FinRL Documentation

Release 0.3.1

FinRL

Dec 18, 2022

CONTENTS

Disclaimer: Nothing herein is financial advice, and NOT a recommendation to trade real money. Please use common sense and always first consult a professional before trading or investing.

AI4Finance community provides this demonstrative and educational resource, in order to efficiently automate trading. FinRL is the first open source framework for financial reinforcement learning.

Reinforcement learning (RL) trains an agent to solve tasks by trial and error, while DRL uses deep neural networks as function approximators. DRL balances exploration (of uncharted territory) and exploitation (of current knowledge), and has been recognized as a competitive edge for automated trading. DRL framework is powerful in solving dynamic decision making problems by learning through interactions with an unknown environment, thus exhibiting two major advantages: portfolio scalability and market model independence. Automated trading is essentially making dynamic decisions, namely **to decide where to trade, at what price, and what quantity**, over a highly stochastic and complex stock market. Taking many complex financial factors into account, DRL trading agents build a multi-factor model and provide algorithmic trading strategies, which are difficult for human traders.

FinRL provides a framework that supports various markets, SOTA DRL algorithms, benchmarks of many quant finance tasks, live trading, etc.

Join or discuss FinRL with us: AI4Finance mailing list.

Feel free to leave us feedback: report bugs using Github issues or discuss FinRL development in the Slack Channel.

CHAPTER

ONE

INTRODUCTION

Table of Contents

Design Principles

- Plug-and-Play (PnP): Modularity; Handle different markets (say T0 vs. T+1)
- Completeness and universal: Multiple markets; Various data sources (APIs, Excel, etc); User-friendly variables.
- Avoid hard-coded parameters
- Closing the sim-real gap using the "training-testing-trading" pipeline: simulation for training and connecting real-time APIs for testing/trading.
- Efficient data sampling: accelerate the data sampling process is the key to DRL training! From the ElegantRL project. We know that multi-processing is powerful to reduce the training time (scheduling between CPU + GPU).
- ransparency: a virtual env that is invisible to the upper layer
- Flexibility and extensibility: Inheritance might be helpful here

Contributions

- FinRL is an open source library specifically designed and implemented for quantitative finance. Trading environments incorporating market frictions are used and provided.
- Trading tasks accompanied by hands-on tutorials with built-in DRL agents are available in a beginner-friendly and reproducible fashion using Jupyter notebook. Customization of trading time steps is feasible.
- FinRL has good scalability, with fine-tuned state-of-the-art DRL algorithms. Adjusting the implementations to the rapid changing stock market is well supported.
- Typical use cases are selected to establish benchmarks for the quantitative finance community. Standard backtesting and evaluation metrics are also provided for easy and effective performance evaluation.

With FinRL library, the implementation of powerful DRL trading strategies becomes more accessible, efficient and delightful.

CHAPTER

TWO

FIRST GLANCE

To quickly understand what is FinRL and how it works, you can go through the notebook FinRL_StockTrading_NeurIPS_2018.ipynb

This is how we use Deep Reinforcement Learning for Stock Trading from scratch.

Tip: Run the code step by step at Google Colab.

The notebook and the following result is based on our paper *Practical deep reinforcement learning approach for stock trading* Xiong, Zhuoran, Xiao-Yang Liu, Shan Zhong, Hongyang Yang, and Anwar Walid. "Practical deep reinforcement learning approach for stock trading." arXiv preprint arXiv:1811.07522 (2018).

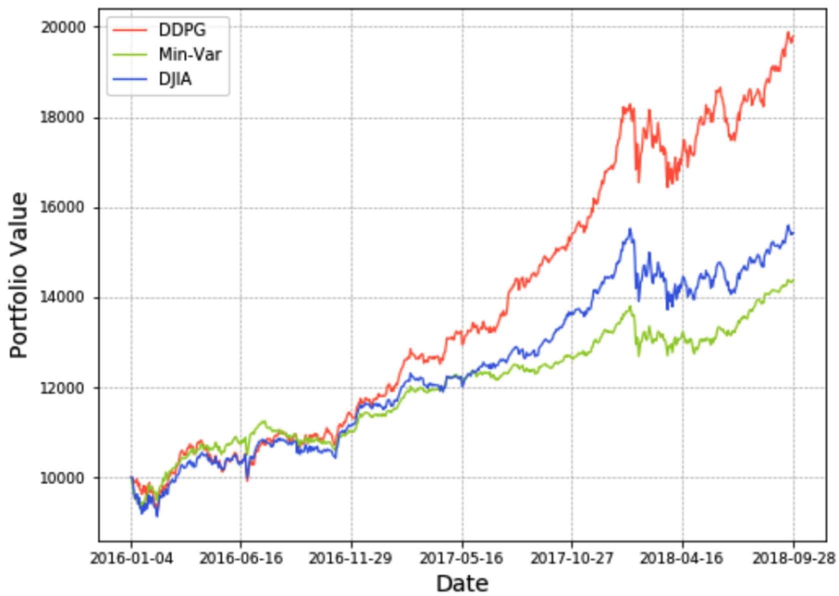


Figure 4: Portfolio value curves of our DDPG scheme, the min-variance portfolio allocation strategy, and the Dow Jones Industrial Average. (Initial portfolio value $10, 000).

Table 1: Trading Performance.

	DDPG (ours)	Min-Variance	DJIA
Initial Portfolio Value	**10, 000**	10, 000	10, 000
Final Portfolio Value	**19, 791**	14, 369	15, 428
Annualized Return	**25.87%**	15.93%	16.40%
Annualized Std. Error	**13.62%**	9.97%	11.70%
Sharpe Ratio	**1.79**	1.45	1.27

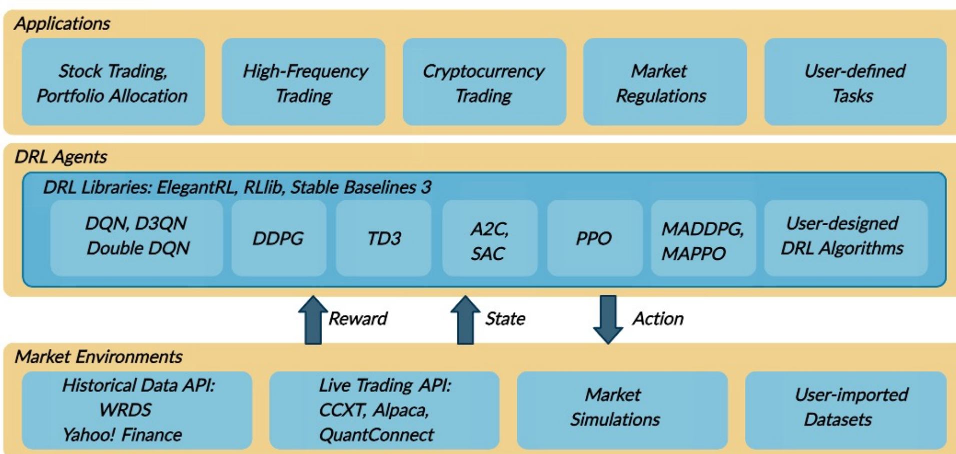
CHAPTER

THREE

THREE-LAYER ARCHITECTURE

After the first glance of how to establish our task on stock trading using DRL, know we are introducing the most central idea of FinRL.

FinRL library consists of three layers: **market environments (FinRL-Meta)**, **DRL agents** and **applications**. The lower layer provides APIs for the upper layer, making the lower layer transparent to the upper layer. The agent layer interacts with the environment layer in an exploration-exploitation manner, whether to repeat prior working-well decisions or to make new actions hoping to get greater cumulative rewards.



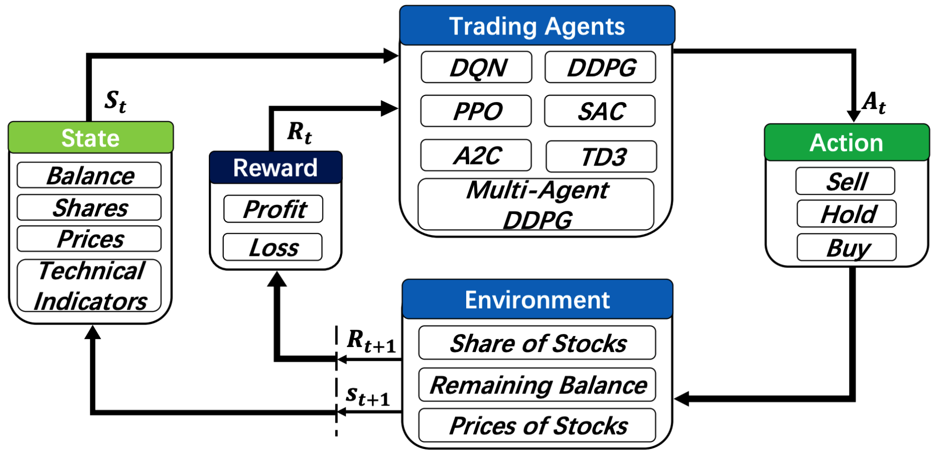
Our construction has following advantages:

Modularity: Each layer includes several modules and each module defines a separate function. One can select certain modules from a layer to implement his/her stock trading task. Furthermore, updating existing modules is possible.

Simplicity, Applicability and Extendibility: Specifically designed for automated stock trading, FinRL presents DRL algorithms as modules. In this way, FinRL is made accessible yet not demanding. FinRL provides three trading tasks as use cases that can be easily reproduced. Each layer includes reserved interfaces that allow users to develop new modules.

Better Market Environment Modeling: We build a trading simulator that replicates live stock markets and provides backtesting support that incorporates important market frictions such as transaction cost, market liquidity and the investor's degree of risk-aversion. All of those are crucial among key determinants of net returns.

A high level view of how FinRL construct the problem in DRL:



Please refer to the following pages for more specific explanation:

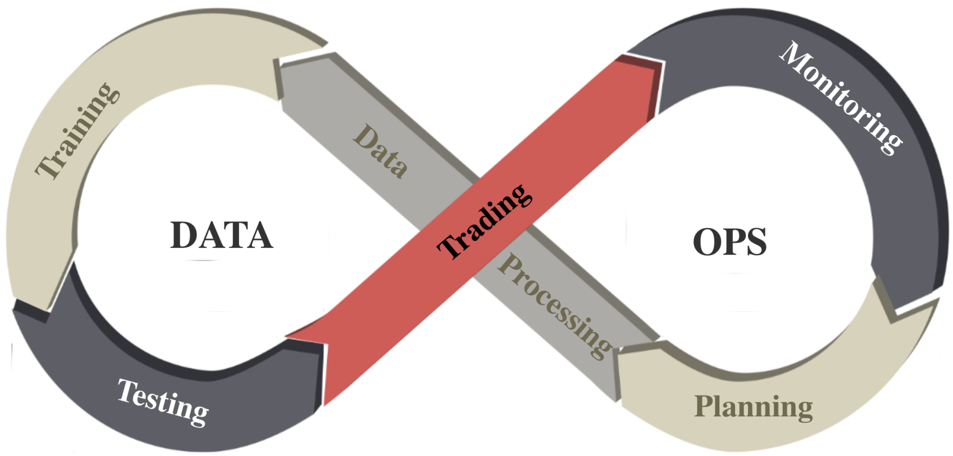
3.1 1. Stock Market Environments

Considering the stochastic and interactive nature of the automated stock trading tasks, a financial task is modeled as a Markov Decision Process (MDP) problem. FinRL-Meta first preprocesses the market data, and then builds stock market environments. The environemnt observes the change of stock price and multiple features, and the agent takes an action and receives the reward from the environment, and finally the agent adjusts its strategy accordingly. By interacting with the environment, the smart agent will derive a trading strategy to maximize the long-term accumulated rewards (also named as Q-value).

Our trading environments, based on OpenAI Gym, simulate the markets with real market data, using time-driven simulation. FinRL library strives to provide trading environments constructed by datasets across many stock exchanges.

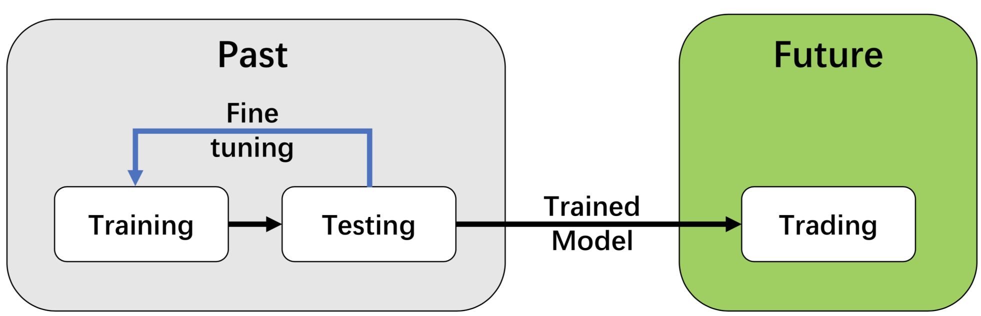
In the Tutorials and Examples section, we will illustrate the detailed MDP formulation with the components of the reinforcement learning environment.

The application of DRL in finance is different from that in other fields, such as playing chess and card games; the latter inherently have clearly defined rules for environments. Various finance markets require different DRL algorithms to get the most appropriate automated trading agent. Realizing that setting up a training environment is time-consuming and laborious work, FinRL provides market environments based on representative listings, including NASDAQ-100, DJIA, S&P 500, SSE 50, CSI 300, and HSI, plus a user-defined environment. Thus, this library frees users from tedious and time-consuming data pre-processing workload. We know that users may want to train trading agents on their own data sets. FinRL library provides convenient support to user-imported data and allows users to adjust the granularity of time steps. We specify the format of the data. According to our data format instructions, users only need to pre-process their data sets.



We follow the DataOps paradigm in the data layer.

- We establish a standard pipeline for financial data engineering in RL, ensuring data of **different formats** from different sources can be incorporated in **a unified framework**.
- We automate this pipeline with a **data processor**, which can access data, clean data, and extract features from various data sources with high quality and efficiency. Our data layer provides agility to model deployment.
- We employ a **training-testing-trading pipeline**. The DRL agent first learns from the training environment and is then validated in the validation environment for further adjustment. Then the validated agent is tested in historical datasets. Finally, the tested agent will be deployed in paper trading or live trading markets. First, this pipeline **solves the information leakage problem** because the trading data are never leaked when adjusting agents. Second, a unified pipeline **allows fair comparisons** among different algorithms and strategies.



For data processing and building environment for DRL in finance, AI4Finance has maintained another project: FinRL-Meta.

3.2 2. DRL Agents

FinRL contains fine-tuned standard DRL algorithms in ElegantRL, Stable Baseline 3, and RLlib. ElegantRL is a scalable and elastic DRL library that maintained by AI4Finance, with faster and more stable performance than Stable Baseline 3 and RLlib. In the *Three-Layer Architecture* section, there will be detailed explanation about how ElegantRL accomplish its role in FinRL perfectly. If interested, please refer to ElegantRL's GitHub page or documentation.

With those three powerful DRL libraries, FinRL provides the following algorithms for users:

start/image/alg_compare.png

As mentioned in the introduction, FinRL's DRL agents are built by fine-tuned standard DRL algorithms depending on three famous DRL library: ElegantRL, Stable Baseline 3, and RLlib.

The supported algorithms include: DQN, DDPG, Multi-Agent DDPG, PPO, SAC, A2C and TD3. We also allow users to design their own DRL algorithms by adapting these DRL algorithms, e.g., Adaptive DDPG, or employing ensemble methods. The comparison of DRL algorithms is shown in the table bellow:

Algorithms	Input	Output	Type	State-action spaces support	Finance use cases support	Features and Improvements	Advantages
DQN	States	Q-value	Value based	Discrete only	Single stock trading	Target network, experience replay	Simple and easy to use
Double DQN	States	Q-value	Value based	Discrete only	Single stock trading	Use two identical neural network models to learn	Reduce overestimations
Dueling DQN	States	Q-value	Value based	Discrete only	Single stock trading	Add a specialized dueling Q head	Better differentiate actions, improves the learning
DDPG	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Being deep Q-learning for continuous action spaces	Better at handling high-dimensional continuous action spaces
A2C	State action pair	Q-value	Actor-critic based	Discrete and continuous	All use cases	Advantage function, parallel gradients updating	Stable, cost-effective, faster and works better with large batch sizes
PPO	State action pair	Q-value	Actor-critic based	Discrete and continuous	All use cases	Clipped surrogate objective function	Improve stability, less variance, simply to implement
SAC	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Entropy regularization, exploration-exploitation trade-off	Improve stability
TD3	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Clipped double Q-Learning, delayed policy update, target policy smoothing.	Improve DDPG performance
MADDPG	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Handle multi-agent RL problem	Improve stability and performance

Users are able to choose their favorite DRL agents for training. Different DRL agents might have different performance in various tasks.

3.2.1 ElegantRL: DRL library

One sentence summary of reinforcement learning (RL): in RL, an agent learns by continuously interacting with an unknown environment, in a trial-and-error manner, making sequential decisions under uncertainty and achieving a balance between exploration (new territory) and exploitation (using knowledge learned from experiences).

Deep reinforcement learning (DRL) has great potential to solve real-world problems that are challenging to humans, such as gaming, natural language processing (NLP), self-driving cars, and financial trading. Starting from the success of AlphaGo, various DRL algorithms and applications are emerging in a disruptive manner. The ElegantRL library enables researchers and practitioners to pipeline the disruptive "design, development and deployment" of DRL technology.

The library to be presented is featured with "elegant" in the following aspects:

- Lightweight: core codes have less than 1,000 lines, e.g., helloworld.
- Efficient: the performance is comparable with Ray RLlib.
- Stable: more stable than Stable Baseline 3.

ElegantRL supports state-of-the-art DRL algorithms, including discrete and continuous ones, and provides user-friendly tutorials in Jupyter notebooks. The ElegantRL implements DRL algorithms under the Actor-Critic framework, where an Agent (a.k.a, a DRL algorithm) consists of an Actor network and a Critic network. Due to the completeness and simplicity of code structure, users are able to easily customize their own agents.

