaction_cost_pct
        self.reward_scaling = reward_scaling # Clarify
        self.state_space = state_space
        self.action_space = action_space
        self.tech_indicator_list = tech_indicator_list

        '''
         Although they normalize the action and state space elements,
         the portfolio weights (which are supposed to be actions) are
         not included in the states.
        '''
        # action_space normalization and shape is self.stock_dim
        self.action_space = spaces.Box(low = 0, high = 1,shape = (self.action_space,)) # Clarify
        self.observation_space = spaces.Box(low=-np.inf, high=np.inf, shape = (self.state_space+len(self

        # load data from a pandas dataframe
        self.data = self.df.loc[self.day,:] # Clarify. Do not confuse with self.df.
        self.covs = self.data['cov_list'].values[0]
        self.state =  np.append(np.array(self.covs), [self.data[tech].values.tolist() for tech in self.t
        self.terminal = False
        self.turbulence_threshold = turbulence_threshold
        # initalize state: inital portfolio return + individual stock return + individual weights
        self.portfolio_value = self.initial_amount

        # memorize portfolio value each step
        self.asset_memory = [self.initial_amount]
        self.portfolio_return_memory = [0]
        self.actions_memory=[[1/self.stock_dim]*self.stock_dim]
        self.date_memory=[self.data.date.unique()[0]] # Clarify

        '''
         Notes:
         - There are no attributes for the state/action spaces
           dimensions.
         - They call the number of stocks the stock dimension.
         - Actions correspond to outputs of neural nets. Portfolio
           weights are softmax outputs of actions.
         - States consist of covariance matrices of returns (computed
           over lookback period) and chosen technical indicators.
         - The number of days in one episode should be added as
           an attribute. The method len(df.index.unique()) used
           to compute it is slow.
         - There is a self.data attribute whose function is not
           explained, but that appears in main computations.
           It's a dataframe of with columns given by:
           date-OHLCV-tech. ind.-cov.- asset returns.
         - The self.day attribute is a date index that is incremented
           with the step() method during transitions.
         - Attibutes with the '_memory' suffix are daily lists.
        '''

    ############################
    ### Gym's reset() method ###
    ############################
    def reset(self):
        self.asset_memory = [self.initial_amount]
        self.day = 0
        self.data = self.df.loc[self.day,:]
        # load states
        self.covs = self.data['cov_list'].values[0]
        self.state =  np.append(np.array(self.covs), [self.data[tech].values.tolist() for tech in self.t
        self.portfolio_value = self.initial_amount
        #self.cost = 0
        #self.trades = 0
        self.terminal = False
        self.portfolio_return_memory = [0]
        self.actions_memory=[[1/self.stock_dim]*self.stock_dim]
        self.date_memory=[self.data.date.unique()[0]]
        return self.state
```

```
    def render(self, mode='human'):
        return self.state

    ############################
    ### Gym's step() method ###
    ############################
    def step(self, actions):
        # Update self.terminal attribute
        self.terminal = self.day >= len(self.df.index.unique())-1
        '''
          They using len(self.df.index.unique()) for the number of days
          in the dataset, because there's no attribute storing that value.
          The length of the data set equals the number of stocks times
          the number of days in the df input (after pre-processing).
        '''

        ## If currently at terminal state ##
        if self.terminal:
            df = pd.DataFrame(self.portfolio_return_memory) # Clarify
            df.columns = ['daily_return']
            plt.plot(df.daily_return.cumsum(),'r')
            plt.savefig('results/cumulative_reward.png')
            plt.close()

            plt.plot(self.portfolio_return_memory,'r')
            plt.savefig('results/rewards.png')
            plt.close()

            print("==================================")
            print("begin_total_asset:{}".format(self.asset_memory[0]))
            print("end_total_asset:{}".format(self.portfolio_value))

            df_daily_return = pd.DataFrame(self.portfolio_return_memory)
            df_daily_return.columns = ['daily_return']

            # Compute the Sharpe ratio at last step of episode.
            if df_daily_return['daily_return'].std() !=0:
                sharpe = (252**0.5)*df_daily_return['daily_return'].mean()/ \
                       df_daily_return['daily_return'].std()
                print("Sharpe: ",sharpe)
            print("==================================")

            return self.state, self.reward, self.terminal,{}

        ## If not at end of episode ##
        else:

            # Compute current weights from actions
            weights = self.softmax_normalization(actions)

            # Store data
            last_day_memory = self.data

            #Load next state
            self.day += 1
            self.data = self.df.loc[self.day,:]
            self.covs = self.data['cov_list'].values[0]
            self.state =  np.append(np.array(self.covs), [self.data[tech].values.tolist() for tech in se

            # Compute potfolio return and log return
            portfolio_return = sum(((self.data.close.values / last_day_memory.close.values)-1)*weights)
            log_portfolio_return = np.log(sum((self.data.close.values / last_day_memory.close.values)*we