Please refer to ElegantRL's GitHub page or documentation for more details.

3.3 3. Applications

CHAPTER

FOUR

INSTALLATION

4.1 MAC OS

4.1.1 Step 1: Install Anaconda

-Download Anaconda Installer, Anaconda has everything you need for Python programming.

-Follow Anaconda's instruction: macOS graphical install, to install the newest version of Anaconda.

-Open your terminal and type: *'which python'*, it should show:

```
/Users/your_user_name/opt/anaconda3/bin/python
```

It means that your Python interpreter path has been pinned to Anaconda's python version. If it shows something like this:

```
/Users/your_user_name/opt/anaconda3/bin/python
```

It means that you still use the default python path, you either fix it and pin it to the anaconda path (try this blog), or you can use Anaconda Navigator to open a terminal manually.

4.1.2 Step 2: Install Homebrew

-Open a terminal and make sure that you have installed Anaconda.

-Install Homebrew:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/
↪install.sh)"
```

4.1.3 Step 3: Install OpenAI

Installation of system packages on Mac requires Homebrew. With Homebrew installed, run the following in your terminal:

```
brew install cmake openmpi
```

4.1.4 Step 4: Install FinRL

Since we are still actively updating the FinRL repository, please install the unstable development version of FinRL using pip:

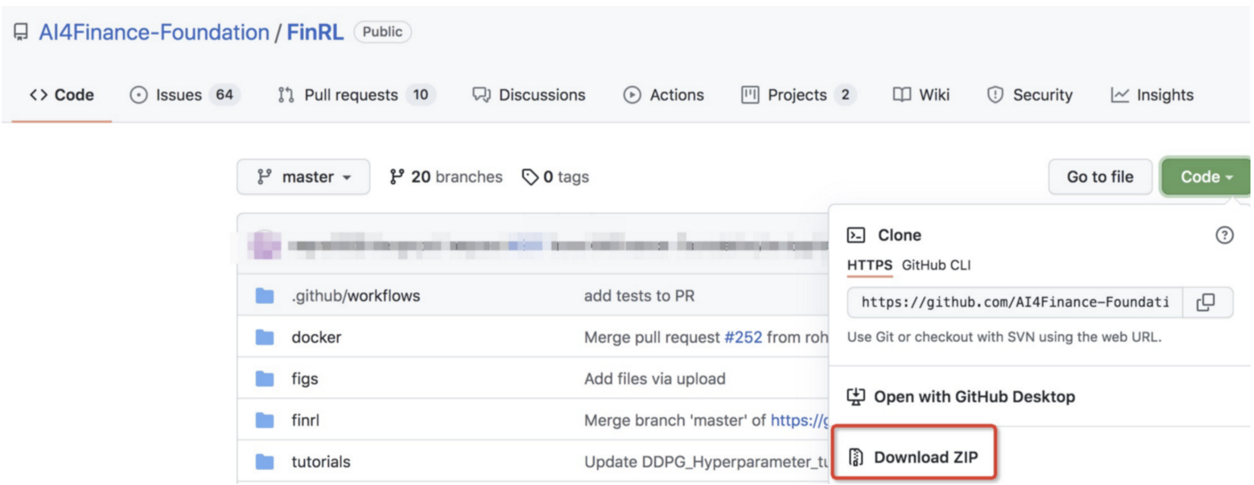
```
pip install git+https://github.com/AI4Finance-Foundation/FinRL.git
```

4.1.5 Step 5: Run FinRL

Download the FinRL repository either use terminal:

```
git clone https://github.com/AI4Finance-Foundation/FinRL.git
```

or download it manually



Open Jupyter Notebook through Anaconda Navigator and locate one of the stock trading notebook in FinRL/tutorials you just downloaded. You should be able to run it.

4.2 Ubuntu

4.2.1 Step 1: Install Anaconda

Please follow the steps in this blog

4.2.2 Step 2: Install OpenAI

Open an ubuntu terminal and type:

```
sudo apt-get update && sudo apt-get install cmake libopenmpi-dev python3-dev zlib1g-dev libgl1-mesa-glx
```

4.2.3 Step 3: Install FinRL

Since we are still actively updating the FinRL repository, please install the unstable development version of FinRL using pip:

```
pip install git+https://github.com/AI4Finance-Foundation/FinRL.git
```

4.2.4 Step 4: Run FinRL

Download the FinRL repository in terminal:

```
git clone https://github.com/AI4Finance-Foundation/FinRL.git
```

Open Jupyter Notebook by typing 'jupyter notebook' in your ubuntu terminal.

Locate one of the stock trading notebook in FinRL/tutorials you just downloaded. You should be able to run it.

4.3 Windows 10

4.3.1 Prepare for install

1. VPN is needed if using YahooFinance in china (pyfolio, elegantRL pip dependencies need pull code, YahooFinance has stopped the service in china). Othewise, please ignore it.
2. python version >=3.7
3. pip remove zipline, if your system has installed zipline, zipline has conflicts with the FinRL.

4.3.2 Step1: Clone FinRL

```
git clone https://github.com/AI4Finance-Foundation/FinRL.git
```

4.3.3 Step2: install dependencies

```
cd FinRL
pip install .
```

4.3.4 Step3: test (If using YahooFinance in China, VPN is needed)

```
python FinRL_StockTrading_NeurIPS_2018.py
```

4.3.5 Tips for running error

If the following outputs appear, take it easy, since installation is still successful.

1. UserWarning: Module "zipline.assets" not found; multipliers will not be applied to position notionals. Module "zipline.assets" not found; multipliers will not be applied'

If following outputs appear, please ensure that VPN helps to access the YahooFinance

1. Failed download: xxxx: No data found for this date range, the stock may be delisted, or the value is missing.

4.4 Windows 10 (wsl install)

4.4.1 Step 1: Install Ubuntu on Windows 10

Please check this video for detailed steps:

4.4.2 Step 2: Install Anaconda

Please follow the steps in this blog

4.4.3 Step 3: Install OpenAI

Open an ubuntu terminal and type:

```
sudo apt-get update && sudo apt-get install cmake libopenmpi-dev python3-dev zlib1g-dev␣
↪libgl1-mesa-glx
```

4.4.4 Step 4: Install FinRL

Since we are still actively updating the FinRL repository, please install the unstable development version of FinRL using pip:

```
pip install git+https://github.com/AI4Finance-Foundation/FinRL.git
```

4.4.5 Step 5: Run FinRL

Download the FinRL repository in terminal:

```
git clone https://github.com/AI4Finance-Foundation/FinRL.git
```

Open Jupyter Notebook by typing 'jupyter notebook' in your ubuntu terminal. Please see jupyter notebook

Locate one of the stock trading notebook in FinRL/tutorials you just downloaded. You should be able to run it.

CHAPTER

FIVE

QUICK START

Open `main.py`

```
import os
```

from typing import List from argparse import ArgumentParser from finrl import config from finrl.config_tickers import DOW_30_TICKER from finrl.config import (

DATA_SAVE_DIR, TRAINED_MODEL_DIR, TENSORBOARD_LOG_DIR, RESULTS_DIR, INDICATORS, TRAIN_START_DATE, TRAIN_END_DATE, TEST_START_DATE, TEST_END_DATE, TRADE_START_DATE, TRADE_END_DATE, ERL_PARAMS, RLlib_PARAMS, SAC_PARAMS, ALPACA_API_KEY, ALPACA_API_SECRET, ALPACA_API_BASE_URL,

)

construct environment from finrl.finrl_meta.env_stock_trading.env_stocktrading_np import StockTradingEnv

def build_parser():
parser = ArgumentParser() parser.add_argument(

"–mode", dest="mode", help="start mode, train, download_data" " backtest", metavar="MODE", default="train",

) return parser

"./" will be added in front of each directory def check_and_make_directories(directories: List[str]):

for directory in directories:

if not os.path.exists("./" + directory):
os.makedirs("./" + directory)

def main():
parser = build_parser() options = parser.parse_args() check_and_make_directories([DATA_SAVE_DIR, TRAINED_MODEL_DIR, TENSORBOARD_LOG_DIR, RESULTS_DIR])

if options.mode == "train":
from finrl import train

env = StockTradingEnv

demo for elegantrl kwargs = {} # in current finrl_meta, with respect yahoofinance, kwargs is {}. For other data sources, such as joinquant, kwargs is not empty train(

start_date=TRAIN_START_DATE, end_date=TRAIN_END_DATE, ticker_list=DOW_30_TICKER, data_source="yahoofinance", time_interval="1D", technical_indicator_list=INDICATORS, drl_lib="elegantrl", env=env, model_name="ppo", cwd="./test_ppo", erl_params=ERL_PARAMS, break_step=1e5, kwargs=kwargs,

)

elif options.mode == "test":
from finrl import test env = StockTradingEnv

demo for elegantrl kwargs = {} # in current finrl_meta, with respect yahoofinance, kwargs is {}. For other data sources, such as joinquant, kwargs is not empty

account_value_erl = test(
start_date=TEST_START_DATE, end_date=TEST_END_DATE, ticker_list=DOW_30_TICKER, data_source="yahoofinance", time_interval="1D", technical_indicator_list=INDICATORS, drl_lib="elegantrl", env=env, model_name="ppo", cwd="./test_ppo", net_dimension=512, kwargs=kwargs,

)

elif options.mode == "trade":
from finrl import trade env = StockTradingEnv kwargs = {} trade(

start_date=TRADE_START_DATE, end_date=TRADE_END_DATE, ticker_list=DOW_30_TICKER, data_source="yahoofinance", time_interval="1D", technical_indicator_list=INDICATORS, drl_lib="elegantrl", env=env, model_name="ppo", API_KEY=ALPACA_API_KEY, API_SECRET=ALPACA_API_SECRET, API_BASE_URL=ALPACA_API_BASE_URL, trade_mode='backtesting', if_vix=True, kwargs=kwargs,

)

else:
raise ValueError("Wrong mode.")

Users can input the following command in terminal # python main.py –mode=train # python main.py –mode=test # python main.py –mode=trade if __name__ == "__main__":

main()

Run the library:

```
python main.py --mode=train # if train. Use DOW_30_TICKER by default.
python main.py --mode=test  # if test. Use DOW_30_TICKER by default.
python main.py --mode=trade # if trade. Users should input your alpaca parameters in␣
↪config.py
```

Choices for `--mode`: start mode, train, download_data, backtest

CHAPTER SIX

BACKGROUND

6.1 Dataset: Financial Big Data

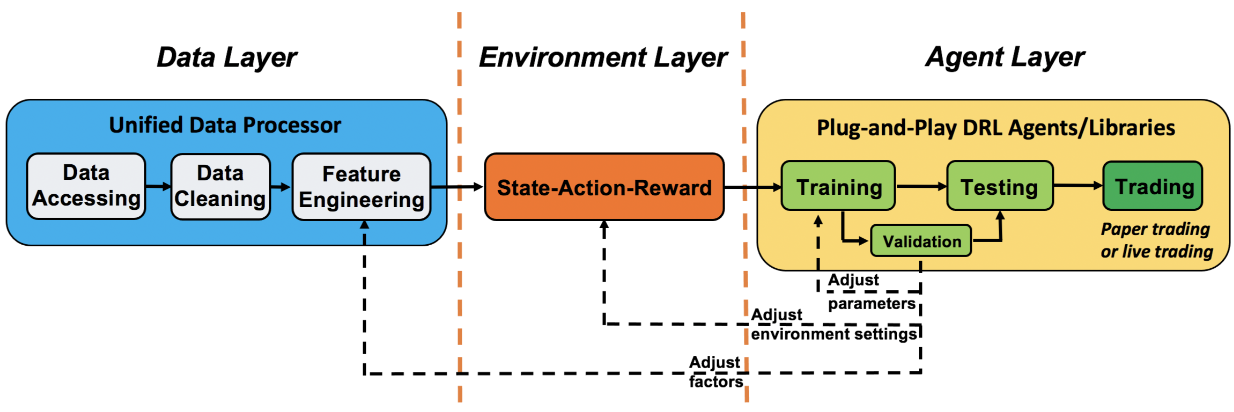
FinRL-Meta provides multiple datasets for financial reinforcement learning. Stepping into the era of internet, the speed of information exchange has an exponential increment. Along with that, the amount of data also explodes into an incredible number, which generates the new concept "big data".

As its data refreshing minute-to-second, finance is one of the most typical domains that big data imbeded in. Financial big data, as a new popular field, gets more and more attention by economists, data scientists, and computer scientists.

In academia, scholors use financial big data to explore more complex and precise understanding of market and economics. While industries use financial big data to refine their analytical strategies and strengthen their prediction models. Realizing the potential of this solid background, AI4Finance community started FinRL-Meta to serve for various needs by researchers and industries.

For datasets, FinRL-Meta has standardized flow of data extraction and cleaning for more than 30 different data sources. The purpose of providing the data pulling tool instead of a fixed dataset is better corresponding to the fast updating property of financial market. The dynamic construction can help users grip data according to their own requirement.

6.2 Benchmark



FinRL-Meta provides multiple benchmarks for financial reinforcement learning.

FinRL-Meta benchmarks work in famous papers and projects, covering stock trading, cyptocurrency trading, portfolio allocation, hyper-parameter tuning, etc. Along with that, there are Jupyter/Python demos that help users to test or design new strategies.

6.3 DataOps

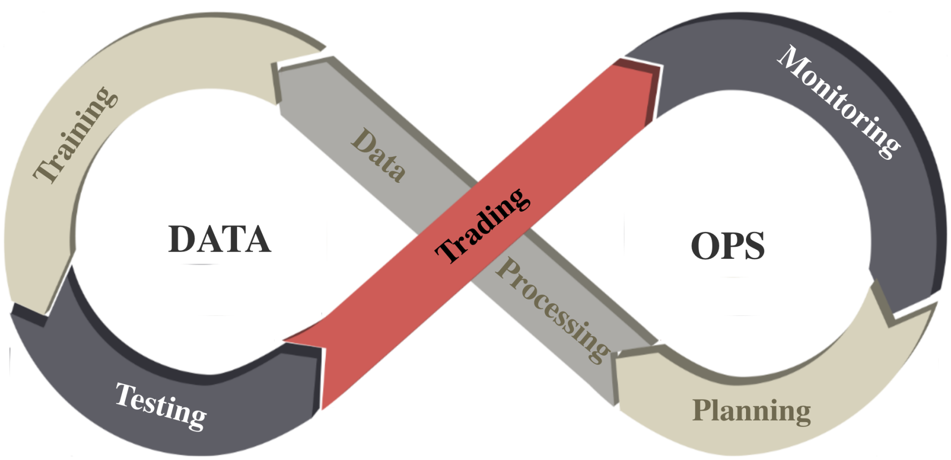
DataOps applies the ideas of lean development and DevOps to the data analytics field. DataOps practices have been developed in companies and organizations to improve the quality of and efficiency of data analytics. These implementations consolidate various data sources, unify and automate the pipeline of data analytics, including data accessing, cleaning, analysis, and visualization.

However, the DataOps methodology has not been applied to financial reinforcement learning researches. Most researchers access data, clean data, and extract technical indicators (features) in a case-by-case manner, which involves heavy manual work and may not guarantee the data quality.

To deal with financial big data (usually unstructured), we follow the DataOps paradigm and implement an automatic pipeline in the following figure: task planning, data processing, training-testing-trading, and monitoring agents' performance. Through this pipeline, we continuously produce DRL benchmarks on dynamic market datasets.

We follow the DataOps paradigm in the data layer.

1. we establish a standard pipeline for financial data engineering in RL, ensuring data of different formats from different sources can be incorporated in a unified framework.
2. we automate this pipeline with a data processor, which can access data, clean data, and extract features from various data sources with high quality and efficiency. Our data layer provides agility to model deployment.
3. we employ a training-testing-trading pipeline. The DRL agent first learns from the training environment and is then validated in the validation environment for further adjustment. Then the validated agent is tested in historical datasets. Finally, the tested agent will be deployed in paper trading or live trading markets. First, this pipeline solves the information leakage problem because the trading data are never leaked when adjusting agents. Second, a unified pipeline allows fair comparisons among different algorithms and strategies.



CHAPTER

SEVEN

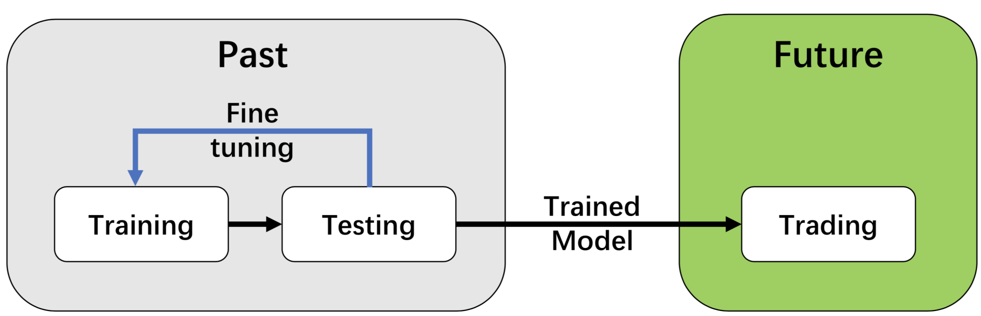
OVERVIEW

Following the *de facto* standard of OpenAI Gym, we build a universe of market environments for data-driven financial reinforcement learning, namely, FinRL-Meta. We keep the following design principles.

7.1 1. Supported trading tasks:

We have supported and achieved satisfactory trading performance for trading tasks such as stock trading, cryptocurrency trading, and portfolio allocation. Derivatives such as futures and forex are also supported. Besides, we have supported multi-agent simulation and execution optimizing tasks by reproducing the experiment in other published papers.

7.2 2. Training-testing-trading pipeline:

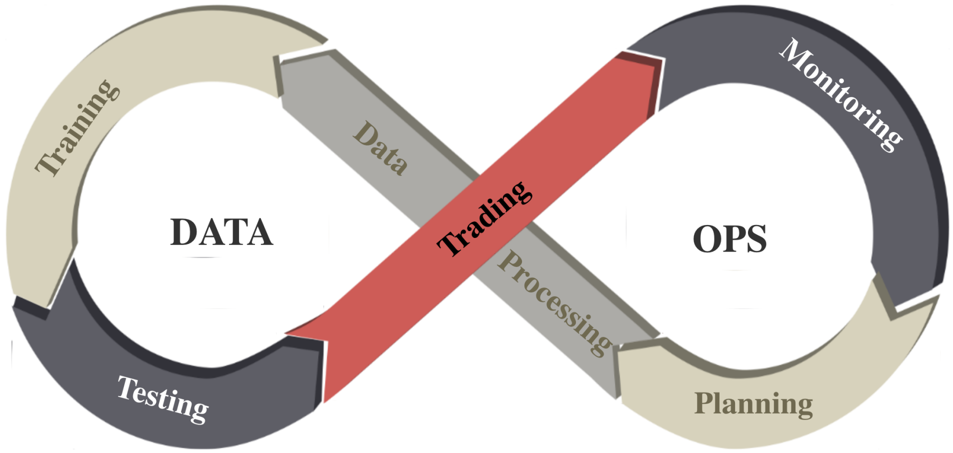


We employ a training-testing-trading pipeline that the DRL approach follows a standard end-to-end pipeline. The DRL agent is first trained in a training environment and then fined-tuned (adjusting hyperparameters) in a validation environment. Then the validated agent is tested on historical datasets (backtesting). Finally, the tested agent will be de- ployed in paper trading or live trading markets.

This pipeline solves the information leakage problem because the trading data are never leaked when training/tuning the agents.

Such a unified pipeline allows fair comparisons among different algorithms and strategies.

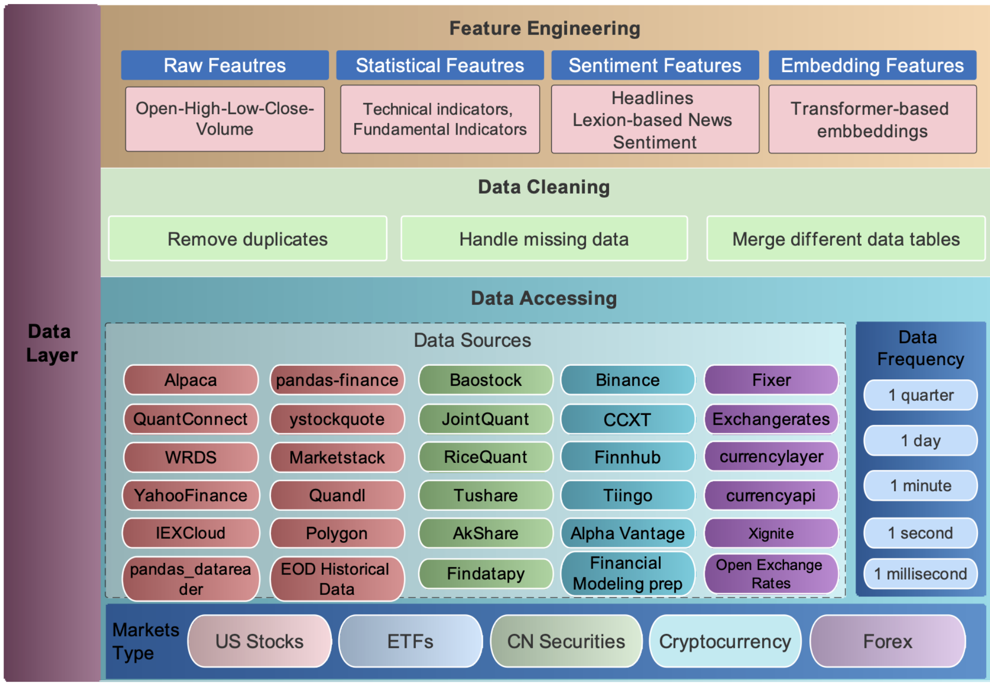
7.3 3. DataOps for data-driven financial reinforcement leanring



We follow the DataOps paradigm in the data layer, as shown in the figure above. First, we establish a standard pipeline for financial data engineering, ensuring data of different formats from different sources can be incorporated in a unified RL framework. Second, we automate this pipeline with a data processor, which can access data, clean data and extract features from various data sources with high quality and efficiency. Our data layer provides agility to model deployment.

7.4 4. Layered structure and extensibility

We adopt a layered structure for RL in finance, which consists of three layers: data layer, environment layer, and agent layer. Each layer executes its functions and is relatively independent. Meanwhile, layers interact through end-to-end interfaces to implement the complete workflow of algorithm trading, achieving high extensibility. For updates and substitutes inside the layer, this structure minimizes the impact on the whole system. Moreover, user-defined functions are easy to extend, and algorithms can be updated fast to keep high performance.



7.5 5. Plug-and-play

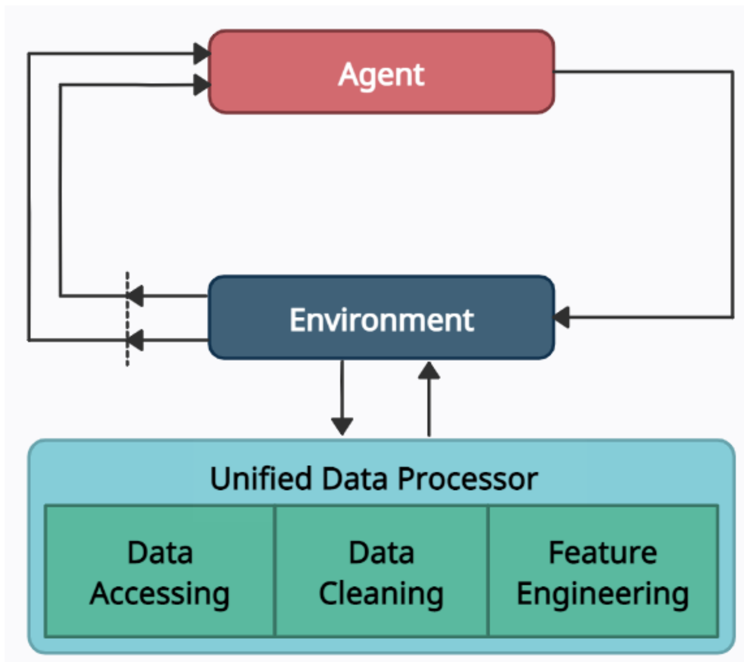
In the development pipeline, we separate market environments from the data layer and the agent layer. Any DRL agent can be directly plugged into our environments, then will be trained and tested. Different agents can run on the same benchmark environment for fair comparisons. Several popular DRL libraries are supported, including ElegantRL, RLlib, and SB3.

CHAPTER

EIGHT

DATA LAYER

In the data layer, we use a unified data processor to access data, clean data, and extract features.



8.1 Data Accessing

We connect data APIs of different platforms and unify them using a FinRL-Meta data processor. Users can access data from various sources given the start date, end date, stock list, time interval, and kwargs.

Data Source	Type	Range and Frequency	Request Limits	Raw Data	Preprocessed Data
Alpaca	US Stocks, ETFs	2015-now, 1min	Account-specific	OHLCV	Prices&Indicators
Baostock	CN Securities	1990-12-19-now, 5min	Account-specific	OHLCV	Prices&Indicators
Binance	Cryptocurrency	API-specific, 1s, 1min	API-specific	Tick-level daily aggregated trades, OHLCV	Prices&Indicators
CCXT	Cryptocurrency	API-specific, 1min	API-specific	OHLCV	Prices&Indicators
IEXCloud	NMS US securities	1970-now, 1 day	100 per second per IP	OHLCV	Prices&Indicators
JoinQuant	CN Securities	2005-now, 1min	3 requests each time	OHLCV	Prices&Indicators
QuantConnect	US Securities	1998-now, 1s	NA	OHLCV	Prices&Indicators
RiceQuant	CN Securities	2005-now, 1ms	Account-specific	OHLCV	Prices&Indicators
Tushare	CN Securities, A share	-now, 1 min	Account-specific	OHLCV	Prices&Indicators
WRDS	US Securities	2003-now, 1ms	5 requests each time	Intraday Trades	Prices&Indicators
YahooFinance	US Securities	Frequency-specific, 1min	2,000/hour	OHLCV	Prices&Indicators

8.2 Data Cleaning

Raw data retrieved from different data sources are usually of various formats and have erroneous or NaN data (missing data) to different extents, making data cleaning highly time-consuming. In FinRL-Meta, we automate the data cleaning process.

The cleaning processes of NaN data are usually different for various time frequencies. For Low-frequency data, except few stocks with extremely low liquidity, the few NaN values usually mean suspension during that time interval. While for high-frequency data, NaN values are pervasive, which usually means no transaction during that time interval. To reduce the simulation-to-reality gap considering of data efficiency, we provide different solutions for these two cases.

In the low-frequency case, we directly delete the rows with NaN values, reflecting suspension in simulated trading environments. However, it is not suitable to directly delete rows with NaN values in high-frequency cases.

In our test of downloading 1-min OHLCV data of DJIA 30 companies from Alpaca during 2021–01–01~2021–05–31, there were 39736 rows for the raw data. However, after dropping rows with NaN values, only 3361 rows are left.

The low data efficiency of the dropping method is unacceptable. Instead, we take an improved forward filling method. We fill the open, high, low, close columns with the last valid value of close price and the volume column with 0, which is a standard method in practice.

Although this filling method sacrifices the authenticity of the simulated environments, it is acceptable compared to significantly improved data efficiency, especially under tickers with high liquidity. Moreover, this filling method can be further improved using bid, ask prices to reduce the simulation-to-reality gap.

8.3 Feature Engineering

Feature engineering is the last part of the data layer. We automate the calculation of technical indicators by connecting the Stockstats or TAlib library in our data processor. Common technical indicators including Moving Average Convergence Divergence (MACD), Relative Strength Index (RSI), Average Directional Index (ADX), and Commodity Channel Index (CCI), and so on, are supported. Users can also quickly add indicators from other libraries, or add the user-defined features directly.

Users can add their features by two ways: 1) Write user-defined feature extraction functions directly. The returned features will be added to a feature array. 2) Store the features in a file, and move it to a specified folder. Then, these features will be obtained by reading from the specified file.

CHAPTER

NINE

ENVIRONMENT LAYER

FinRL-Meta follows the OpenAI gym-style [8] to create market environments using the cleaned data from the data layer. It provides hundreds of environments with a common interface. Users can build their environments based on FinRL-Meta environments easily, share their results and compare the strategies' performance. We will add more environments for convenience in the future.

CHAPTER

TEN

BENCHMARK

10.1 Performance Metrics

FinRL-Meta provides the following unified metrics to measure the trading performance:

- **Cumulative return:** $R = \frac{V - V_0}{V_0}$, where V is final portfolio value, and V_0 is original capital.
- **Annualized return:** $r = (1 + R)^{\frac{365}{t}} - 1$, where t is the number of trading days.
- **Annualized volatility:** $\sigma_a = \sqrt{\frac{\sum_{i=1}^{n} (r_i - \bar{r})^2}{n-1}}$, where r_i is the annualized return in year i, $\bar{r}$ is the average annualized return, and n is the number of years.
- **Sharpe ratio:** $S = \frac{r - r_f}{\sigma_a}$, where r_f is the risk-free rate.
- **Max. drawdown** The maximal percentage loss in portfolio value.

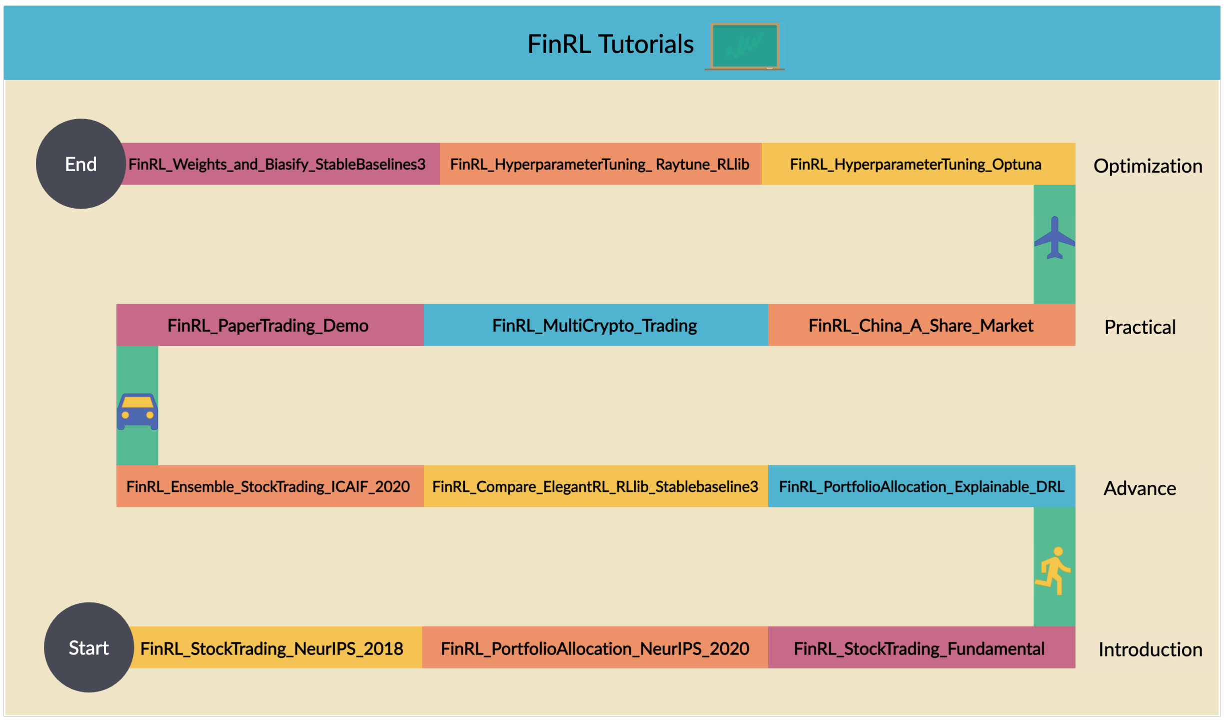
10.2 Experiment Settings

CHAPTER

ELEVEN

TUTORIALS GUIDE

Welcome to FinRL's tutorial! In this section, you can walk through the tutorial notebooks we prepared. If you are new to FinRL, we would suggest you the following sequence:



Mission: provide user-friendly demos in notebook or python.

Outline

1-Introduction: basic demos for beginners.

2-Advance: advanced demos, e.g., ensemble stock trading.

3-Practical: paper trading and live trading.

4-Optimization: hyperparameter tuning.

5-Others: other demos.

11.1 1-Introduction

11.1.1 Single Stock Trading

Deep Reinforcement Learning for Stock Trading from Scratch: Single Stock Trading

Tip: Run the code step by step at Google Colab.

Step 1: Preparation

Step 1.1: Overview

As deep reinforcement learning (DRL) has been recognized as an effective approach in quantitative finance, getting hands-on experiences is attractive to beginners. However, to train a practical DRL trading agent that decides where to trade, at what price, and what quantity involves error-prone and arduous development and debugging.

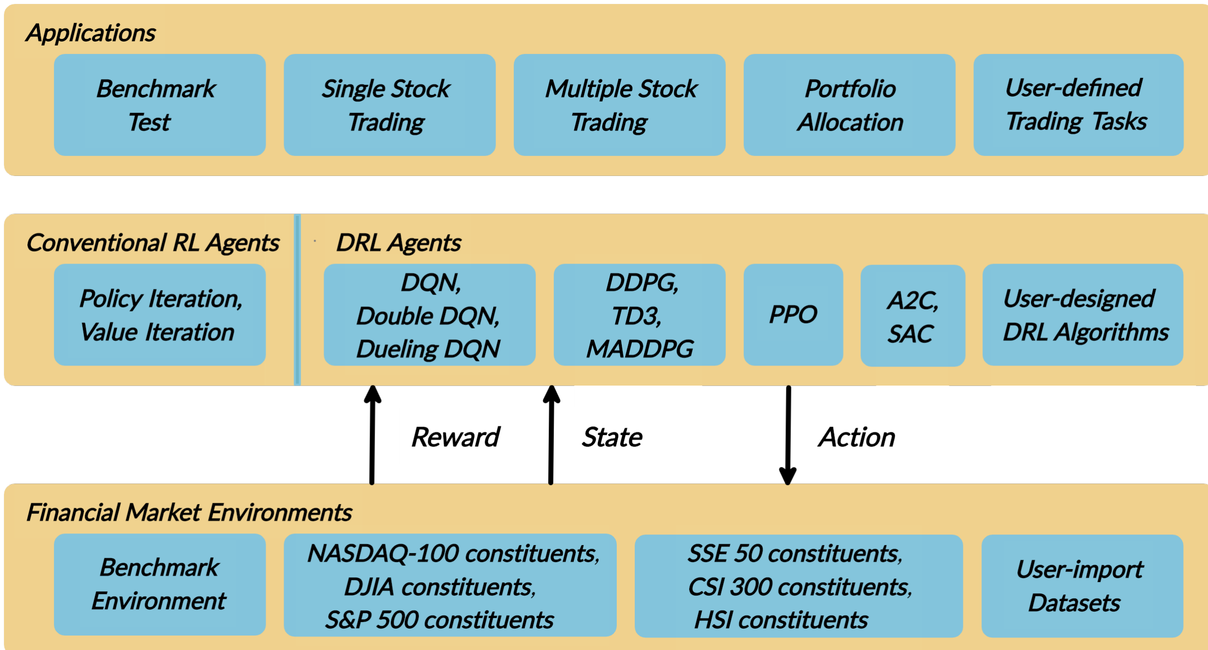
We introduce a DRL library FinRL that facilitates beginners to expose themselves to quantitative finance and to develop their own stock trading strategies. Along with easily-reproducible tutorials, FinRL library allows users to streamline their own developments and to compare with existing schemes easily.

FinRL is a beginner-friendly library with fine-tuned standard DRL algorithms. It has been developed under three primary principles:

- Completeness: Our library shall cover components of the DRL framework completely, which is a fundamental requirement;
- Hands-on tutorials: We aim for a library that is friendly to beginners. Tutorials with detailed walk-through will help users to explore the functionalities of our library;
- Reproducibility: Our library shall guarantee reproducibility to ensure the transparency and also provide users with confidence in what they have done

This article is focusing on one of the use cases in our paper: Single Stock Trading. We use one Jupyter notebook to include all the necessary steps.

We use Apple Inc. stock: AAPL as an example throughout this article, because it is one of the most popular stocks.



Step 1.2: Problem Definition

This problem is to design an automated trading solution for single stock trading. We model the stock trading process as a Markov Decision Process (MDP). We then formulate our trading goal as a maximization problem.

The components of the reinforcement learning environment are:

- Action: The action space describes the allowed actions that the agent interacts with the environment. Normally, a A includes three actions: a {1, 0, 1}, where 1, 0, 1 represent selling, holding, and buying one stock. Also, an action can be carried upon multiple shares. We use an action space {k, . . . , 1, 0, 1, . . . , k}, where k denotes the number of shares. For example, "Buy 10 shares of AAPL" or "Sell 10 shares of AAPL" are 10 or 10, respectively
- Reward function: r(s, a, s) is the incentive mechanism for an agent to learn a better action. The change of the portfolio value when action a is taken at state s and arriving at new state s', i.e., r(s, a, s) = v v, where v and v represent the portfolio values at state s and s, respectively
- State: The state space describes the observations that the agent receives from the environment. Just as a human trader needs to analyze various information before executing a trade, so our trading agent observes many different features to better learn in an interactive environment.
- Environment: single stock trading for AAPL

The data of the single stock that we will be using for this case study is obtained from Yahoo Finance API. The data contains Open-High-Low-Close price and volume.

Step 1.3: Python Package Installation

As a first step we check if the additional packages needed are present, if not install them.

- Yahoo Finance API
- pandas
- matplotlib
- stockstats
- OpenAI gym

- stable-baselines
- tensorflow