            # Compute reward (portfolio value)
            ### (Reward is the new portfolio value or end portfolo value)
            new_portfolio_value = self.portfolio_value*(1+portfolio_return)
            self.portfolio_value = new_portfolio_value
            self.reward = new_portfolio_value

            # Update memory attributes
```

```
            self.actions_memory.append(weights)
            self.portfolio_return_memory.append(portfolio_return)
            self.date_memory.append(self.data.date.unique()[0])
            self.asset_memory.append(new_portfolio_value)

            return self.state, self.reward, self.terminal, {}


    ###############
    ### Softmax ###
    ###############
    ### 22/09/28, AJ Zerouali
    '''
      This is to compute the portfolio weights.
      The actions in this environment correspond
      to portfolio weights, but the output of the policy
      networks used is not necessarily normalized.
    '''
    def softmax_normalization(self, actions):
        numerator = np.exp(actions)
        denominator = np.sum(np.exp(actions))
        softmax_output = numerator/denominator
        return softmax_output

    def save_asset_memory(self):
        """
            Makes an 'account_value' dataframe with dates and daily portfolio values
            Should actually be called get_pf_value_hist.
        """
        date_list = self.date_memory
        portfolio_return = self.portfolio_return_memory
        #print(len(date_list))
        #print(len(asset_list))
        df_account_value = pd.DataFrame({'date':date_list,'daily_return':portfolio_return})
        return df_account_value

    def save_action_memory(self):
        """
            Makes a dataframe with dates and daily portfolio weights.
            Should actually be called get_pf_weights_hist.
        """
        # date and close price length must match actions length
        date_list = self.date_memory
        df_date = pd.DataFrame(date_list)
        df_date.columns = ['date']

        action_list = self.actions_memory
        df_actions = pd.DataFrame(action_list)
        df_actions.columns = self.data.tic.values
        df_actions.index = df_date.date
        #df_actions = pd.DataFrame({'date':date_list,'actions':action_list})
        return df_actions