```
import pkg_resources
import pip
installedPackages = {pkg.key for pkg in pkg_resources.working_set}
required = {'yfinance', 'pandas', 'matplotlib', 'stockstats','stable-baselines','gym',
↪'tensorflow'}
missing = required - installedPackages
if missing:
    !pip install yfinance
    !pip install pandas
    !pip install matplotlib
    !pip install stockstats
    !pip install gym
    !pip install stable-baselines[mpi]
    !pip install tensorflow==1.15.4
```

Step 1.4: Import packages

```
import yfinance as yf
from stockstats import StockDataFrame as Sdf

import pandas as pd
import matplotlib.pyplot as plt

import gym
from stable_baselines import PPO2, DDPG, A2C, ACKTR, TD3
from stable_baselines import DDPG
from stable_baselines import A2C
from stable_baselines import SAC
from stable_baselines.common.vec_env import DummyVecEnv
from stable_baselines.common.policies import MlpPolicy
```

Step 2: Download Data

Yahoo Finance is a website that provides stock data, financial news, financial reports, etc. All the data provided by Yahoo Finance is free.

This Medium blog explains how to use Yahoo Finance API to extract data directly in Python.

- FinRL uses a class YahooDownloader to fetch data from Yahoo Finance API
- Call Limit: Using the Public API (without authentication), you are limited to 2,000 requests per hour per IP (or up to a total of 48,000 requests a day).

We can either download the stock data like open-high-low-close price manually by entering a stock ticker symbol like AAPL into the website search bar, or we just use Yahoo Finance API to extract data automatically.

FinRL uses a YahooDownloader class to extract data.

```
class YahooDownloader:
    """
    Provides methods for retrieving daily stock data from Yahoo Finance API
```

(continues on next page)

(continued from previous page)

```
    Attributes
    ----------
        start_date : str
            start date of the data (modified from config.py)
        end_date : str
            end date of the data (modified from config.py)
        ticker_list : list
            a list of stock tickers (modified from config.py)

    Methods
    -------
        fetch_data()
            Fetches data from yahoo API
    """
```

Download and save the data in a pandas DataFrame:

```
# Download and save the data in a pandas DataFrame:
df = YahooDownloader(start_date = '2009-01-01',
                          end_date = '2020-09-30',
                          ticker_list = config_tickers.DOW_30_TICKER).fetch_data()

print(df.sort_values(['date','tic'],ignore_index=True).head(30))
```

image/single_1.png

Step 3: Preprocess Data

Data preprocessing is a crucial step for training a high quality machine learning model. We need to check for missing data and do feature engineering in order to convert the data into a model-ready state.

- FinRL uses a FeatureEngineer class to preprocess the data
- Add technical indicators. In practical trading, various information needs to be taken into account, for example the historical stock prices, current holding shares, technical indicators, etc.

Calculate technical indicators

In practical trading, various information needs to be taken into account, for example the historical stock prices, current holding shares, technical indicators, etc.

- FinRL uses stockstats to calcualte technical indicators such as Moving Average Convergence Divergence (MACD), Relative Strength Index (RSI), Average Directional Index (ADX), Commodity Channel Index (CCI) and other various indicators and stats.
- stockstats: supplies a wrapper StockDataFrame based on the pandas.DataFrame with inline stock statistics/indicators support.
- we store the stockstats technical indicator column names in config.py
- config.INDICATORS = ['macd', 'rsi_30', 'cci_30', 'dx_30']

- User can add more technical indicators, check https://github.com/jealous/stockstats for different names

FinRL uses a FeatureEngineer class to preprocess data.

```
class FeatureEngineer:
    """
    Provides methods for preprocessing the stock price data

    Attributes
    ----------
        df: DataFrame
            data downloaded from Yahoo API
        feature_number : int
            number of features we used
        use_technical_indicator : boolean
            we technical indicator or not
        use_turbulence : boolean
            use turbulence index or not

    Methods
    -------
        preprocess_data()
            main method to do the feature engineering
    """
```

Perform Feature Engineering:

```
# Perform Feature Engineering:
df = FeatureEngineer(df.copy(),
                     use_technical_indicator=True,
                     tech_indicator_list = config.INDICATORS,
                     use_turbulence=True,
                     user_defined_feature = False).preprocess_data()
```

Step 4: Build Environment

Considering the stochastic and interactive nature of the automated stock trading tasks, a financial task is modeled as a Markov Decision Process (MDP) problem. The training process involves observing stock price change, taking an action and reward's calculation to have the agent adjusting its strategy accordingly. By interacting with the environment, the trading agent will derive a trading strategy with the maximized rewards as time proceeds.

Our trading environments, based on OpenAI Gym framework, simulate live stock markets with real market data according to the principle of time-driven simulation.

Environment design is one of the most important part in DRL, because it varies a lot from applications to applications and from markets to markets. We can't use an environment for stock trading to trade bitcoin, and vice versa.

The action space describes the allowed actions that the agent interacts with the environment. Normally, action a includes three actions: {-1, 0, 1}, where -1, 0, 1 represent selling, holding, and buying one share. Also, an action can be carried upon multiple shares. We use an action space {-k,…,-1, 0, 1, …, k}, where k denotes the number of shares to buy and -k denotes the number of shares to sell. For example, "Buy 10 shares of AAPL" or "Sell 10 shares of AAPL" are 10 or -10, respectively. The continuous action space needs to be normalized to [-1, 1], since the policy is defined on a Gaussian distribution, which needs to be normalized and symmetric.

In this article, I set k=200, the entire action space is 200*2+1 = 401 for AAPL.

FinRL uses a EnvSetup class to setup environment.

```
class EnvSetup:

    """
    Provides methods for retrieving daily stock data from
    Yahoo Finance API

    Attributes
    ----------
        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
        tech_indicator_list: list
            a list of technical indicator names (modified from config.py)
    Methods
    -------
        fetch_data()
            Fetches data from yahoo API
    """
```

Initialize an environment class:

```
# Initialize env:
env_setup = EnvSetup(stock_dim = stock_dimension,
                     state_space = state_space,
                     hmax = 100,
                     initial_amount = 1000000,
                     transaction_cost_pct = 0.001,
                     tech_indicator_list = config.INDICATORS)

env_train = env_setup.create_env_training(data = train,
                                          env_class = StockEnvTrain)
```

User-defined Environment: a simulation environment class.

FinRL provides blueprint for single stock trading environment.

```
class SingleStockEnv(gym.Env):
    """
    A single stock trading environment for OpenAI gym

    Attributes
    ----------
        df: DataFrame
            input data
        stock_dim : int
```

(continues on next page)

(continued from previous page)

```
        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

Methods
-------
    _sell_stock()
        perform sell action based on the sign of the action
    _buy_stock()
        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 account value at each time step
    save_action_memory()
        return actions/positions at each time step
"""
```

Tutorial for how to design a customized trading environment will be pulished in the future soon.

Step 5: Implement DRL Algorithms

The implementation of the DRL algorithms are based on OpenAI Baselines and Stable Baselines. Stable Baselines is a fork of OpenAI Baselines, with a major structural refactoring, and code cleanups.

Tip: FinRL library includes fine-tuned standard DRL algorithms, such as DQN, DDPG, Multi-Agent DDPG, PPO, SAC, A2C and TD3. We also allow users to design their own DRL algorithms by adapting these DRL algorithms.

Algorithms	Input	Output	Type	State-action spaces support	Finance use cases support	Features and Improvements	Advantages
DQN	States	Q-value	Value based	Discrete only	Single stock trading	Target network, experience replay	Simple and easy to use
Double DQN	States	Q-value	Value based	Discrete only	Single stock trading	Use two identical neural network models to learn	Reduce overestimations
Dueling DQN	States	Q-value	Value based	Discrete only	Single stock trading	Add a specialized dueling Q head	Better differentiate actions, improves the learning
DDPG	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Being deep Q-learning for continuous action spaces	Better at handling high-dimensional continuous action spaces
A2C	State action pair	Q-value	Actor-critic based	Discrete and continuous	All use cases	Advantage function, parallel gradients updating	Stable, cost-effective, faster and works better with large batch sizes
PPO	State action pair	Q-value	Actor-critic based	Discrete and continuous	All use cases	Clipped surrogate objective function	Improve stability, less variance, simply to implement
SAC	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Entropy regularization, exploration-exploitation trade-off	Improve stability
TD3	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Clipped double Q-Learning, delayed policy update, target policy smoothing.	Improve DDPG performance
MADDPG	State action pair	Q-value	Actor-critic based	Continuous only	Multiple stock trading, portfolio allocation	Handle multi-agent RL problem	Improve stability and performance

FinRL uses a DRLAgent class to implement the algorithms.

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

    Attributes
    ----------
        env: gym environment class
             user-defined class
    Methods
    -------
        train_PPO()
            the implementation for PPO algorithm
        train_A2C()
            the implementation for A2C algorithm
        train_DDPG()
            the implementation for DDPG algorithm
        train_TD3()
            the implementation for TD3 algorithm
        DRL_prediction()
            make a prediction in a test dataset and get results
    """
```

Step 6: Model Training

We use 5 DRL models in this article, namely PPO, A2C, DDPG, SAC and TD3. I introduced these models in the previous article. TD3 is an improvement over DDPG.

Tensorboard: reward and loss function plot We use tensorboard integration for hyperparameter tuning and model picking. Tensorboard generates nice looking charts.

Once the learn function is called, you can monitor the RL agent during or after the training, with the following bash command:

```
# cd to the tensorboard_log folder, run the following command
tensorboard --logdir ./A2C_20201127-19h01/
# you can also add past logging folder
tensorboard --logdir ./a2c_tensorboard/;./ppo2_tensorboard/
```

Total rewards for each of the algorithm:

image/single_2.png

total_timesteps (int): the total number of samples to train on. It is one of the most important hyperparameters, there are also other important parameters such as learning rate, batch size, buffer size, etc.

To compare these algorithms, I set the total_timesteps = 100k. If we set the total_timesteps too large, then we will face a risk of overfitting.

By observing the episode_reward chart, we can see that these algorithms will converge to an optimal policy eventually as the step grows. TD3 converges very fast.

actor_loss for DDPG and policy_loss for TD3:

image/single_3.png

image/single_4.png

Picking models

We pick the TD3 model, because it converges pretty fast and it's a state of the art model over DDPG. By observing the episode_reward chart, TD3 doesn't need to reach full 100k total_timesteps to converge.

Four models: PPO A2C, DDPG, TD3

Model 1: PPO

```
#tensorboard --logdir ./single_stock_tensorboard/
env_train = DummyVecEnv([lambda: SingleStockEnv(train)])
model_ppo = PPO2('MlpPolicy', env_train, tensorboard_log="./single_stock_trading_2_tensorboard/")
model_ppo.learn(total_timesteps=100000,tb_log_name="run_aapl_ppo")
#model.save('AAPL_ppo_100k')
```

Model 2: DDPG

```
#tensorboard --logdir ./single_stock_tensorboard/
env_train = DummyVecEnv([lambda: SingleStockEnv(train)])
model_ddpg = DDPG('MlpPolicy', env_train, tensorboard_log="./single_stock_trading_2_tensorboard/")
model_ddpg.learn(total_timesteps=100000, tb_log_name="run_aapl_ddpg")
#model.save('AAPL_ddpg_50k')
```

Model 3: A2C

```
#tensorboard --logdir ./single_stock_tensorboard/
env_train = DummyVecEnv([lambda: SingleStockEnv(train)])
model_a2c = A2C('MlpPolicy', env_train, tensorboard_log="./single_stock_trading_2_tensorboard/")
model_a2c.learn(total_timesteps=100000,tb_log_name="run_aapl_a2c")
#model.save('AAPL_a2c_50k')
```

Model 4: TD3

```
#tensorboard --logdir ./single_stock_tensorboard/
#DQN<DDPG<TD3
env_train = DummyVecEnv([lambda: SingleStockEnv(train)])
model_td3 = TD3('MlpPolicy', env_train, tensorboard_log="./single_stock_trading_2_tensorboard/")
model_td3.learn(total_timesteps=100000,tb_log_name="run_aapl_td3")
#model.save('AAPL_td3_50k')
```

Testing data

```
test = data_clean[(data_clean.datadate>='2019-01-01') ]
# the index needs to start from 0
test=test.reset_index(drop=True)
```

Trading

Assume that we have $100,000 initial capital at 2019-01-01. We use the TD3 model to trade AAPL.

```
model = model_td3
env_test = DummyVecEnv([lambda: SingleStockEnv(test)])
obs_test = env_test.reset()
print("==============Model Prediction===========")
for i in range(len(test.index.unique())):
    action, _states = model.predict(obs_test)
    obs_test, rewards, dones, info = env_test.step(action)
    env_test.render()
```

```
# create trading env
env_trade, obs_trade = env_setup.create_env_trading(data = trade,
                                        env_class = StockEnvTrade,
                                         turbulence_threshold=250)
## make a prediction and get the account value change
df_account_value = DRLAgent.DRL_prediction(model=model_sac,
                                            test_data = trade,
                                            test_env = env_trade,
                                            test_obs = obs_trade)
```

image/single_5.png

Step 7: Backtest Our Strategy

Backtesting plays a key role in evaluating the performance of a trading strategy. Automated backtesting tool is preferred because it reduces the human error. We usually use the **`Quantopian pyfolio`_** package to backtest our trading strategies. It is easy to use and consists of various individual plots that provide a comprehensive image of the performance of a trading strategy.

For simplicity purposes, in the article, we just calculate the Sharpe ratio and the annual return manually.

```
def get_DRL_sharpe():
    df_total_value=pd.read_csv('account_value.csv',index_col=0)
    df_total_value.columns = ['account_value']
    df_total_value['daily_return']=df_total_value.pct_change(1)
    sharpe = (252**0.5)*df_total_value['daily_return'].mean()/ \
    df_total_value['daily_return'].std()

    annual_return = ((df_total_value['daily_return'].mean()+1)**252-1)*100
    print("annual return: ", annual_return)
    print("sharpe ratio: ", sharpe)
    return df_total_value


def get_buy_and_hold_sharpe(test):
    test['daily_return']=test['adjcp'].pct_change(1)
    sharpe = (252**0.5)*test['daily_return'].mean()/ \
    test['daily_return'].std()
    annual_return = ((test['daily_return'].mean()+1)**252-1)*100
    print("annual return: ", annual_return)

    print("sharpe ratio: ", sharpe)
    #return sharpe
```

11.1.2 Multiple Stock Trading

Deep Reinforcement Learning for Stock Trading from Scratch: Multiple Stock Trading

Tip: Run the code step by step at Google Colab.

Step 1: Preparation

Step 1.1: Overview

To begin with, I would like explain the logic of multiple stock trading using Deep Reinforcement Learning.

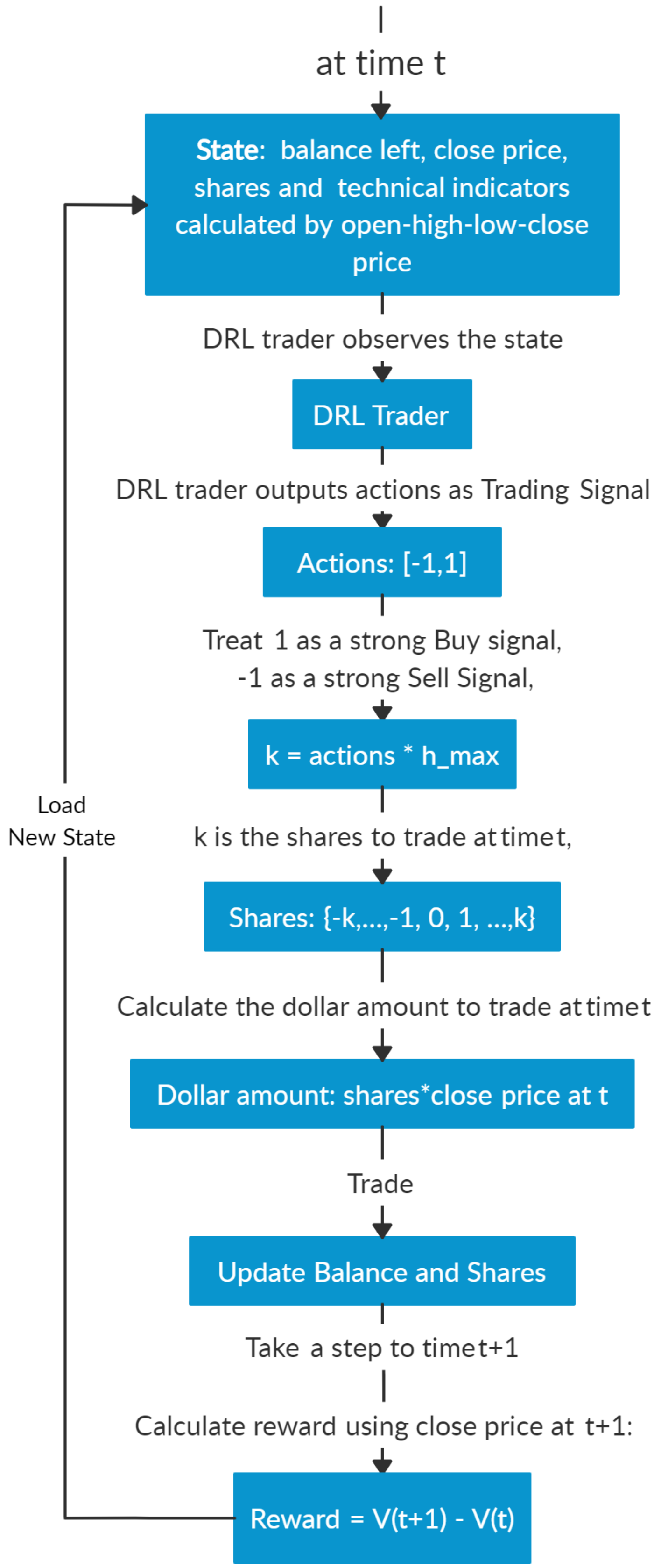
We use Dow 30 constituents as an example throughout this article, because those are the most popular stocks.

A lot of people are terrified by the word "Deep Reinforcement Learning", actually, you can just treat it as a "Smart AI" or "Smart Stock Trader" or "R2-D2 Trader" if you want, and just use it.

Suppose that we have a well trained DRL agent "DRL Trader", we want to use it to trade multiple stocks in our portfolio.

- Assume we are at time t, at the end of day at time t, we will know the open-high-low-close price of the Dow 30 constituents stocks. We can use these information to calculate technical indicators such as MACD, RSI, CCI, ADX. In Reinforcement Learning we call these data or features as "states".
- We know that our portfolio value V(t) = balance (t) + dollar amount of the stocks (t).
- We feed the states into our well trained DRL Trader, the trader will output a list of actions, the action for each stock is a value within [-1, 1], we can treat this value as the trading signal, 1 means a strong buy signal, -1 means a strong sell signal.
- We calculate k = actions *h_max, h_max is a predefined parameter that sets as the maximum amount of shares to trade. So we will have a list of shares to trade.
- The dollar amount of shares = shares to trade* close price (t).
- Update balance and shares. These dollar amount of shares are the money we need to trade at time t. The updated balance = balance (t) amount of money we pay to buy shares +amount of money we receive to sell shares. The updated shares = shares held (t) shares to sell +shares to buy.
- So we take actions to trade based on the advice of our DRL Trader at the end of day at time t (time t's close price equals time t+1's open price). We hope that we will benefit from these actions by the end of day at time t+1.
- Take a step to time t+1, at the end of day, we will know the close price at t+1, the dollar amount of the stocks (t+1)= sum(updated shares * close price (t+1)). The portfolio value V(t+1)=balance (t+1) + dollar amount of the stocks (t+1).
- So the step reward by taking the actions from DRL Trader at time t to t+1 is r = v(t+1) v(t). The reward can be positive or negative in the training stage. But of course, we need a positive reward in trading to say that our DRL Trader is effective.
- Repeat this process until termination.

Below are the logic chart of multiple stock trading and a made-up example for demonstration purpose:


at time t
State: balance left, close price, shares and technical indicators calculated by open-high-low-close price
DRL trader observes the state
DRL Trader
DRL trader outputs actions as Trading Signal
Actions: [-1,1]
Treat 1 as a strong Buy signal, -1 as a strong Sell Signal,
k = actions * h_max
Load New State
k is the shares to trade at time t,
Shares: {-k,...,-1, 0, 1, ...,k}
Calculate the dollar amount to trade at time t
Dollar amount: shares*close price at t
Trade
Update Balance and Shares
Take a step to time t+1
Calculate reward using close price at t+1:
Reward = V(t+1) - V(t)

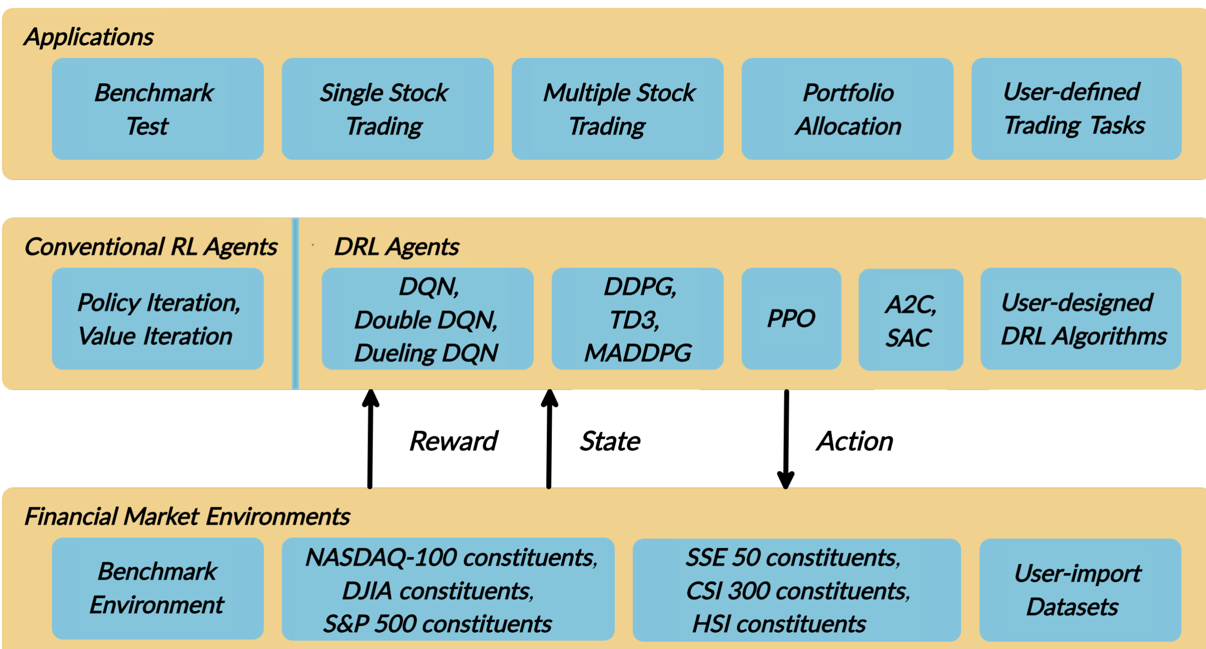
image/multiple_2.png

Multiple stock trading is different from single stock trading because as the number of stocks increase, the dimension of the data will increase, the state and action space in reinforcement learning will grow exponentially. So stability and reproducibility are very essential here.

We introduce a DRL library FinRL that facilitates beginners to expose themselves to quantitative finance and to develop their own stock trading strategies.

FinRL is characterized by its reproducibility, scalability, simplicity, applicability and extendibility.

This article is focusing on one of the use cases in our paper: Mutiple Stock Trading. We use one Jupyter notebook to include all the necessary steps.



Step 1.2: Problem Definition

This problem is to design an automated solution for stock trading. We model the stock trading process as a Markov Decision Process (MDP). We then formulate our trading goal as a maximization problem. The algorithm is trained using Deep Reinforcement Learning (DRL) algorithms and the components of the reinforcement learning environment are:

- Action: The action space describes the allowed actions that the agent interacts with the environment. Normally, a A includes three actions: a {1, 0, 1}, where 1, 0, 1 represent selling, holding, and buying one stock. Also, an action can be carried upon multiple shares. We use an action space {k, …, 1, 0, 1, …, k}, where k denotes the number of shares. For example, "Buy 10 shares of AAPL" or "Sell 10 shares of AAPL" are 10 or 10, respectively
- Reward function: r(s, a, s) is the incentive mechanism for an agent to learn a better action. The change of the portfolio value when action a is taken at state s and arriving at new state s', i.e., r(s, a, s) = v v, where v and v represent the portfolio values at state s and s, respectively

- State: The state space describes the observations that the agent receives from the environment. Just as a human trader needs to analyze various information before executing a trade, so our trading agent observes many different features to better learn in an interactive environment.
- Environment: Dow 30 constituents

The data of the stocks for this case study is obtained from Yahoo Finance API. The data contains Open-High-Low-Close price and volume.

Step 1.3: FinRL installation

```
## install finrl library
!pip install git+https://github.com/AI4Finance-LLC/FinRL-Library.git
```

Then we import the packages needed for this demonstration.

Step 1.4: Import packages

```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
# matplotlib.use('Agg')
import datetime

%matplotlib inline
from finrl import config
from finrl import config_tickers
from finrl.finrl_meta.preprocessor.yahoodownloader import YahooDownloader
from finrl.finrl_meta.preprocessor.preprocessors import FeatureEngineer, data_split
from finrl.finrl_meta.env_stock_trading.env_stocktrading import StockTradingEnv
from finrl.agents.stablebaselines3.models import DRLAgent

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

import sys
sys.path.append("../FinRL-Library")

import itertools
```

Finally, create folders for storage.

Step 1.5: Create folders

```
import os
if not os.path.exists("./" + config.DATA_SAVE_DIR):
    os.makedirs("./" + config.DATA_SAVE_DIR)
if not os.path.exists("./" + config.TRAINED_MODEL_DIR):
    os.makedirs("./" + config.TRAINED_MODEL_DIR)
if not os.path.exists("./" + config.TENSORBOARD_LOG_DIR):
    os.makedirs("./" + config.TENSORBOARD_LOG_DIR)
if not os.path.exists("./" + config.RESULTS_DIR):
    os.makedirs("./" + config.RESULTS_DIR)
```

Then all the preparation work are done. We can start now!

Step 2: Download Data

Before training our DRL agent, we need to get the historical data of DOW30 stocks first. Here we use the data from Yahoo! Finance. Yahoo! Finance is a website that provides stock data, financial news, financial reports, etc. All the data provided by Yahoo Finance is free. yfinance is an open-source library that provides APIs to download data from Yahoo! Finance. We will use this package to download data here.

FinRL uses a YahooDownloader class to extract data.

```
class YahooDownloader:
    """
    Provides methods for retrieving daily stock data from Yahoo Finance API

    Attributes
    ----------
        start_date : str
            start date of the data (modified from config.py)
        end_date : str
            end date of the data (modified from config.py)
        ticker_list : list
            a list of stock tickers (modified from config.py)

    Methods
    -------
        fetch_data()
            Fetches data from yahoo API
    """
```

Download and save the data in a pandas DataFrame:

```
# Download and save the data in a pandas DataFrame:
df = YahooDownloader(start_date = '2009-01-01',
                          end_date = '2020-09-30',
                          ticker_list = config_tickers.DOW_30_TICKER).fetch_data()

print(df.sort_values(['date','tic'],ignore_index=True).head(30))
```

image/multiple_3.png

Step 3: Preprocess Data

Data preprocessing is a crucial step for training a high quality machine learning model. We need to check for missing data and do feature engineering in order to convert the data into a model-ready state.

Step 3.1: Check missing data

```
# check missing data
dow_30.isnull().values.any()
```

Step 3.2: Add technical indicators

In practical trading, various information needs to be taken into account, for example the historical stock prices, current holding shares, technical indicators, etc. In this article, we demonstrate two trend-following technical indicators: MACD and RSI.

```
def add_technical_indicator(df):
        """
        calcualte technical indicators
        use stockstats package to add technical inidactors
        :param data: (df) pandas dataframe
        :return: (df) pandas dataframe
        """
        stock = Sdf.retype(df.copy())
        stock['close'] = stock['adjcp']
        unique_ticker = stock.tic.unique()

        macd = pd.DataFrame()
        rsi = pd.DataFrame()

        #temp = stock[stock.tic == unique_ticker[0]]['macd']
        for i in range(len(unique_ticker)):
            ## macd
            temp_macd = stock[stock.tic == unique_ticker[i]]['macd']
            temp_macd = pd.DataFrame(temp_macd)
            macd = macd.append(temp_macd, ignore_index=True)
            ## rsi
            temp_rsi = stock[stock.tic == unique_ticker[i]]['rsi_30']
            temp_rsi = pd.DataFrame(temp_rsi)
            rsi = rsi.append(temp_rsi, ignore_index=True)

        df['macd'] = macd
        df['rsi'] = rsi
        return df
```

Step 3.3: Add turbulence index

Risk-aversion reflects whether an investor will choose to preserve the capital. It also influences one's trading strategy when facing different market volatility level.

To control the risk in a worst-case scenario, such as financial crisis of 2007–2008, FinRL employs the financial turbulence index that measures extreme asset price fluctuation.

```
def add_turbulence(df):
    """
    add turbulence index from a precalcualted dataframe
    :param data: (df) pandas dataframe
    :return: (df) pandas dataframe
    """
    turbulence_index = calcualte_turbulence(df)
    df = df.merge(turbulence_index, on='datadate')
    df = df.sort_values(['datadate','tic']).reset_index(drop=True)
    return df



def calcualte_turbulence(df):
```

(continues on next page)

(continued from previous page)

```
    """calculate turbulence index based on dow 30"""
    # can add other market assets

    df_price_pivot=df.pivot(index='datadate', columns='tic', values='adjcp')
    unique_date = df.datadate.unique()
    # start after a year
    start = 252
    turbulence_index = [0]*start
    #turbulence_index = [0]
    count=0
    for i in range(start,len(unique_date)):
        current_price = df_price_pivot[df_price_pivot.index == unique_date[i]]
        hist_price = df_price_pivot[[n in unique_date[0:i] for n in df_price_pivot.index ]]
        cov_temp = hist_price.cov()
        current_temp=(current_price - np.mean(hist_price,axis=0))
        temp = current_temp.values.dot(np.linalg.inv(cov_temp)).dot(current_temp.values.T)
        if temp>0:
            count+=1
            if count>2:
                turbulence_temp = temp[0][0]
            else:
                #avoid large outlier because of the calculation just begins
                turbulence_temp=0
        else:
            turbulence_temp=0
        turbulence_index.append(turbulence_temp)


    turbulence_index = pd.DataFrame({'datadate':df_price_pivot.index,
                                     'turbulence':turbulence_index})
    return turbulence_index
```

Step 3.4 Feature Engineering

FinRL uses a FeatureEngineer class to preprocess data.

Perform Feature Engineering:

```
# Perform Feature Engineering:
df = FeatureEngineer(df.copy(),
                     use_technical_indicator=True,
                     tech_indicator_list = config.INDICATORS,
                     use_turbulence=True,
                     user_defined_feature = False).preprocess_data()
```

image/multiple_4.png

Step 4: Design Environment

Considering the stochastic and interactive nature of the automated stock trading tasks, a financial task is modeled as a Markov Decision Process (MDP) problem. The training process involves observing stock price change, taking an action and reward's calculation to have the agent adjusting its strategy accordingly. By interacting with the environment, the trading agent will derive a trading strategy with the maximized rewards as time proceeds.

Our trading environments, based on OpenAI Gym framework, simulate live stock markets with real market data according to the principle of time-driven simulation.

The action space describes the allowed actions that the agent interacts with the environment. Normally, action a includes three actions: {-1, 0, 1}, where -1, 0, 1 represent selling, holding, and buying one share. Also, an action can be carried upon multiple shares. We use an action space {-k,…,-1, 0, 1, …, k}, where k denotes the number of shares to buy and -k denotes the number of shares to sell. For example, "Buy 10 shares of AAPL" or "Sell 10 shares of AAPL" are 10 or -10, respectively. The continuous action space needs to be normalized to [-1, 1], since the policy is defined on a Gaussian distribution, which needs to be normalized and symmetric.