    ####################
    ### Seed method ###
    ####################
    def _seed(self, seed=None):
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    ########################
    ### SB3 environment ###
    ########################
    def get_sb_env(self):
        e = DummyVecEnv([lambda: self])
        obs = e.reset()
        return e, obs
```

Remark: From the *StockPortfolioEnv* class above, we see that there's a method *get_sb_env()* which instantiates a *DummyVecEnv* object (in the submodule *stable_baselines3.common.vec_env*). In brief, since we use SB3

agents/models, we convert the FinRL environment to an SB3 one using *get_sb_env()*. We emphasize that this last method only converts the FinRL environment to an SB3 one that is reset to the first date of the (pre-processed) dataset.

b) Attributes

We give a brief description of the attributes of *StockPortfolioEnv*, grouped by category:

Data management attributes:

- *df*: The input *pd.DataFrame* with pre-processed data. The columns consist of 'date', OHLCV columns, technical indicator columns, 'tic' for ticker, 'cov_list' and 'return_list'. Furthermore:
  - The indexes of the *df* dataframe correspond to the day index (see *day* below), while the 'date' column contains the dates as strings in the 'yyyy-mm-dd' format.
  - The 'return_list' column stores the daily asset returns for the portfolio assets over the previous n=*lookback* days. On a given date, all assets in the portfolio have the same list of returns.
  - The 'cov_list' column stores the covariance matrix of the asset returns over the previous n=*lookback* days. On a given date, all assets in the portfolio have the same covariance array.
- *lookback*: Number of lookback days used for the computation of the arrays in the 'cov_list' and 'return_list' columns. This parameter is specified at the data processing step.
- *day*: Day index in the df dataset, or "current time step" of the episode/trading period. The methods of *StockPortfolioEnv* use this *int* for the computations rather than the dates. *day* is set to 0 in *reset()* and incremented in the *step()* method.
- *stock_dim*: Number of (unique) stocks in the 'tic' column of *df*, i.e. no. of portfolio assets. Used for the contruction of the state space and action space, as well as their related attributes.

State attributes:

- *initial_amount*: Initial cash in hand for the portfolio modelled by the environment.
- *data*: Sub-dataframe of *df* with index t=*day*. Contains the prices, tech. indicators and covariances/asst. rets. on a fixed day, for all tickers. Updated in the *step()* method.
- *state_space*: Described as the no. of input features, but used as the no. of portfolio stocks.
- *observation_space*: Represents the state space of the MDP. Initialized in the constructor as a *gym.spaces.Box* object, of shape = (n_stocks+n_tech_ind, n_stocks) and no upper/lower bounds. Here, n_tech_ind is the length of the *tech_indicator_list* list below.
- *tech_indicator_list*: List of technical indicators used for the construction of the MDP states. It only needs to be a subset of the technical indicators in the input dataframe *df*.
- *terminal*: Boolean indicator whether the current state of the environment is terminal, i.e. whether we have reached the last *day* index in *df*. True when *day* is grater or equals *len(self.df.index.unique())-1* (see step method).
- *covs*: Stores the covariance matrix of asset returns at time step t=*day*, i.e. the entry of the 'cov_list' column of *data*.
- ***state*** The state is obtained by concatenating the covariance matrix *covs* and the technical indicators specified in the *tech_indicator_list* attribute. The states are encoded as *float64 np.ndarray*'s, with *shape*= *(n_stocks+n_tech_indicators, n_stocks)*.

- *portfolio_value*: The portfolio value at time-step *day*, computed in terms of the portfolio return. In the *step()* method, the portfolio return is computed as the weighted sum of asset returns over the close prices in *data*.

Action attributes:

- *action_space*: As a parameter for the constructor, this is the number of portfolio stocks. The constructor then assigns a *gym.spaces.Box* object to this attribute, with shape (n_stocks,) and no upper/lower bounds.
- *hmax*: Maximum number of shares to trade. Not used in this implementation.

Reward attributes:

- *reward*: This attribute is not initialized in the constructor. The only part where is appears is in the *step()* method.
- *transaction_cost_pct*: Transaction cost per trade. Not used in the present implementation.
- *reward_scaling*: Scaling factor for rewards. Not used in this implementation.
- *turbulence_threshold*: Risk aversion threshold related to the turbulence technical indicator. Not used in this implementation.

Portfolio history attributes:

All of the following are lists updated when the *step()* method is executed.

- *date_memory*: A list of dates in *str* format. Initialized with the date in *df* corresponding to *day*=0, and then filled with the contents of the 'date' column as the *day* index is incremented.
- *actions_memory*: A list of daily portfolio weights (the latter are also encoded as a list). Stores the portfolio weights history, and can be accessed using the *save_action_memory()* method.
- *portfolio_return_memory*: History of daily portfolio returns, list of floats.
- *asset_memory*: History of daily portfolio values, list of floats. This is the return of the *save_asset_memory()* method.

Comments:

1) For a more realistic implementation of a portfolio optimization environment, there should also be:

- An array of integers storing the number of shares for each of the porfolio assets.
- A cash-in-hand attribute.
- The state of the portfolio could possibly store the number of shares, the cash in hand, as well as the portfolio value.
- The portfolio weights should be converted to numbers of shares to buy or sell at a given time step.

2) In terms of software design:

- Instead of *stock_dim*, I would've called this attribute *N_stocks*, and I would have added *N_days* for the number of days in the input dataset. The implementation accessed this quantity using *len(self.df.date.unique())*, which is slow given that Pandas needs to search through a 15MB dataframe.
- The format of the input dataframe *df* is far from optimal. Using the sub-dataframe *data* at each time step could be inadequate for deploying models to paper-trading over 30min rebalancing of the portfolio for instance.

We additionally give some related comments at the end of the next subsection.

c) The *step()* and *reset()* methods

In brief, here are the main things to record from the transitions obtained using this implementation of *step()*:

(1) When the environment is not in a terminal state:

- FinRL encode portfolio weights as the softmax of the actions obtained from the agents.
- The reward function used is **not** the portfolio return, it's the entire **portfolio value**. This raises a conceptual question regarding the meaning of the expected cumulative returns obtained from these rewards.
- We note that the *transaction_cost_pct* is not used anywhere in this implementation.

Other than that, the overall methodology is standard.

(2) When the environment is at a terminal state:

- The Sharpe ratio is computed over the portfolio values list *asset_memory*.
- The cumulative returns and the daily returns are plotted and saved to png files.
- The reward is not updated in this implementation.

The *reset()* method executes essentially the same instructions, with the appropriate instructions for the portfolio value and return on *day*=0.

Now we comment on what is missing from the *step()* method, and more generally from *StockPortfolioEnv*:

- There should be a *get_N_shares()* method that converts the portfolio weights to signed integers representing the number of shares to buy or sell for each asset. This pre-supposes the existence of a *N_stock_shares* attribute. The *step()* method should call *get_N_shares()* right after the computation of the portfolio weights from the actions determined by the agent.

- There should be an *execute_trades()* method that buys and sells asset shares to realize the output of *get_N_shares()*. The tricky part here is selling shares before buying new ones, and the order in which each trade should be executed (sort by return?). This seems like a mathematical finance question of its own.

- There should be a *rebalance_portfolio()* method. This one would be called by *step()* before the state and hitory lists updates, and would in turn call the execute trades method. It could possibly also be the method that sends orders to a brokerage API during paper-trading, and would potentially require the implementation of additional *buy()* and *sell()* methods.

- Instead of computing the portfolio weights in *step()* using *StockPortfolioEnv.softmax_normalization()*, it might be more optimal and accurate to use:

  ```
  weights = torch.nn.functional.softmax(torch.tensor(action)).numpy().
  ```

---

4 - Deep RL agents

The purpose of this section is to give a description of how a FinRL agent is implemented and used. In many respects however, this implementation is impractical for several reasons.

Recall that in FinRL, creating a deep RL agent, training it, and running a backtest is executed as follows:

In [ ]:
```
# Create gym/SB3 training environments
gym_env_train = StockTradingEnv(df = data_train, **env_kwargs)
sb3_env_train, _ = gym_env_train.get_sb_env()

# (a) - Inst. agent and model
```

```
    agent = DRLAgent(env = env_train)
    model_ = agent.get_model("model_name")

    # (b) -Train agent
    trained_model = agent.train_model(model=model_,
                                      tb_log_name='model_name',
                                      total_timesteps=50000)

    # (c) - Run a backtest
    gym_env_test = StockPortfolioEnv(df = data_test, **env_kwargs)
    sb3_env_test, _ = gym_env_test.get_sb_env()
    df_test_daily_return, df_test_actions = DRLAgent.DRL_prediction(model=trained_model,
                                                                    environment = sb3_env_test)
    # (d) - Plot results and get performance stats
    backtest_plot(df_daily_return_ppo,
             baseline_ticker = 'Baseline_ticker',
             baseline_start = df_test_daily_return.loc[0,'date'],
             baseline_end = df_test_daily_return.loc[len(df_test_daily_return)-1,'date'])
```

The next subsections discuss each of blocks (a)-(d) above.

a) The DRLAgent class

This class is imported via:

In [ ]:
```
from finrl.agents.stablebaselines3.models import DRLAgent
```

and is implemented here:

https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/agents/stablebaselines3/models.py,

where one finds the description:

In [ ]:
```
'''
            Attributes
            ----------
                env: gym environment class
                    user-defined class
            Methods
            -------
                get_model()
                    setup DRL algorithms
                train_model()
                    train DRL algorithms in a train dataset
                    and output the trained model
                DRL_prediction()
                    make a prediction in a test dataset and get results
'''
```

Looking at the file linked above, we note that:

- The constructor of DRLAgent assigns only the environment as an attribute. This is an SB3 environment passed-on to the constructor of the SB3 algorithm used. As mentioned in the sections on Stable Baselines 3 algorithms, the environment is required for the training, but the argument can be *None* when **loading** a trained agent.
- The file models.py imports one of the following models from sb3: A2C, DDPG, PPO or SAC. The point here is that *get_model()* returns one of the algorithms implemented in stable_baselines3, but FinRL does not support all SB3 agents because of how the actions are defined in this library.
- Another point to discuss below is the policy argument of *get_model()*. By default, it's an "MlpPolicy" object from SB3.
- The *train_model(self, model, tb_log_name, total_timesteps)* method is essentially a wrapper for the *model.train()* method of SB3 models.

- The method *DRL_prediction(model, env, deterministic)* first creates a test environment, and then loops over the contents of the attribute *env.df* of the environment and calls . **Remark:** Understanding this method hinges on understanding FinRL's environments.
- Throughout the implementation, one finds several instructions for the management of tensorboard callbacks.

Now let's comment on some downsides:

- The DRLAgent class **does not store** the SB3 algorithm as an attribute. There is only one attribute in this class, which is the environment. I don't understand this design choice, and I absolutely do not like it. I will re-write/modify this class at some point.
- From the implementation, it seems like the DRLAgent is just a book-keeping object that creates a model on the side. It doesn't even have a save method.
- The *DRL_prediction()* method is also clunky and unclear. Why do they use a test environment?
- There are several tautological instructions in this implementation. See comments in code here and next subsections.

For future reference, here is the original implementation of DRLAgent:

In [ ]:
```
'''
##### Imports #####

from stable_baselines3 import A2C
from stable_baselines3 import DDPG
from stable_baselines3 import PPO
from stable_baselines3 import SAC
from stable_baselines3 import TD3
from stable_baselines3.common.callbacks import BaseCallback
from stable_baselines3.common.noise import NormalActionNoise
from stable_baselines3.common.noise import OrnsteinUhlenbeckActionNoise
from stable_baselines3.common.vec_env import DummyVecEnv

from finrl import config
from finrl.meta.env_stock_trading.env_stocktrading import StockTradingEnv
from finrl.meta.preprocessor.preprocessors import data_split

'''
MODELS = {"a2c": A2C, "ddpg": DDPG, "td3": TD3, "sac": SAC, "ppo": PPO}

MODEL_KWARGS = {x: config.__dict__[f"{x.upper()}_PARAMS"] for x in MODELS.keys()}

NOISE = {
    "normal": NormalActionNoise,
    "ornstein_uhlenbeck": OrnsteinUhlenbeckActionNoise,
}


class TensorboardCallback(BaseCallback):
    """
    Custom callback for plotting additional values in tensorboard.
    """

    def __init__(self, verbose=0):
        super().__init__(verbose)

    def _on_step(self) -> bool:
        try:
            self.logger.record(key="train/reward", value=self.locals["rewards"][0])
        except BaseException:
            self.logger.record(key="train/reward", value=self.locals["reward"][0])
        return True


class DRLAgent:
    """Provides implementations for DRL algorithms
```

```
    Attributes
    ----------
        env: gym environment class
            user-defined class