Step 4.1: Environment for Training

```python
## Environment for Training
import numpy as np
import pandas as pd
from gym.utils import seeding
import gym
from gym import spaces
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

# shares normalization factor
# 100 shares per trade
HMAX_NORMALIZE = 100
# initial amount of money we have in our account
INITIAL_ACCOUNT_BALANCE=1000000
# total number of stocks in our portfolio
STOCK_DIM = 30
# transaction fee: 1/1000 reasonable percentage
TRANSACTION_FEE_PERCENT = 0.001

REWARD_SCALING = 1e-4


class StockEnvTrain(gym.Env):
    """A stock trading environment for OpenAI gym"""
    metadata = {'render.modes': ['human']}

    def __init__(self, df,day = 0):
        #super(StockEnv, self).__init__()
        self.day = day
        self.df = df

        # action_space normalization and shape is STOCK_DIM
        self.action_space = spaces.Box(low = -1, high = 1,shape = (STOCK_DIM,))
        # Shape = 181: [Current Balance]+[prices 1-30]+[owned shares 1-30]
        # +[macd 1-30]+ [rsi 1-30] + [cci 1-30] + [adx 1-30]
```

(continues on next page)

(continued from previous page)

```
        self.observation_space = spaces.Box(low=0, high=np.inf, shape = (121,))
        # load data from a pandas dataframe
        self.data = self.df.loc[self.day,:]
        self.terminal = False
        # initalize state
        self.state = [INITIAL_ACCOUNT_BALANCE] + \
                      self.data.adjcp.values.tolist() + \
                      [0]*STOCK_DIM + \
                      self.data.macd.values.tolist() + \
                      self.data.rsi.values.tolist()
                      #self.data.cci.values.tolist() + \
                      #self.data.adx.values.tolist()
        # initialize reward
        self.reward = 0
        self.cost = 0
        # memorize all the total balance change
        self.asset_memory = [INITIAL_ACCOUNT_BALANCE]
        self.rewards_memory = []
        self.trades = 0
        self._seed()

    def _sell_stock(self, index, action):
        # perform sell action based on the sign of the action
        if self.state[index+STOCK_DIM+1] > 0:
            #update balance
            self.state[0] += \
            self.state[index+1]*min(abs(action),self.state[index+STOCK_DIM+1]) * \
             (1- TRANSACTION_FEE_PERCENT)

            self.state[index+STOCK_DIM+1] -= min(abs(action), self.state[index+STOCK_DIM+1])
            self.cost +=self.state[index+1]*min(abs(action),self.state[index+STOCK_DIM+1]) * \
             TRANSACTION_FEE_PERCENT
            self.trades+=1
        else:
            pass

    def _buy_stock(self, index, action):
        # perform buy action based on the sign of the action
        available_amount = self.state[0] // self.state[index+1]
        # print('available_amount:{}'.format(available_amount))

        #update balance
        self.state[0] -= self.state[index+1]*min(available_amount, action)* \
                          (1+ TRANSACTION_FEE_PERCENT)

        self.state[index+STOCK_DIM+1] += min(available_amount, action)

        self.cost+=self.state[index+1]*min(available_amount, action)* \
                          TRANSACTION_FEE_PERCENT
        self.trades+=1
```

(continues on next page)

(continued from previous page)

```

    def step(self, actions):
        # print(self.day)
        self.terminal = self.day >= len(self.df.index.unique())-1
        # print(actions)

        if self.terminal:
            plt.plot(self.asset_memory,'r')
            plt.savefig('account_value_train.png')
            plt.close()
            end_total_asset = self.state[0]+ \
            sum(np.array(self.state[1:(STOCK_DIM+1)])*np.array(self.state[(STOCK_
↪DIM+1):(STOCK_DIM*2+1)]))
            print("previous_total_asset:{}".format(self.asset_memory[0]))

            print("end_total_asset:{}".format(end_total_asset))
            df_total_value = pd.DataFrame(self.asset_memory)
            df_total_value.to_csv('account_value_train.csv')
            print("total_reward:{}".format(self.state[0]+sum(np.array(self.
↪state[1:(STOCK_DIM+1)])*np.array(self.state[(STOCK_DIM+1):61]))- INITIAL_ACCOUNT_
↪BALANCE ))
            print("total_cost: ", self.cost)
            print("total_trades: ", self.trades)
            df_total_value.columns = ['account_value']
            df_total_value['daily_return']=df_total_value.pct_change(1)
            sharpe = (252**0.5)*df_total_value['daily_return'].mean()/ \
                  df_total_value['daily_return'].std()
            print("Sharpe: ",sharpe)
            print("=================================")
            df_rewards = pd.DataFrame(self.rewards_memory)
            df_rewards.to_csv('account_rewards_train.csv')

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

        else:
            actions = actions * HMAX_NORMALIZE

            begin_total_asset = self.state[0]+ \
            sum(np.array(self.state[1:(STOCK_DIM+1)])*np.array(self.state[(STOCK_
↪DIM+1):61]))
            #print("begin_total_asset:{}".format(begin_total_asset))

            argsort_actions = np.argsort(actions)

            sell_index = argsort_actions[:np.where(actions < 0)[0].shape[0]]
            buy_index = argsort_actions[::-1][:np.where(actions > 0)[0].shape[0]]

            for index in sell_index:
                # print('take sell action'.format(actions[index]))
                self._sell_stock(index, actions[index])

            for index in buy_index:
```

(continues on next page)

(continued from previous page)

```
                # print('take buy action: {}'.format(actions[index]))
                self._buy_stock(index, actions[index])

            self.day += 1
            self.data = self.df.loc[self.day,:]
            #load next state
            # print("stock_shares:{}".format(self.state[29:]))
            self.state =  [self.state[0]] + \
                    self.data.adjcp.values.tolist() + \
                    list(self.state[(STOCK_DIM+1):61]) + \
                    self.data.macd.values.tolist() + \
                    self.data.rsi.values.tolist()

            end_total_asset = self.state[0]+ \
            sum(np.array(self.state[1:(STOCK_DIM+1)])*np.array(self.state[(STOCK_
↪DIM+1):61]))

            #print("end_total_asset:{}".format(end_total_asset))

            self.reward = end_total_asset - begin_total_asset
            self.rewards_memory.append(self.reward)

            self.reward = self.reward * REWARD_SCALING
            # print("step_reward:{}".format(self.reward))

            self.asset_memory.append(end_total_asset)


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

    def reset(self):
        self.asset_memory = [INITIAL_ACCOUNT_BALANCE]
        self.day = 0
        self.data = self.df.loc[self.day,:]
        self.cost = 0
        self.trades = 0
        self.terminal = False
        self.rewards_memory = []
        #initiate state
        self.state = [INITIAL_ACCOUNT_BALANCE] + \
                      self.data.adjcp.values.tolist() + \
                      [0]*STOCK_DIM + \
                      self.data.macd.values.tolist() + \
                      self.data.rsi.values.tolist()
        return self.state

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

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

Step 4.2: Environment for Trading

```python
## Environment for Trading
import numpy as np
import pandas as pd
from gym.utils import seeding
import gym
from gym import spaces
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

# shares normalization factor
# 100 shares per trade
HMAX_NORMALIZE = 100
# initial amount of money we have in our account
INITIAL_ACCOUNT_BALANCE=1000000
# total number of stocks in our portfolio
STOCK_DIM = 30
# transaction fee: 1/1000 reasonable percentage
TRANSACTION_FEE_PERCENT = 0.001

# turbulence index: 90-150 reasonable threshold
#TURBULENCE_THRESHOLD = 140
REWARD_SCALING = 1e-4

class StockEnvTrade(gym.Env):
    """A stock trading environment for OpenAI gym"""
    metadata = {'render.modes': ['human']}

    def __init__(self, df,day = 0,turbulence_threshold=140):
        #super(StockEnv, self).__init__()
        #money = 10 , scope = 1
        self.day = day
        self.df = df
        # action_space normalization and shape is STOCK_DIM
        self.action_space = spaces.Box(low = -1, high = 1,shape = (STOCK_DIM,))
        # Shape = 181: [Current Balance]+[prices 1-30]+[owned shares 1-30]
        # +[macd 1-30]+ [rsi 1-30] + [cci 1-30] + [adx 1-30]
        self.observation_space = spaces.Box(low=0, high=np.inf, shape = (121,))
        # load data from a pandas dataframe
        self.data = self.df.loc[self.day,:]
        self.terminal = False
        self.turbulence_threshold = turbulence_threshold
        # initalize state
        self.state = [INITIAL_ACCOUNT_BALANCE] + \
                      self.data.adjcp.values.tolist() + \
                      [0]*STOCK_DIM + \
                      self.data.macd.values.tolist() + \
                      self.data.rsi.values.tolist()

        # initialize reward
        self.reward = 0
```

(continues on next page)

(continued from previous page)

```
        self.turbulence = 0
        self.cost = 0
        self.trades = 0
        # memorize all the total balance change
        self.asset_memory = [INITIAL_ACCOUNT_BALANCE]
        self.rewards_memory = []
        self.actions_memory=[]
        self.date_memory=[]
        self._seed()


    def _sell_stock(self, index, action):
        # perform sell action based on the sign of the action
        if self.turbulence<self.turbulence_threshold:
            if self.state[index+STOCK_DIM+1] > 0:
                #update balance
                self.state[0] += \
                self.state[index+1]*min(abs(action),self.state[index+STOCK_DIM+1]) * \
                 (1- TRANSACTION_FEE_PERCENT)

                self.state[index+STOCK_DIM+1] -= min(abs(action), self.state[index+STOCK_
↪DIM+1])
                self.cost +=self.state[index+1]*min(abs(action),self.state[index+STOCK_
↪DIM+1]) * \
                 TRANSACTION_FEE_PERCENT
                self.trades+=1
            else:
                pass
        else:
            # if turbulence goes over threshold, just clear out all positions
            if self.state[index+STOCK_DIM+1] > 0:
                #update balance
                self.state[0] += self.state[index+1]*self.state[index+STOCK_DIM+1]* \
                              (1- TRANSACTION_FEE_PERCENT)
                self.state[index+STOCK_DIM+1] =0
                self.cost += self.state[index+1]*self.state[index+STOCK_DIM+1]* \
                              TRANSACTION_FEE_PERCENT
                self.trades+=1
            else:
                pass

    def _buy_stock(self, index, action):
        # perform buy action based on the sign of the action
        if self.turbulence< self.turbulence_threshold:
            available_amount = self.state[0] // self.state[index+1]
            # print('available_amount:{}'.format(available_amount))

            #update balance
            self.state[0] -= self.state[index+1]*min(available_amount, action)* \
                              (1+ TRANSACTION_FEE_PERCENT)

            self.state[index+STOCK_DIM+1] += min(available_amount, action)
```

(continues on next page)

(continued from previous page)

```

            self.cost+=self.state[index+1]*min(available_amount, action)* \
                              TRANSACTION_FEE_PERCENT
            self.trades+=1
        else:
            # if turbulence goes over threshold, just stop buying
            pass

    def step(self, actions):
        # print(self.day)
        self.terminal = self.day >= len(self.df.index.unique())-1
        # print(actions)

        if self.terminal:
            plt.plot(self.asset_memory,'r')
            plt.savefig('account_value_trade.png')
            plt.close()

            df_date = pd.DataFrame(self.date_memory)
            df_date.columns = ['datadate']
            df_date.to_csv('df_date.csv')


            df_actions = pd.DataFrame(self.actions_memory)
            df_actions.columns = self.data.tic.values
            df_actions.index = df_date.datadate
            df_actions.to_csv('df_actions.csv')

            df_total_value = pd.DataFrame(self.asset_memory)
            df_total_value.to_csv('account_value_trade.csv')
            end_total_asset = self.state[0]+ \
            sum(np.array(self.state[1:(STOCK_DIM+1)])*np.array(self.state[(STOCK_
↪DIM+1):(STOCK_DIM*2+1)]))
            print("previous_total_asset:{}".format(self.asset_memory[0]))

            print("end_total_asset:{}".format(end_total_asset))
            print("total_reward:{}".format(self.state[0]+sum(np.array(self.
↪state[1:(STOCK_DIM+1)])*np.array(self.state[(STOCK_DIM+1):61]))- self.asset_memory[0]␣
↪))
            print("total_cost: ", self.cost)
            print("total trades: ", self.trades)

            df_total_value.columns = ['account_value']
            df_total_value['daily_return']=df_total_value.pct_change(1)
            sharpe = (252**0.5)*df_total_value['daily_return'].mean()/ \
                  df_total_value['daily_return'].std()
            print("Sharpe: ",sharpe)

            df_rewards = pd.DataFrame(self.rewards_memory)
            df_rewards.to_csv('account_rewards_trade.csv')

            # print('total asset: {}'.format(self.state[0]+ sum(np.array(self.
```

(continues on next page)

(continued from previous page)

```
↪state[1:29])*np.array(self.state[29:]))))
            #with open('obs.pkl', 'wb') as f:
            #    pickle.dump(self.state, f)

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

        else:
            # print(np.array(self.state[1:29]))
            self.date_memory.append(self.data.datadate.unique())

            #print(self.data)
            actions = actions * HMAX_NORMALIZE
            if self.turbulence>=self.turbulence_threshold:
                actions=np.array([-HMAX_NORMALIZE]*STOCK_DIM)
            self.actions_memory.append(actions)

            #actions = (actions.astype(int))

            begin_total_asset = self.state[0]+ \
            sum(np.array(self.state[1:(STOCK_DIM+1)])*np.array(self.state[(STOCK_
↪DIM+1):(STOCK_DIM*2+1)]))
            #print("begin_total_asset:{}".format(begin_total_asset))

            argsort_actions = np.argsort(actions)
            #print(argsort_actions)

            sell_index = argsort_actions[:np.where(actions < 0)[0].shape[0]]
            buy_index = argsort_actions[::-1][:np.where(actions > 0)[0].shape[0]]

            for index in sell_index:
                # print('take sell action'.format(actions[index]))
                self._sell_stock(index, actions[index])

            for index in buy_index:
                # print('take buy action: {}'.format(actions[index]))
                self._buy_stock(index, actions[index])

            self.day += 1
            self.data = self.df.loc[self.day,:]
            self.turbulence = self.data['turbulence'].values[0]
            #print(self.turbulence)
            #load next state
            # print("stock_shares:{}".format(self.state[29:]))
            self.state =  [self.state[0]] + \
                    self.data.adjcp.values.tolist() + \
                    list(self.state[(STOCK_DIM+1):(STOCK_DIM*2+1)]) + \
                    self.data.macd.values.tolist() + \
                    self.data.rsi.values.tolist()

            end_total_asset = self.state[0]+ \
            sum(np.array(self.state[1:(STOCK_DIM+1)])*np.array(self.state[(STOCK_
↪DIM+1):(STOCK_DIM*2+1)]))
```

(continues on next page)

(continued from previous page)

```
            #print("end_total_asset:{}".format(end_total_asset))

            self.reward = end_total_asset - begin_total_asset
            self.rewards_memory.append(self.reward)

            self.reward = self.reward * REWARD_SCALING

            self.asset_memory.append(end_total_asset)

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

    def reset(self):
        self.asset_memory = [INITIAL_ACCOUNT_BALANCE]
        self.day = 0
        self.data = self.df.loc[self.day,:]
        self.turbulence = 0
        self.cost = 0
        self.trades = 0
        self.terminal = False
        #self.iteration=self.iteration
        self.rewards_memory = []
        self.actions_memory=[]
        self.date_memory=[]
        #initiate state
        self.state = [INITIAL_ACCOUNT_BALANCE] + \
                      self.data.adjcp.values.tolist() + \
                      [0]*STOCK_DIM + \
                      self.data.macd.values.tolist() + \
                      self.data.rsi.values.tolist()

        return self.state

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


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

Step 5: Implement DRL Algorithms

The implementation of the DRL algorithms are based on OpenAI Baselines and Stable Baselines. Stable Baselines is a fork of OpenAI Baselines, with a major structural refactoring, and code cleanups.

Step 5.1: Training data split: 2009-01-01 to 2018-12-31

```
def data_split(df,start,end):
    """
    split the dataset into training or testing using date
```

(continues on next page)

(continued from previous page)

```
    :param data: (df) pandas dataframe, start, end
    :return: (df) pandas dataframe
    """
    data = df[(df.datadate >= start) & (df.datadate < end)]
    data=data.sort_values(['datadate','tic'],ignore_index=True)
    data.index = data.datadate.factorize()[0]
    return data
```

Step 5.2: Model training: DDPG

```
## tensorboard --logdir ./multiple_stock_tensorboard/
# add noise to the action in DDPG helps in learning for better exploration
n_actions = env_train.action_space.shape[-1]
param_noise = None
action_noise = OrnsteinUhlenbeckActionNoise(mean=np.zeros(n_actions), sigma=float(0.5) * np.ones(n_actions))

# model settings
model_ddpg = DDPG('MlpPolicy',
                   env_train,
                   batch_size=64,
                   buffer_size=100000,
                   param_noise=param_noise,
                   action_noise=action_noise,
                   verbose=0,
                   tensorboard_log="./multiple_stock_tensorboard/")

## 250k timesteps: took about 20 mins to finish
model_ddpg.learn(total_timesteps=250000, tb_log_name="DDPG_run_1")
```

Step 5.3: Trading

Assume that we have $1,000,000 initial capital at 2019-01-01. We use the DDPG model to trade Dow jones 30 stocks.

Step 5.4: Set turbulence threshold

Set the turbulence threshold to be the 99% quantile of insample turbulence data, if current turbulence index is greater than the threshold, then we assume that the current market is volatile

```
insample_turbulence = dow_30[(dow_30.datadate<'2019-01-01') & (dow_30.datadate>='2009-01-01')]
insample_turbulence = insample_turbulence.drop_duplicates(subset=['datadate'])
```

Step 5.5: Prepare test data and environment

```
# test data
test = data_split(dow_30, start='2019-01-01', end='2020-10-30')
# testing env
env_test = DummyVecEnv([lambda: StockEnvTrade(test, turbulence_threshold=insample_turbulence_threshold)])
obs_test = env_test.reset()
```

Step 5.6: Prediction

```
def DRL_prediction(model, data, env, obs):
    print("==============Model Prediction===========")
    for i in range(len(data.index.unique())):
        action, _states = model.predict(obs)
        obs, rewards, dones, info = env.step(action)
        env.render()
```

Step 6: Backtest Our Strategy

For simplicity purposes, in the article, we just calculate the Sharpe ratio and the annual return manually.

```
def backtest_strat(df):
    strategy_ret= df.copy()
    strategy_ret['Date'] = pd.to_datetime(strategy_ret['Date'])
    strategy_ret.set_index('Date', drop = False, inplace = True)
    strategy_ret.index = strategy_ret.index.tz_localize('UTC')
    del strategy_ret['Date']
    ts = pd.Series(strategy_ret['daily_return'].values, index=strategy_ret.index)
    return ts
```

Step 6.1: Dow Jones Industrial Average

```
def get_buy_and_hold_sharpe(test):
    test['daily_return']=test['adjcp'].pct_change(1)
    sharpe = (252**0.5)*test['daily_return'].mean()/ \
    test['daily_return'].std()
    annual_return = ((test['daily_return'].mean()+1)**252-1)*100
    print("annual return: ", annual_return)

    print("sharpe ratio: ", sharpe)
    #return sharpe
```

Step 6.2: Our DRL trading strategy

```
def get_daily_return(df):
    df['daily_return']=df.account_value.pct_change(1)
    #df=df.dropna()
    sharpe = (252**0.5)*df['daily_return'].mean()/ \
    df['daily_return'].std()

    annual_return = ((df['daily_return'].mean()+1)**252-1)*100
    print("annual return: ", annual_return)
    print("sharpe ratio: ", sharpe)
    return df
```

Step 6.3: Plot the results using Quantopian pyfolio

Backtesting plays a key role in evaluating the performance of a trading strategy. Automated backtesting tool is preferred because it reduces the human error. We usually use the Quantopian pyfolio package to backtest our trading strategies. It is easy to use and consists of various individual plots that provide a comprehensive image of the performance of a trading strategy.

```
%matplotlib inline
with pyfolio.plotting.plotting_context(font_scale=1.1):
    pyfolio.create_full_tear_sheet(returns = DRL_strat,
                                   benchmark_rets=dow_strat, set_context=False)
```

11.1.3 Portfolio Allocation

Our paper: FinRL: A Deep Reinforcement Learning Library for Automated Stock Trading in Quantitative Finance.