    Methods
    -------
        get_model()
            setup DRL algorithms
        train_model()
            train DRL algorithms in a train dataset
            and output the trained model
        DRL_prediction()
            make a prediction in a test dataset and get results
    """

    def __init__(self, env):
        self.env = env

    ######################
    ### Build SB3 model ###
    ######################
    def get_model(
        self,
        model_name,
        policy="MlpPolicy",
        policy_kwargs=None,
        model_kwargs=None,
        verbose=1,
        seed=None,
        tensorboard_log=None,
    ):
        if model_name not in MODELS:
            raise NotImplementedError("NotImplementedError")

        if model_kwargs is None:
            model_kwargs = MODEL_KWARGS[model_name]

        if "action_noise" in model_kwargs:
            n_actions = self.env.action_space.shape[-1]
            model_kwargs["action_noise"] = NOISE[model_kwargs["action_noise"]](
                mean=np.zeros(n_actions), sigma=0.1 * np.ones(n_actions)
            )
        print(model_kwargs)
        return MODELS[model_name](
            policy=policy,
            env=self.env,
            tensorboard_log=tensorboard_log,
            verbose=verbose,
            policy_kwargs=policy_kwargs,
            seed=seed,
            **model_kwargs,
        )

    def train_model(self, model, tb_log_name, total_timesteps=5000):
        model = model.learn(
            total_timesteps=total_timesteps,
            tb_log_name=tb_log_name,
            callback=TensorboardCallback(),
        )
        return model

    ######################
    ### Predict method ###
    ######################
    @staticmethod
    def DRL_prediction(model, environment, deterministic=True):
        test_env, test_obs = environment.get_sb_env() # Clarify what this does
        """make a prediction"""
        account_memory = []
```

```
        actions_memory = []
        #         state_memory=[] #add memory pool to store states
        test_env.reset() # Clarify
        for i in range(len(environment.df.index.unique())): # Fix this
            action, _states = model.predict(test_obs, deterministic=deterministic)
            # account_memory = test_env.env_method(method_name="save_asset_memory")
            # actions_memory = test_env.env_method(method_name="save_action_memory")
            test_obs, rewards, dones, info = test_env.step(action)
            if i == (len(environment.df.index.unique()) - 2): # Fix this
                account_memory = test_env.env_method(method_name="save_asset_memory")
                actions_memory = test_env.env_method(method_name="save_action_memory")
            #                 state_memory=test_env.env_method(method_name="save_state_memory") # add cu
            if dones[0]:
                print("hit end!")
                break
        return account_memory[0], actions_memory[0]

    @staticmethod
    def DRL_prediction_load_from_file(model_name, environment, cwd, deterministic=True):
        if model_name not in MODELS:
            raise NotImplementedError("NotImplementedError")
        try:
            # load agent
            model = MODELS[model_name].load(cwd)
            print("Successfully load model", cwd)
        except BaseException:
            raise ValueError("Fail to load agent!")

        # test on the testing env
        state = environment.reset()
        episode_returns = []  # the cumulative_return / initial_account
        episode_total_assets = [environment.initial_total_asset]
        done = False
        while not done:
            action = model.predict(state, deterministic=deterministic)[0]
            state, reward, done, _ = environment.step(action)

            total_asset = (
                environment.amount
                + (environment.price_ary[environment.day] * environment.stocks).sum()
            )
            episode_total_assets.append(total_asset)
            episode_return = total_asset / environment.initial_total_asset
            episode_returns.append(episode_return)

        print("episode_return", episode_return)
        print("Test Finished!")
        return episode_total_assets
```

b) Training - The *DRLAgent.train_model()* model

This is the first method that links SB3 algorithm classes to the FinRL portfolio environment. The training method is implemented as follows:

In [ ]:
```
        def train_model(self, model, tb_log_name, total_timesteps=5000):

            model = model.learn(
                total_timesteps=total_timesteps,
                tb_log_name=tb_log_name,
                callback=TensorboardCallback())

            return model
```

- The *learn()* method of *On/OffPolicyAlgorithm* calls its *collect_rollouts()* method, which takes the training environment of the algorithm as an argument. (The latter is where the *step()* method is called)

- The detailed description of functions called through *BaseAlgorithm.learn()*, the reader is referred to Sections 2.b-c on SB3 models.

In [ ]:

In [ ]:

c) Backtesting - The *DRLAgent.DRL_prediction()* method

Here is the backtesting algorithm, from *agents.stablebaselines3.models* submodule of FinRL:

In [ ]:

```
def DRL_prediction(model, environment, deterministic=True):
    '''
        Perform backtest of portfolio optimization using
        a stable_baselines3 deep RL algorithm
    '''