Presented at NeurIPS 2020: Deep RL Workshop.

The Jupyter notebook codes are available on our Github and Google Colab.

Tip:

- FinRL Single Stock Trading at Google Colab.
- FinRL Multiple Stocks Trading at Google Colab:

Check our previous tutorials: Single Stock Trading and Multiple Stock Trading for detailed explanation of the FinRL architecture and modules.

Overview

To begin with, we would like to explain the logic of portfolio allocation using Deep Reinforcement Learning.We use Dow 30 constituents as an example throughout this article, because those are the most popular stocks.

Let's say that we got a million dollars at the beginning of 2019. We want to invest this $1,000,000 into stock markets, in this case is Dow Jones 30 constituents.Assume that no margin, no short sale, no treasury bill (use all the money to trade only these 30 stocks). So that the weight of each individual stock is non-negative, and the weights of all the stocks add up to one.

We hire a smart portfolio manager- Mr. Deep Reinforcement Learning. Mr. DRL will give us daily advice includes the portfolio weights or the proportions of money to invest in these 30 stocks. So every day we just need to rebalance the portfolio weights of the stocks.The basic logic is as follows.

tutorial/image/portfolio_allocation_1.png

Portfolio allocation is different from multiple stock trading because we are essentially rebalancing the weights at each time step, and we have to use all available money.

The traditional and the most popular way of doing portfolio allocation is mean-variance or modern portfolio theory (MPT):

image/portfolio_allocation_2.png

However, MPT performs not so well in out-of-sample data. MPT is calculated only based on stock returns, if we want to take other relevant factors into account, for example some of the technical indicators like MACD or RSI, MPT may not be able to combine these information together well.

We introduce a DRL library FinRL that facilitates beginners to expose themselves to quantitative finance. FinRL is a DRL library designed specifically for automated stock trading with an effort for educational and demonstrative purpose.

This article is focusing on one of the use cases in our paper: Portfolio Allocation. We use one Jupyter notebook to include all the necessary steps.

Problem Definition

This problem is to design an automated trading solution for portfolio allocation. We model the stock trading process as a Markov Decision Process (MDP). We then formulate our trading goal as a maximization problem.

The components of the reinforcement learning environment are:

- **Action**: portfolio weight of each stock is within [0,1]. We use softmax function to normalize the actions to sum to 1.
- **State: {Covariance Matrix, MACD, RSI, CCI, ADX}, **state space** shape is (34, 30). 34 is the number of rows, 30 is the number of columns.
- **Reward function**: r(s, a, s) = p_t, p_t is the cumulative portfolio value.
- **Environment**: portfolio allocation for Dow 30 constituents.

Covariance matrix is a good feature because portfolio managers use it to quantify the risk (standard deviation) associated with a particular portfolio.

We also assume no transaction cost, because we are trying to make a simple portfolio allocation case as a starting point.

Load Python Packages

Install the unstable development version of FinRL:

```
# Install the unstable development version in Jupyter notebook:
!pip install git+https://github.com/AI4Finance-LLC/FinRL-Library.git
```

Import Packages:

```
# import packages
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
matplotlib.use('Agg')
import datetime

from finrl import config
from finrl import config_tickers
from finrl.marketdata.yahoodownloader import YahooDownloader
from finrl.preprocessing.preprocessors import FeatureEngineer
from finrl.preprocessing.data import data_split
from finrl.env.environment import EnvSetup
from finrl.env.EnvMultipleStock_train import StockEnvTrain
from finrl.env.EnvMultipleStock_trade import StockEnvTrade
```

(continues on next page)

(continued from previous page)

```python
from finrl.model.models import DRLAgent
from finrl.trade.backtest import BackTestStats, BaselineStats, BackTestPlot, backtest_strat, baseline_strat
from finrl.trade.backtest import backtest_strat, baseline_strat

import os
if not os.path.exists("./" + config.DATA_SAVE_DIR):
    os.makedirs("./" + config.DATA_SAVE_DIR)
if not os.path.exists("./" + config.TRAINED_MODEL_DIR):
    os.makedirs("./" + config.TRAINED_MODEL_DIR)
if not os.path.exists("./" + config.TENSORBOARD_LOG_DIR):
    os.makedirs("./" + config.TENSORBOARD_LOG_DIR)
if not os.path.exists("./" + config.RESULTS_DIR):
    os.makedirs("./" + config.RESULTS_DIR)
```

Download Data

FinRL uses a YahooDownloader class to extract data.

```python
class YahooDownloader:
    """
    Provides methods for retrieving daily stock data from Yahoo Finance API

    Attributes
    ----------
        start_date : str
            start date of the data (modified from config.py)
        end_date : str
            end date of the data (modified from config.py)
        ticker_list : list
            a list of stock tickers (modified from config.py)

    Methods
    -------
        fetch_data()
            Fetches data from yahoo API
    """
```

Download and save the data in a pandas DataFrame:

```python
# Download and save the data in a pandas DataFrame:
df = YahooDownloader(start_date = '2008-01-01',
                     end_date = '2020-12-01',
                     ticker_list = config_tickers.DOW_30_TICKER).fetch_data()
```

Preprocess Data

FinRL uses a FeatureEngineer class to preprocess data.

```
class FeatureEngineer:
    """
    Provides methods for preprocessing the stock price data

    Attributes
    ----------
        df: DataFrame
            data downloaded from Yahoo API
        feature_number : int
            number of features we used
        use_technical_indicator : boolean
            we technical indicator or not
        use_turbulence : boolean
            use turbulence index or not

    Methods
    -------
        preprocess_data()
            main method to do the feature engineering
    """
```

Perform Feature Engineering: covariance matrix + technical indicators:

```
# Perform Feature Engineering:
df = FeatureEngineer(df.copy(),
                    use_technical_indicator=True,
                    use_turbulence=False).preprocess_data()


# add covariance matrix as states
df=df.sort_values(['date','tic'],ignore_index=True)
df.index = df.date.factorize()[0]

cov_list = []
# look back is one year
lookback=252
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()
  covs = return_lookback.cov().values
  cov_list.append(covs)

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

image/portfolio_allocation_3.png

Build Environment

FinRL uses a EnvSetup class to setup environment.

```
class EnvSetup:
    """
    Provides methods for retrieving daily stock data from
    Yahoo Finance API

    Attributes
        ----------
        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
        tech_indicator_list: list
            a list of technical indicator names (modified from config.py)
    Methods
        -------
        create_env_training()
            create env class for training
        create_env_validation()
            create env class for validation
        create_env_trading()
            create env class for trading
    """
```

Initialize an environment class:

User-defined Environment: a simulation environment class.The environment for portfolio allocation:

```
import numpy as np
import pandas as pd
from gym.utils import seeding
import gym
from gym import spaces
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

```

(continues on next page)

(continued from previous page)

```
class StockPortfolioEnv(gym.Env):
    """A single stock trading 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
    Methods
    -------
    _sell_stock()
        perform sell action based on the sign of the action
    _buy_stock()
        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 account value at each time step
    save_action_memory()
        return actions/positions at each time step

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

    def __init__(self,
                df,
                stock_dim,
                hmax,
                initial_amount,
```

(continues on next page)

(continued from previous page)

```
                transaction_cost_pct,
                reward_scaling,
                state_space,
                action_space,
                tech_indicator_list,
                turbulence_threshold,
                lookback=252,
                day = 0):
        #super(StockEnv, self).__init__()
        #money = 10 , scope = 1
        self.day = day
        self.lookback=lookback
        self.df = df
        self.stock_dim = stock_dim
        self.hmax = hmax
        self.initial_amount = initial_amount
        self.transaction_cost_pct =transaction_cost_pct
        self.reward_scaling = reward_scaling
        self.state_space = state_space
        self.action_space = action_space
        self.tech_indicator_list = tech_indicator_list

        # action_space normalization and shape is self.stock_dim
        self.action_space = spaces.Box(low = 0, high = 1,shape = (self.action_space,))
        # Shape = (34, 30)
        # covariance matrix + technical indicators
        self.observation_space = spaces.Box(low=0,
                                            high=np.inf,
                                            shape = (self.state_space+len(self.tech_indicator_list),
                                                     self.state_space))

        # load data from a pandas dataframe
        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 self.tech_indicator_list ], axis=0)
        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]
        # memorize portfolio return each step
        self.portfolio_return_memory = [0]
        self.actions_memory=[[1/self.stock_dim]*self.stock_dim]
        self.date_memory=[self.data.date.unique()[0]]


```

(continues on next page)

(continued from previous page)

```
    def step(self, actions):
        # print(self.day)
        self.terminal = self.day >= len(self.df.index.unique())-1
        # print(actions)

        if self.terminal:
            df = pd.DataFrame(self.portfolio_return_memory)
            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']
            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,{}

        else:
            #print(actions)
            # actions are the portfolio weight
            # normalize to sum of 1
            norm_actions = (np.array(actions) - np.array(actions).min()) / (np.array(actions) - np.array(actions).min()).sum()
            weights = norm_actions
            #print(weights)
            self.actions_memory.append(weights)
            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 self.tech_indicator_list ], axis=0)
            # calcualte portfolio return
            # individual stocks' return * weight
            portfolio_return = sum(((self.data.close.values / last_day_memory.close.values)-1)*weights)
            # update portfolio value
```

(continues on next page)

(continued from previous page)

```
            new_portfolio_value = self.portfolio_value*(1+portfolio_return)
            self.portfolio_value = new_portfolio_value

            # save into memory
            self.portfolio_return_memory.append(portfolio_return)
            self.date_memory.append(self.data.date.unique()[0])
            self.asset_memory.append(new_portfolio_value)

            # the reward is the new portfolio value or end portfolo value
            self.reward = new_portfolio_value
            #self.reward = self.reward*self.reward_scaling


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

    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.tech_indicator_list ], axis=0)
        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

    def save_asset_memory(self):
        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):
        # 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)
```

(continues on next page)

(continued from previous page)

```python
        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

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

Implement DRL Algorithms

FinRL uses a DRLAgent class to implement the algorithms.

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

    Attributes
    ----------
        env: gym environment class
             user-defined class
    Methods
    -------
        train_PPO()
            the implementation for PPO algorithm
        train_A2C()
            the implementation for A2C algorithm
        train_DDPG()
            the implementation for DDPG algorithm
        train_TD3()
            the implementation for TD3 algorithm
        DRL_prediction()
            make a prediction in a test dataset and get results
    """
```

Model Training:

We use A2C for portfolio allocation, because it is stable, cost-effective, faster and works better with large batch sizes.

Trading:Assume that we have $1,000,000 initial capital at 2019/01/01. We use the A2C model to perform portfolio allocation of the Dow 30 stocks.

```python
trade = data_split(df,'2019-01-01', '2020-12-01')

env_trade, obs_trade = env_setup.create_env_trading(data = trade,
                                         env_class = StockPortfolioEnv)

df_daily_return, df_actions = DRLAgent.DRL_prediction(model=model_a2c,
                        test_data = trade,
                        test_env = env_trade,
                        test_obs = obs_trade)
```

image/portfolio_allocation_4.png

The output actions or the portfolio weights look like this:

image/portfolio_allocation_5.png

Backtesting Performance

FinRL uses a set of functions to do the backtesting with Quantopian pyfolio.

```
from pyfolio import timeseries
DRL_strat = backtest_strat(df_daily_return)
perf_func = timeseries.perf_stats
perf_stats_all = perf_func( returns=DRL_strat,
                              factor_returns=DRL_strat,
                                positions=None, transactions=None, turnover_denom="AGB")
print("==============DRL Strategy Stats===========")
perf_stats_all
print("==============Get Index Stats===========")
baesline_perf_stats=BaselineStats('^DJI',
                                  baseline_start = '2019-01-01',
                                  baseline_end = '2020-12-01')


# plot
dji, dow_strat = baseline_strat('^DJI','2019-01-01','2020-12-01')
import pyfolio
%matplotlib inline
with pyfolio.plotting.plotting_context(font_scale=1.1):
        pyfolio.create_full_tear_sheet(returns = DRL_strat,
                                       benchmark_rets=dow_strat, set_context=False)
```

The left table is the stats for backtesting performance, the right table is the stats for Index (DJIA) performance.

Plots:

11.2 2-Advance

11.3 3-Practical

11.4 4-Optimization

11.5 5-Others

CHAPTER

TWELVE

FILE ARCHITECTURE

FinRL's file architecture strictly follow the *Three-layer Architecture*.

```
FinRL
├── finrl (the main folder)
    ├── applications
            ├── cryptocurrency_trading
            ├── high_frequency_trading
            ├── portfolio_allocation
            └── stock_trading
    ├── agents
            ├── elegantrl
            ├── rllib
            └── stablebaseline3
    ├── finrl_meta
            ├── data_processors
            ├── env_cryptocurrency_trading
            ├── env_portfolio_allocation
            ├── env_stock_trading
            ├── preprocessor
            ├── data_processor.py
            └── finrl_meta_config.py
    ├── config.py
    ├── config_tickers.py
    ├── main.py
    ├── train.py
    ├── test.py
    ├── trade.py
    └── plot.py
```

CHAPTER

THIRTEEN

DEVELOPMENT SETUP WITH PYCHARM

This setup with pycharm makes it easy to work on all of AI4Finance-Foundation's repositories simultaneously, while allowing easy debugging, committing to the respective repo and creating PRs/MRs.

13.1 Step 1: Download Software

-Download and install Anaconda.

-Download and install PyCharm. The Community Edition (free version) offers everything you need except running Jupyter notebooks. The Full-fledged Professional Edition offers everything. A workaround to run existing notebooks in the Community edition is to copy all notebook cells into .py files. For notebook support, you can consider PyCharm Professional Edition.

-On GitHub, fork FinRL to your private Github repo.

-On GitHub, fork ElegantRL to your private Github repo.

-On GitHub, fork FinRL-Meta to your private Github repo.

-All next steps happen on your local computer.

13.2 Step 2: Git Clone

```
mkdir ~/ai4finance
cd ~/ai4finance
git clone https://github.com/[your_github_username]/FinRL.git
git clone https://github.com/[your_github_username]/ElegantRL.git
git clone https://github.com/[your_github_username]/FinRL-Meta.git
```

13.3 Step 3: Create a Conda Environment

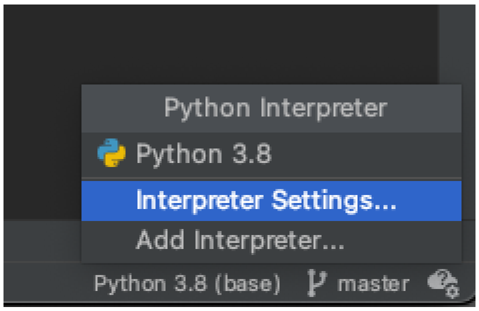
```
cd ~/ai4finance
conda create --name ai4finance python=3.8
conda activate ai4finance