    # Init. SB3 environment
    test_env, test_obs = environment.get_sb_env()
    test_env.reset() # Tautological,

    # Init. lists
    account_memory = []
    actions_memory = []

    # Main loop
    for i in range(len(environment.df.index.unique())):
        # Compute action corresponding to current observation
        action, _states = model.predict(test_obs, deterministic=deterministic)

        # Execute the step() function
        test_obs, rewards, dones, info = test_env.step(action)

        # Store history
        if i == (len(environment.df.index.unique()) - 2):
            account_memory = test_env.env_method(method_name="save_asset_memory")
            actions_memory = test_env.env_method(method_name="save_action_memory")

        if dones[0]:
            print("hit end!")
            break

    return account_memory[0], actions_memory[0]
```

Ref: https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/agents/stablebaselines3/models.py.

We note the following:

- The *test_env* object is an SB3 *DummyVecEnv*. This is for normalization purposes since *model* is an SB3 algorithm.
- As mentioned in the section on SB3's *DummyVecEnv/VecEnv* environment classes, the *reset()* and *step()* methods of these classes are just wrappers for the corresponding methods of *StockPortfolioEnv*. The *step()* method called above is precisely FinRL's.
- Regarding the output: The code above returns the *asset_memory* and *actions_memory* attributes of the underlying *StockPortfolioEnv*. These are respectively the daily portfolio values and portfolio weights during the backtest, up to the day before last.
- A more in-depth description of the process behind *model.predict()* call is given in Sections 2.b-c on SB3 models.

In [ ]:

d) Backtest plots and statistics

The submodule *finrl.plot* contains several utilities for analyzing the results of a backtest:

https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/plot.py.

The function used in FinRL's tutorials is *backtest_plot()*, for which:

- One needs the start and end dates of the backtest, as well as the baseline ticker.
- For the baseline, *backtest_plot()* calls *finrl.plot.get_baseline()* which uses the Yahoo downloader to acquire the baseline data.
- The backtest output required by the method is the daily portfolio value. *backtest_plot()* has a *value_col_name* for specifying which column of the input dataframe *account_value* contains the daily account/portfolio values.

The last block of *backtest_plot()* opens a context manager to call the *create_full_tear_sheet()* function of Quantopian's *PyFolio*. We note that:

- The inputs of *create_full_tear_sheet()* are the daily portfolio returns and benchmark returns, and the
- The implemetation is here: https://github.com/quantopian/pyfolio/blob/master/pyfolio/tears.py.
- For the computation of drawdown periods, this method calls the *timeseries* submodule of *pyfolio*, in which there is (still) a bug at the time of writing (22/09/30). *pyfolio.timeseries* can easily be patched following the discussion here: https://stackoverflow.com/questions/63554616/attributeerror-numpy-int64-object-has-no-attribute-to-pydatetime.

To summarize, FinRL's performance plots and statistics are just those of *PyFolio*. More details on the metrics and statistics computed by this library's tearsheets can be found at the following links:

- Brief description: https://pyfolio.ml4trading.io/.
- Full tear sheet example: https://pyfolio.ml4trading.io/notebooks/zipline_algo_example.html#Full-tear-sheet-example.

---

5 - Financial data - Acquisition and Preprocessing

a) Downloader

The most used downloader in FinRL's tutorials is the *YahooDownloader* class in *meta.preprocessor.yahoodownloader*:

https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/meta/preprocessor/yahoodownloader.py

As illustrated in the first section on general pipelines, these objects are used for their *fetch_data()* method:

In [ ]:
```
def fetch_data(self, proxy=None) -> pd.DataFrame:
        """Fetches data from Yahoo API
        Parameters
        ----------
        Returns
        -------
        `pd.DataFrame`
            7 columns: A date, open, high, low, close, volume and tick symbol
            for the specified stock ticker
        """
        # Download and save the data in a pandas DataFrame:
        data_df = pd.DataFrame()
        for tic in self.ticker_list:
            temp_df = yf.download(
```

```
                tic, start=self.start_date, end=self.end_date, proxy=proxy
            )
            temp_df["tic"] = tic
            data_df = data_df.append(temp_df)
        # reset the index, we want to use numbers as index instead of dates
        data_df = data_df.reset_index()
        try:
            # convert the column names to standardized names
            data_df.columns = ["date", "open", "high", "low",
                               "close", "adjcp","volume","tic"]
            # use adjusted close price instead of close price
            data_df["close"] = data_df["adjcp"]
            # drop the adjusted close price column
            data_df = data_df.drop(labels="adjcp", axis=1)
        except NotImplementedError:
            print("the features are not supported currently")
        # create day of the week column (monday = 0)
        data_df["day"] = data_df["date"].dt.dayofweek
        # convert date to standard string format, easy to filter
        data_df["date"] = data_df.date.apply(lambda x: x.strftime("%Y-%m-%d"))
        # drop missing data
        data_df = data_df.dropna()
        data_df = data_df.reset_index(drop=True)
        print("Shape of DataFrame: ", data_df.shape)
        # print("Display DataFrame: ", data_df.head())

        data_df = data_df.sort_values(by=["date", "tic"]).reset_index(drop=True)

        return data_df
```

Here, the principal takeaway is the formatting of the Pandas dataframes into FinRL's format as described in Section 3.b (see *df* input of *StockPortfolioEnv* constructor).

Comment: I doubt that dropping the adjusted close column is a good idea for portfolio optimization. It makes sense for the stock trading environment since the adjusted closes aren't used for (live) paper trading, but this information is important for realistic backtests.

b) The *FeatureEngineer* class

This class is found in the *meta.preprocessor.preprocessors* submodule:

https://github.com/AI4Finance-Foundation/FinRL/blob/master/finrl/meta/preprocessor/preprocessors.py.

Its principal method is *preprocess_data()*, which is used for example as follows:

In [ ]:
```
fe = FeatureEngineer(
        use_technical_indicator=True,
        tech_indicator_list = INDICATORS,
        use_vix=True,
        use_turbulence=True,
        user_defined_feature = False)
df = fe.preprocess_data(df)
```

To comment on the arguments of *FeatureEngineer*, each of the following are computed using a corresponding class method:

- The users can implement their own features by modifying the *preprocess_data()* method.
- The *use_turbulence* boolean indicates whether to add the *turbulence index* of stocks (calls *calculate_turbulence()*) to the dataset. In technical terms, the turbulence index is the Mahalanobis distance between the historical average of asset returns and their values at a given period (see https://www.top1000funds.com/wp-content/uploads/2010/11/FAJskulls.pdf by Kritzman-Li). More informally, it is supposed to quantify the degree of unusual return behavior within a "universe" of assets.

- The *use_vix* boolean indicates whether to add the *CBOE volatility index* (VIX) to the data. This is essentially an additional (real-time) ticker that is derived from prices of SPX index options with near-term expiration dates, the VIX generates a 30-day forward projection of volatility (https://www.investopedia.com/terms/v/vix.asp). This is supposed to gauge market sentiment.

- The *use_technical_indicator* boolean indicates whether to add the technical indicators specified in *tech_indicator_list* to the dataset. These are computed using the *stockstats* library, a certain add-on to *Pandas* that we discuss more below.

- Before computing any of the quantities above, *preprocess_data()* calls *clean_data()* that drops tickers with inconsistent merged closes (see code on GitHub).

Returning now to the *stockstats* library, the description and code are at the following pages:

- Docs: https://openbase.com/python/stockstats/documentation
- GitHub Page: https://github.com/jealous/stockstats

This library provides a wrapper class *StockDataFrame* for *pandas.DataFrame* and computes about 40 technical indicators from the date-close-high-low-volume columns of the data.

In FinRL's portfolio optimization tutorial, the technical indicators used are the following:

- Moving Average Convergence Divergence(MACD): A trend-following momentum indicator indicator that shows the relationship between two exponential moving averages (EMA's) of a security's price. The MACD is calculated by subtracting the 26-period exponential moving average (EMA) from the 12-period EMA. (Source: https://www.investopedia.com/terms/m/macd.asp)

- Relative Strength Index (RSI): A momentum indicator used in technical analysis. RSI measures the speed and magnitude of a security's recent price changes to evaluate overvalued or undervalued conditions in the price of that security (source: https://www.investopedia.com/terms/r/rsi.asp). FinRL use *rsi_30* for RSI over 30 days.

- Commodity Channel Index (CCI): A momentum-based oscillator used to help determine when an investment vehicle is reaching a condition of being overbought or oversold (source: https://www.investopedia.com/terms/c/commoditychannelindex.asp). FinRL use *cci_30* for CCI over 30 days.

- Directional Movement Index (DX): A technical indicator that measures both the strength and direction of a price movement and is intended to reduce false signals (source: https://www.investopedia.com/terms/d/dmi.asp). FinRL use 'dx_30'.

Comments:

1) In a more refined version of *StockPortfolioEnv*, it would be ideal to have technical indicators computed using the C-based *TA-Lib* library (https://mrjbq7.github.io/ta-lib/doc_index.html). I've noticed some inconsistent values in the dataframes obtained from *stockstats*, and I would rather use *TA-Lib* if possible, given that it's been developed and maintained for 17 years before *stockstats*.

2) Technical analysis is astrology for traders. (Source: https://i.redd.it/0hkqhlj7e4w61.jpg)

c) Return covariances

In principle, the calculation of return covariances over the lookback period should be part of the *FeatureEngineer.preprocess_data()* function discussed above. In the portfolio optimization tutorial it is implemented by hand before the training:

In [ ]:
```
### Add covariance matrix to states
```

```
df=df.sort_values(['date','tic'],ignore_index=True)
df.index = df.date.factorize()[0]
cov_list = []
return_list = []
lookback=252 # look back is one year

for i in range(lookback,len(df.index.unique())):
    # Init. temp. lookback dataframe
    data_lookback = df.loc[i-lookback:i,:]