cd FinRL
pip install -r requirements.txt
```

Install ElegantRL using requirements.txt, or open ElegantRL/setup.py in a text editor and pip install anything you can find: gym, matplotlib, numpy, pybullet, torch, opencv-python, and box2d-py.

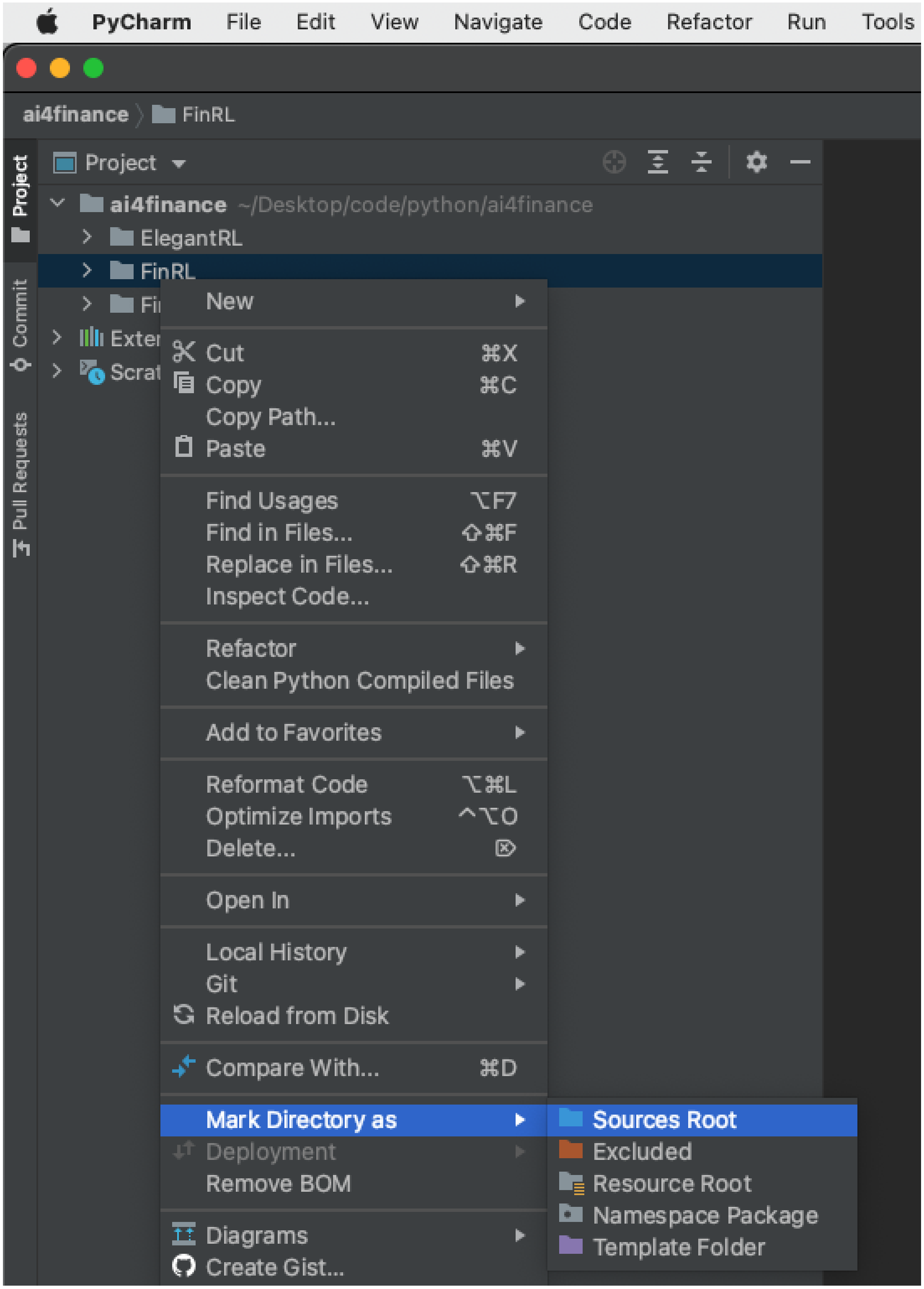
13.4 Step 4: Configure a PyCharm Project

-Launch PyCharm

-File > Open > [ai4finance project folder]



-At the bottom right of the status bar, change or add the interpreter to the ai4finance conda environment. Make sure when you click the “terminal” bar at the bottom left, it shows ai4finance.



-At the left of the screen, in the project file tree:

- Right-click on the FinRL folder > Mark Directory as > Sources Root
- Right-click on the ElegantRL folder > Mark Directory as > Sources Root
- Right-click on the FinRL-Meta folder > Mark Directory as > Sources Root

-Once you run a .py file, you will notice that you may still have some missing packages. In that case, simply pip install them.

For example, we revise FinRL.

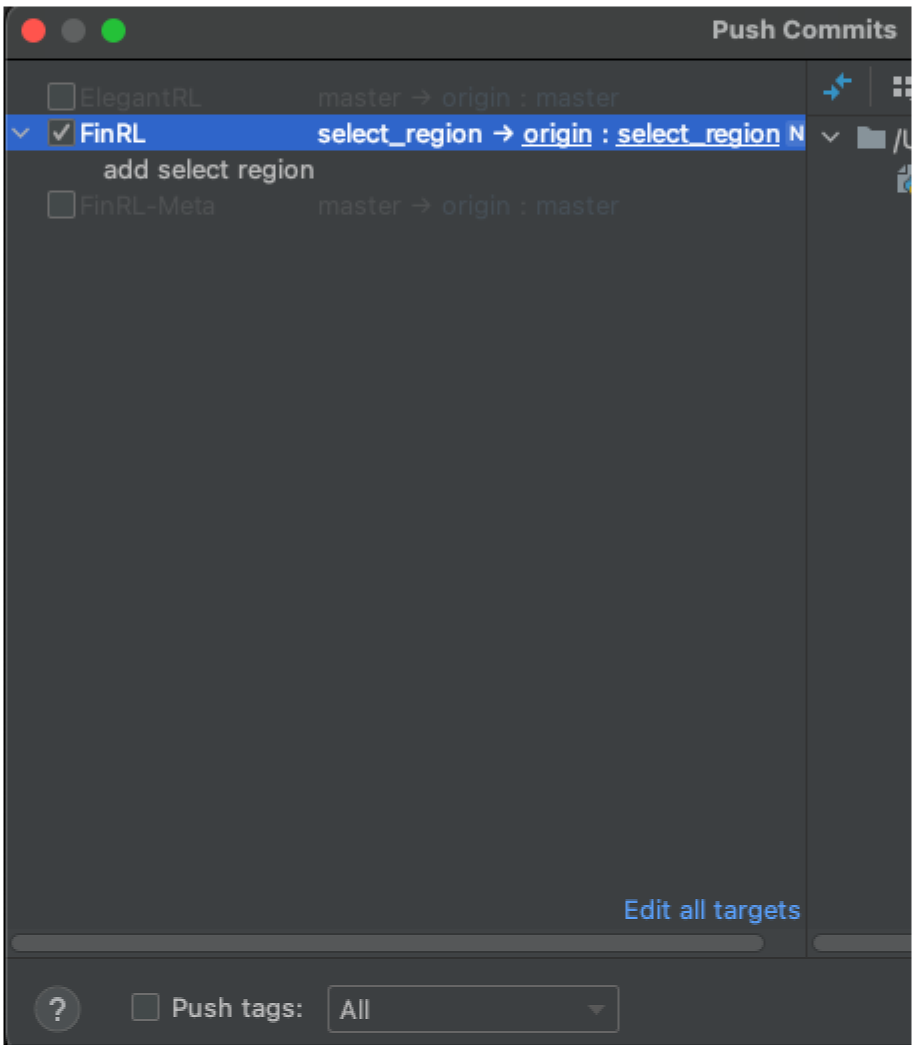
```
cd ~/ai4finance
cd ./FinRL
git checkout -b branch_xxx
```

where branch_xxx is a new branch name. In this branch, we revise config.py.

13.5 Step 5: Creating Commits and PRs/MRs

-Create commits as you usually do through PyCharm.

-Make sure that each commit covers only 1 of the 3 repo's. Don't create a commit that spans more than one repo, e.g., FinRL and ElegantRL.



-When you do a Git Push, PyCharm will ask you to which of the 3 repos you want to push. Just like the above figure, we select the repo "FinRL".

With respect to creating a pull request (PR) or merge quest (MR), please refer to Create a PR or Opensource Create a PR.

CHAPTER

FOURTEEN

PUBLICATIONS

Papers by the Columbia research team can be found at Google Scholar.

Table 1: Publications

Title	Conference	Link	Citations	Year
FinRL-Meta: A Universe of Near-Real Market Environments for Data-Driven Deep Reinforcement Learning in Quantitative Finance	NeurIPS 2021 Data-Centric AI Workshop	paper, code	2	2021
Explainable deep reinforcement learning for portfolio management: An empirical approach	ICAIF 2021: ACM International Conference on AI in Finance	paper, code	1	2021
FinRL-Podracer: High performance and scalable deep reinforcement learning for quantitative finance	ICAIF 2021: ACM International Conference on AI in Finance	paper, code	2	2021
FinRL: Deep reinforcement learning framework to automate trading in quantitative finance	ICAIF 2021: ACM International Conference on AI in Finance	paper, code	7	2021
FinRL: A deep reinforcement learning library for automated stock trading in quantitative finance	NeurIPS 2020 Deep RL Workshop	paper, code	25	2020
Deep reinforcement learning for automated stock trading: An ensemble strategy	ICAIF 2020: ACM International Conference on AI in Finance	paper, code	44	2020
Multi-agent reinforcement learning for liquidation strategy analysis	ICML 2019 Workshop on AI in Finance: Applications and Infrastructure for Multi-Agent Learning	paper, code	19	2019
Practical deep reinforcement learning approach for stock trading	NeurIPS 2018 Workshop on Challenges and Opportunities for AI in Financial Services	paper, code	86	2018

CHAPTER FIFTEEN

EXTERNAL SOURCES

The following contents are collected and referred by AI4Finance community during the development of FinRL and related projects. Some of them are educational and relatively easy while some others are professional and need advanced knowledge. We appreciate and respect the effort of all these contents' authors and developers.

15.1 Proof-of-concept

[1] FinRL: Deep Reinforcement Learning Framework to Automate Trading in Quantitative Finance Deep reinforcement learning framework to automate trading in quantitative finance, ACM International Conference on AI in Finance, ICAIF 2021.

[2] FinRL: A Deep Reinforcement Learning Library for Automated Stock Trading in Quantitative Finance A deep reinforcement learning library for automated stock trading in quantitative finance, Deep RL Workshop, NeurIPS 2020.

[3] Practical deep reinforcement learning approach for stock trading. NeurIPS Workshop on Challenges and Opportunities for AI in Financial Services: the Impact of Fairness, Explainability, Accuracy, and Privacy, 2018.

[4] Deep Reinforcement Learning for Trading. Zhang, Zihao, Stefan Zohren, and Stephen Roberts. The Journal of Financial Data Science 2, no. 2 (2020): 25-40.

[5] A Deep Reinforcement Learning Framework for the Financial Portfolio Management Problem. Jiang, Zhengyao, Dixing Xu, and Jinjun Liang. arXiv preprint arXiv:1706.10059 (2017).

15.2 DRL Algorithms/Libraries

[1] Documentation of ElegentRL by AI4Finance Foundation.

[2] Spinning Up in Deep RL by OpenAI.

15.3 Theory

[1] Deep Reinforcement Learning: An Overview Li, Yuxi. arXiv preprint arXiv:1701.07274 (2017).

[2] Continuous-time mean–variance portfolio selection: A reinforcement learning framework. Mathematical Finance, 30(4), pp.1273-1308. Wang, H. and Zhou, X.Y., 2020.

[3] Mao Guan and Xiao-Yang Liu. Explainable deep reinforcement learning for portfolio man- agement: An empirical approach. ACM International Conference on AI in Finance, ICAIF 2021.

[4] ICAIF International Conference on AI in Finance.

15.4 Trading Strategies

[1] Deep reinforcement learning for automated stock trading: an ensemble strategy. ACM International Conference on AI in Finance, 2020.

[2] FinRL-Podracer: High performance and scalable deep reinforcement learning for quantitative finance. ACM International Conference on AI in Finance, ICAIF 2021.

[3] Multi-agent reinforcement learning for liquidation strategy analysis, paper and codes. Workshop on Applications and Infrastructure for Multi-Agent Learning, ICML 2019.

[4] Risk-Sensitive Reinforcement Learning: a Martingale Approach to Reward Uncertainty. International Conference on AI in Finance, ICAIF 2020.

[5] Cryptocurrency Trading Using Machine Learning. Journal of Risk and Financial Management, August 2020.

[6] Multi-Agent Reinforcement Learning in a Realistic Limit Order Book Market Simulation. Michaël Karpe, Jin Fang, Zhongyao Ma, Chen Wang. International Conference on AI in Finance (ICAIF'20), September 2020.

[7] Market Making via Reinforcement Learning. Thomas Spooner, John Fearnley, Rahul Savani, Andreas Koukorinis. AAMAS2018 Conference Proceedings

[8] Financial Trading as a Game: A Deep Reinforcement Learning Approach Huang, Chien Yi. arXiv preprint arXiv:1807.02787 (2018).

[9] Deep Hedging: Hedging Derivatives Under Generic Market Frictions Using Reinforcement Learning Buehler, Hans, Lukas Gonon, Josef Teichmann, Ben Wood, Baranidharan Mohan, and Jonathan Kochems. Swiss Finance Institute Research Paper 19-80 (2019).

15.5 Financial Big Data

[1] FinRL-Meta: A Universe of Near-Real Market Environments for Data-Driven Deep Reinforcement Learning in Quantitative Finance. NeurIPS 2021 Data-Centric AI Workshop

15.6 Interpretation and Explainability

[1] Explainable Deep Reinforcement Learning for Portfolio Management: An Empirical Approach. Guan, M. and Liu, X.Y.. ACM International Conference on AI in Finance, 2021.

15.7 Tools or Softwares

[1] FinRL by AI4Finance Foundation.

[2] FinRL-Meta: A Universe of Near-Real Market Environments for Data-Driven Deep Reinforcement Learning in Quantitative Finance, by AI4Finance Foundation.

[3] ElegantRL: a DRL library developed by AI4Finance Foundation.

[4] Stable-Baselines3: Reliable Reinforcement Learning Implementations.

15.8 Survey

[1] Recent Advances in Reinforcement Learning in Finance. Hambly, B., Xu, R. and Yang, H., 2021.

[2] Deep Reinforcement Learning for Trading—A Critical Survey. Adrian Millea, 2021.

[3] Modern Perspectives on Reinforcement Learning in Finance Kolm, Petter N. and Ritter, Gordon. The Journal of Machine Learning in Finance, Vol. 1, No. 1, 2020.

[4] Reinforcement Learning in Economics and Finance Charpentier, Arthur, Romuald Elie, and Carl Remlinger. Computational Economics (2021): 1-38.

[5] Comprehensive Review of Deep Reinforcement Learning Methods and Applications in Economics Mosavi, Amirhosein, Yaser Faghan, Pedram Ghamisi, Puhong Duan, Sina Faizollahzadeh Ardabili, Ely Salwana, and Shahab S. Band. Mathematics 8, no. 10 (2020): 1640.

15.9 Education

[1] Coursera Overview of Advanced Methods of Reinforcement Learning in Finance. By Igor Halperin, at NYU.

[2] *Foundations of reinforcement learning with applications in finance* by Ashwin Rao, Tikhon Jelvis, Stanford University

CHAPTER SIXTEEN

FAQ

Version
0.3

Date
05-29-2022

Contributors
Roberto Fray da Silva, Xiao-Yang Liu, Ziyi Xia, Ming Zhu

This document contains the most frequently asked questions related to FinRL, which are based on questions posted on the slack channels and Github issues.

16.1 Outline

- *1-Inputs and datasets*
- *2-Code and implementation*
- *3-Model evaluation*
- *4-Miscellaneous*
- *5-Common issues/bugs*

16.2 1-Inputs and datasets

- *Not yet. We're developing this functionality*
- *Not yet. We're developing this functionality*
- *Not yet. We're developing this functionality*
- *Not yet*
- *Yahoo Finance (through the yfinance library)*

- *Yahoo Finance (only up to last 7 days), through the yfinance library. It is the only option besides scraping (or paying for a service provider)*
- *No, as this is more of an execution strategy related to risk control. You can use it as part of your system, adding the risk control part as a separate component*
- *Yes, you can add it. Remember to check on the code that this additional feature is being fed to the model (state)*
- *No, you'll have to use a paid service or library/code to scrape news and obtain the sentiment from them (normally, using deep learning and NLP)*

16.3 2-Code and implementation

- *Yes, it does*
- *Yes, because the current parameters are defined for daily data. You'll have to tune the model for intraday trading*
- *Not many yet, but we're working on providing different reward functions and an easy way to set your own reward function*
- *Yes, but none is available at the moment. Sometimes in the literature you'll find this referred to as transfer learning*
- *Each model has its own hyperparameters, but the most important is the total_timesteps (think of it as epochs in a neural network: even if all the other hyperparameters are optimal, with few epochs the model will have a bad performance). The other important hyperparameters, in general, are: learning_rate, batch_size, ent_coef, buffer_size, policy, and reward scaling*
- *There are several, such as: Ray Tune and Optuna. You can start from our examples in the tutorials*
- *We suggest using ElegantRL or Stable Baselines 3. We tested the following models with success: A2C, A3C, DDPG, PPO, SAC, TD3, TRPO. You can also create your own algorithm, with an OpenAI Gym-style market environment*
-

Please update to latest version (https://github.com/AI4Finance-LLC/FinRL-Library), check if the hyperparameters used were not outside a normal range (ex: learning rate too high), and run the code again. If you still have problems, please check Section 2 (What to do when you experience problems)

- **raw-html**

  *What to do when you experience problems? *

1. Check if it is not already answered on this FAQ 2. Check if it is posted on the GitHub repo issues. If not, welcome to submit an issue on GitHub 3. Use the correct channel on the AI4Finance slack or Wechat group.*

- **raw-html**

  *Does anyone know if there is a trading environment for a single stock? There is one in the docs, but the collab link seems to be broken. *

  We did not update the single stock for long time. The performance for single stock is not very good, since the state space is too small so that the agent extract little information from the environment. Please use the multi stock environment, and after training only use the single stock to trade.

16.4 3-Model evaluation

- *Not exactly. Depending on the period, the asset, the model chosen, and the hyperparameters used, BH may be very difficult to beat (it's almost never beaten on stocks/periods with low volatility and steady growth). Nevertheless, update the library and its dependencies (the github repo has the most recent version), and check the example notebook for the specific environment type (single, multi, portfolio optimization) to see if the code is running correctly*

- *We use the Pyfolio backtest library from Quantopian (https://github.com/quantopian/pyfolio), especially the simple tear sheet and its charts. In general, the most important metrics are: annual returns, cumulative returns, annual volatility, sharpe ratio, calmar ratio, stability, and max drawdown*

- *There are several metrics, but we recommend the following, as they are the most used in the market: annual returns, cumulative returns, annual volatility, sharpe ratio, calmar ratio, stability, and max drawdown*

- *We recommend using buy and hold (BH), as it is a strategy that can be followed on any market and tends to provide good results in the long run. You can also compare with other DRL models and trading strategies such as the minimum variance portfolio*

16.5 4-Miscellaneous

-

  *1. Read the documentation from the very beginning 2. Go through * `tutorials <https://github.com/AI4Finance-Foundation/FinRL/tree/master/tutorials> `_ *3. read our papers*

-

  This is available on our Github repo https://github.com/AI4Finance-LLC/FinRL-Library

-

  Participate on the slack channels, check the current issues and the roadmap, and help any way you can (sharing the library with others, testing the library of different markets/models/strategies, contributing with code development, etc)

-

  Please read 1-Inputs and datasets

-

  Please read 4-Miscellaneous

-

  Please check our development roadmap at our Github repo: https://github.com/AI4Finance-LLC/FinRL-Library

-

  FinRL aims for education and demonstration, while FinRL-Meta aims for building financial big data and a metaverse of data-driven financial RL.

16.6 5-Common issues/bugs

- **Package trading_calendars reports errors in Windows system:**
  Trading_calendars is not maintained now. It may report errors in Windows system (python>=3.7). These are two possible solutions: 1). Use python=3.6 environment. 2). Replace trading_calendars with exchange_caldenars.