    # Init. temp. close prices dataframe
    price_lookback=data_lookback.pivot_table(index = 'date',columns = 'tic', values = 'close')

    # Compute daily returns of price_lookback and append return_list
    return_lookback = price_lookback.pct_change().dropna()
    return_list.append(return_lookback)

    # Compute returns covariance and append to cov_list
    covs = return_lookback.cov().values
    cov_list.append(covs)

df_cov = pd.DataFrame({'date':df.date.unique()[lookback:],'cov_list':cov_list,'return_list':return_list})

# Assign 'cov_list' and 'return_list' to all tickers and sort
df = df.merge(df_cov, on='date')
df = df.sort_values(['date','tic']).reset_index(drop=True) # This is what drops the first 252 days
```

Appendix

a) Imports

For the sake of completeness, we include a (more or less complete) list of imports to give an idea of he packages used in the text above.

In [ ]:
```
# Kill warnings
from warnings import filterwarnings
filterwarnings("ignore")
```

In [ ]:
```
### Basics
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
#matplotlib.use('Agg')
%matplotlib inline
import datetime
import itertools

### OpenAI Gym and StableBaselines3
from gym.utils import seeding
import gym
from gym import spaces
from stable_baselines3.common.vec_env import DummyVecEnv

### Data: Acquisition and preprocessing
from finrl.meta.preprocessor.yahoodownloader import YahooDownloader
from finrl.meta.preprocessor.preprocessors import FeatureEngineer, data_split
from finrl.meta.data_processor import DataProcessor

### FinRL: Environments
from finrl.meta.env_stock_trading.env_stocktrading import StockTradingEnv
## NOTE: FinRL re-implement their portfolio optimization environment in a notebook.

### FinRL: Deep RL algos
```

```
from finrl.agents.stablebaselines3.models import DRLAgent

### Backtesting
from finrl.plot import backtest_stats, backtest_plot, get_daily_return, get_baseline
from pprint import pprint

### What is this?
'''
import sys
sys.path.append("../FinRL")
'''
# Backtest imports
import torch
import plotly.express as px

### PyPortFolioOpt
from pypfopt.efficient_frontier import EfficientFrontier
from pypfopt import risk_models
from pypfopt import EfficientFrontier
from pypfopt import risk_models
from pypfopt import expected_returns
from pypfopt import objective_functions

### PyFolio and FinRL backtesting
import pyfolio
from pyfolio import timeseries # WARNING: Make sure this one is debugged
from finrl.plot import backtest_stats, backtest_plot, get_daily_return, get_baseline
from finrl.plot import convert_daily_return_to_pyfolio_ts # 22/09/15: Issue here
```

b) Questions that should be answered (22/10/13):

The actual purpose of this notebook is to clarify the following for future users of FinRL:

- Write-down the typical FinRL pipeline, from data acquisition to backtest result analysis. **(Done)**
- The portfolio optimization environment:
  - Where are the states/actions written and/or modified. **(Done)**
  - Where are the rewards modified. **(Done)**
  - Objective functions: FinRL tutorial use the Sharpe ratio by default. Find how to switch this to cumulative returns. **(Done. Not Sharpe, see rewards)**
- The FinRL DRLAgent class VS StableBaselines3:
  - Find the original stablebaselines Model class, clarify how the latter is modified. **(Done)**
  - Understand the implementation of states, actions, and rewards. **(Done)**
  - Understand how the deep network(s) associated to an agent are instantiated **(Done)**, and how to customize them **(Done. See SB3.)**.
  - Understand the training method of the DRLAgent class. **(Done)**
  - Clarify what the output of an agent's network is in FinRL's implementation. **(Done. Actions are outputs of actor networks. Their softmax gives the portfolio weights)**
  - Continuing previous, clarify how the policy is used for stock trading and for portfolio optimization. Might have to do this case by case (A2C, PPO, DDPG, SAC etc.) **(Done)**
  - Clarify where the transaction costs are computed/added and if they are accounted for in the rewards. **(Done. Not used in present implementation.)**
  - Understand argument passing between the environment and the agents in FinRL. **(Done. See StockPortfolioEnv.step() and DRLAgent.DRL_prediction())**
  - **Conceptual:** What does the discounted expected cumulative returns represent if the returns are the portfolio value?
- Backtesting:
  - Clarify FinRL's backtesting algorithms. **(Done)**
  - Analyzing results with pyfolio (comment on bugs in that library). **(Done)**
  - Saving the results of a backtest. **(Just export to csv?)**

- Data handling:
  - Clarify the preprocessing routines. **(Done)**
  - Comment on technical indicators, turbulence and adding other alpha factors. **(Done)**
- Software engineering:
  - Document everything. Several classes and functions will potentially have to be modified.
  - Create a submodule with the modifications, and a functional "__main__" file with the full pipeline.
  - Update the environment requirements file.

In [ ]: