FinRL Ecosystem: Using Reinforcement Learning to Efficiently Automate Trading

by AI4Finance Foundation

An open-source community sharing AI tools for finance.

Financial Reinforcement Learning by AI4Finance Foundation

"Finance serves a purpose. … Investors are lured to gamble their wealth on wide hunches originated by charlatans and encouraged by mass media. One day in the near future, ML will dominate finance, science will curtail guessing, and investing will not mean gambling."

—-By Marcos Lopes De Prado
Advanced in Financial Machine Learning

Our Mission: *to efficiently automate trading. We continuously develop and share codes for finance.*

Our Vision: *AI community has accumulated an open-source code ocean over the past decade. We believe applying these intellectual and engineering properties to finance will initiate a paradigm shift from the conventional trading routine to an automated machine learning approach, even RLOps in finance.*

Materials:

AI4Finance Foundation:

FinRL, *FinRL-Meta*, and *Website*

ElegantRL and *Website*.

FinRL Ecosystem: *Deep Reinforcement Learning to Automate Trading in Quantitative Finance. Talk at Wolfe Research 5th Annual Virtual Global Quantitative and Macro Investment Conference, Nov. 08, 2021.*

Awesome_DRL4Finance_List: *Awesome Deep Reinforcement Learning in Finance*

Textbooks:

De Prado, M.L., 2018. *Advances in financial machine learning*. John Wiley & Sons.

Assessing this file on Google Doc at:
https://docs.google.com/document/d/1FxfdiwJ8L8xJeObPMIVFxi9ozykC9HujMuxYVYEmR5g/edit

Contributors: (Please add your information)

Jiechao Gao
Xiao-Yang Liu
Bruce Yang
Christina Dan Wang
Roberto Fray da Silva
Astarag Mohapatra
Marc Hollyoak
Jingyang Rui
Ming Zhu
Dan Yang
Mao Guan
Markus Kefeder
Ziyi Xia
Shixun Wu
Momin Haider
David Wilt
Berend Gort

Educational Usage of This Doc:

1. Research credit course at Columbia University.
2. Capstone projects at NYU.
3. Senior projects at Northwestern University.

Summary

Deep reinforcement learning (DRL) has been recognized as an effective approach in quantitative finance, thus getting hands-on experiences is attractive to beginners. However, training a profitable DRL trading agent that ***decides where to trade, at what price*****, and** ***what quantity*** involves error-prone and arduous code development and debugging. This proposal is based on the first open-source DRL framework, *FinRL,* that facilitates beginners to expose themselves to quantitative finance.

This project proposal consists of three parts:

(1) ***Introduction to FinRL Ecosystem***: We give an introduction to the FinRL library and help students understand the framework.

(2) ***Tutorials and Examples***: We provide tutorials and examples:
FinRL: **Multiple Stock Trading** is based on our paper: *FinRL: A Deep Reinforcement Learning Library for Automated Stock Trading in Quantitative Finance*, Deep RL Workshop, NeurIPS 2020. For more information, please visit the website
Explainable FinRL: **An Empirical Approach** is based on our paper: *Explainable Deep Reinforcement Learning for Portfolio Management: An Empirical Approach*, 2nd ACM International Conference on AI in Finance. For more information, please visit the website
ElegantRL: Our examples contain step-by-step tutorials from problem definition to backtesting. In Google Collab files, each step includes a detailed explanation to help students understand the code. For more information, please visit the website
FinRL-Meta: is a universe of market environments for data-driven financial reinforcement learning. Users can use FinRL-Meta as the metaverse of their financial environments. FinRL-Meta is based on our paper: *FinRL-Meta: A Universe of Near-Real Market Environments for Data-Driven Deep Reinforcement Learning in Quantitative Finance.* Data-Centric AI Workshop, NeurIPS 2021. For more information, please visit the website

(3) ***Senior/capstone Projects***: We would like to propose tens of senior/capstone projects based on our ongoing projects: ***FinRL, ElegantRL*** and ***FinRL-Meta***.
- **FinRL:** use FinRL to pipeline your own trading strategy.
- **ElegantRL:** develop and optimize deep reinforcement learning algorithms.
- **FinRL-Meta:** apply to different markets.

Along with easily-reproducible tutorials, FinRL allows users to streamline their developments and compare with existing schemes. Within FinRL, virtual environments are configured with market data, trading agents are trained with neural networks, and extensive backtesting is analyzed via trading performance. Moreover, it incorporates essential trading constraints such as transaction cost, market liquidity, and investor's degree of risk-aversion.

FinRL is featured with *completeness*, *hands-on tutorials* and *reproducibility* that favor beginners: (i) at multiple levels of time granularity, FinRL simulated trading environments across various markets, including NASDAQ-100, S&P 500, HSI, SSE 50, and CSI 300; (ii) organized in a layered architecture

with modular structure, FinRL provides fine-tuned state-of-the-art DRL algorithms (DQN, DDPG, PPO, SAC, A2C, TD3, etc.), commonly used reward functions and standard evaluation baselines to alleviate the debugging workloads and promote the reproducibility, and (iii) being highly extendable, FinRL reserves a complete set of user-import interfaces.

1. Introduction to FinRL

FinRL is the first **open-source** framework to help practitioners establish the development pipeline of trading strategies using **deep reinforcement learning (DRL)**.

Moreover, as a proof-of-concept, FinRL is designed specifically with an effort for **educational and demonstrative purposes**.

1.1 Target Audience

- **Students** who want to get hands-on experience on a real-life end-to-end project that joins the fields of AI and Finance.
- **Data Scientists** who want to learn best practices of financial big data processing, utilize open-source data engineering tools for financial big data, and catch up with the most recent state-of-the-art DRL algorithms.
- **Software Developers** who want to step into machine learning and reinforcement learning.
- **Quantitative Researchers** who want to design and deploy their deep reinforcement learning strategy into paper trading, and learn cloud-native solutions for high performance and high scalability training.

1.2 Project Overview

Guideline and how to approach this project

This project is mainly separated into three parts: FinRL, FinRL-Meta, and ElegantRL. Each part serves a particular purpose in the FinRL ecosystem. A brief overview of each part is given in terms of goals, designing principles and overall framework.

A. FinRL

- **Goals of FinRL**

  The design of a deep reinforcement learning trading strategy includes:
  1. preprocessing market data,
  2. building a training environment,
  3. managing trading states,
  4. and backtesting trading performance.

It is a very tedious debugging and error-prone programming process. The end-to-end pipeline is also pretty comprehensive.

FinRL's Goal:

1. FinRL has a **full pipeline** to help quantitative traders overcome the **steep learning curve**.
2. FinRL implements fine-tuned state-of-the-art DRL algorithms and common reward functions, while **alleviating the debugging workloads**.
3. FinRL framework **automatically streamlines** the development of trading strategies, so as to help researchers and quantitative traders to **iterate their strategies at a high turnover rate**.

- **Designing Principles**

1. **Full-stack framework**. To provide a full-stack DRL framework with finance-oriented optimizations, including **market data APIs, data preprocessing, DRL algorithms, and automated backtesting**. Users can transparently make use of such a development pipeline.
2. **Customization**. To maintain modularity and extensibility in development by including **state-of-the-art DRL algorithms** and supporting design of new algorithms. The DRL algorithms can be used to construct trading strategies by simple configurations.
3. **Reproducibility and hands-on tutoring**. To provide tutorials such as **step-by-step Jupyter notebooks** and user guides to help users walk through the pipeline and reproduce the use cases.

- **Framework of FinRL**

The FinRL framework has three layers: application layer, agent layer, and environment layer.

1. For the **application layer**, FinRL aims to provide **hundreds of demonstrative trading tasks**, serving as stepping stones for users to develop their strategies.
2. For the **agent layer**, FinRL supports fine-tuned DRL algorithms from **DRL libraries in a plug-and-play** manner, following the unified workflow.
3. For the **environment layer,** FinRL aims to wrap **historical data and live trading APIs** of hundreds of markets into training environments, following the de facto standard Gym.

B. FinRL-Meta

- **Goals of FinRL-Meta**

To support different trading tasks, we need to train multiple agents using various environments. This requires a diverse RL-based market environment. The current work targets developing trading strategies instead of market simulation.

Yet, no prior work focuses on building the **financial market RL environments** as OpenAI Gym did for Atari games RL environments.

FinRL-Meta's Goal:

1. FinRL-Meta separates **financial data processing** from the design pipeline of DRL-based strategy and provides open-source data engineering tools for **financial big data**.
2. FinRL-Meta provides hundreds of **market environment simulations** for various trading tasks.
3. FinRL-Meta enables **multiprocessing simulation** and training by exploiting thousands of GPU cores.

- **Designing Principles**

1. **DataOps for Data-Driven DRL in Finance**. The DataOps paradigm is adopted to the data engineering pipeline, providing agility to agent deployment.
2. **Layered Structure & Extensibility**. A layered structure specialized for RL in finance. This specialized structure realizes the extensibility of FinRL-Meta.
3. **Plug-and-Play**. Any DRL agent can be directly plugged into the environments, then trained and tested. Different agents can run on the same benchmark environment for fair comparison.

- **Framework of FinRL-Meta**

  FinRL-Meta consists of three layers: data layer, environment layer, and agent layer. Each layer executes its functions and is relatively independent.

1. For the **data layer**, we use a **unified data processor** to access data, clean data, and extract features.
2. For the **environment layer**, we incorporate trading constraints and model market frictions to **reduce the simulation -to -reality gap**.
3. For the **agent layer**, three DRL libraries (**ElegantRL, RLlib, Stable--Baselines3**) are directly supported, while others can also be plugged in.

C. ElegantRL

- **Goals of ElegantRL**

  ElegantRL is designed for researchers and practitioners with finance-oriented optimizations.

1. ElegantRL implements **state-of-the-art DRL algorithms** from scratch, including both discrete and continuous ones, and provides user-friendly tutorials in Jupyter Notebooks.
2. The ElegantRL performs DRL algorithms under the **Actor-Critic framework**
3. The ElegantRL library enables researchers and practitioners to pipeline the disruptive "design, development and deployment" of DRL technology.

- **Designing Principles**

1. **Lightweight**: core codes have less than 1,000 lines, less dependable packages, only using PyTorch (train), OpenAI Gym (env), NumPy, Matplotlib (plot),
2. **Efficient**: in many testing cases, we find it more efficient than Ray RLlib. ElegantRL provides a cloud-native solution for RLOps in finance.
3. **Stable**: much more stable than Stable Baselines 3. Stable Baselines 3 can only use a single GPU, but ElegantRL can use 1~8 GPUs for stable training.

- **Framework of ElegantRL**

  ElegantRL implements the following model-free deep reinforcement learning (DRL) algorithms:
  - DDPG, TD3, SAC, PPO, PPO (GAE),REDQ for continuous actions
  - DQN, DoubleDQN, D3QN, SAC for discrete actions
  - QMIX, VDN; MADDPG, MAPPO, MATD3 for multi-agent environment

  For the details of DRL algorithms, please check out the educational webpage OpenAI Spinning Up.

1.3 Requirements

Python:
- Confidence with Python programming, and familiar with Jupyter notebook, and Pycharm
- Familiar with Python scripts and executing them from the command line interface
- Familiar with numerical computing libraries: Numpy, and pandas.

Git and Github:
- Knowledge of basic Git commands
- Clone,fork, branch creation and checkout
- Git status, git add, git commit, git pull and git push

Software:
- Python and Anaconda Installation
- Git installation or Github desktop

Account:
- Github account
- Cloud: AWS account or Google Account
- Paper trading account: alpaca, binance

1.4 Install and Setup

Check this blog: **FinRL Install and Setup Tutorial for Beginners** for detailed instructions.

It includes instructions for:

- Mac OS
- AWS Ubuntu
- Windows 10
- Google Colab

1.5 Project materials

- The codes:
- Datasets: APIs
- Presentations: PPT
- AI & RL Knowledge
- Finance & Trading Knowledge

1.6 Additional materials

2. Overview and Tutorials

We demonstrate three existing works as tutorials and examples: **FinRL: Multiple Stock Trading, ElegantRL, and FinRL-Meta.**

FinRL: Multiple Stock Trading is based on our paper: **FinRL: A Deep Reinforcement Learning Library for Automated Stock Trading in Quantitative Finance, Deep RL Workshop, NeurIPS 2020.**

ElegantRL is based on our **blog and Github**.

FinRL-Meta is based on our paper: **FinRL-Meta: A Universe of Near-Real Market Environments for Data-Driven Deep Reinforcement Learning in Quantitative Finance. Data-Centric AI Workshop, NeurIPS 2021.**

2.1. FinRL Overview

Deep reinforcement learning (DRL), which balances exploration (of uncharted territory) and exploitation (of current knowledge), has been recognized as a promising approach for automated trading. DRL algorithms are powerful in solving dynamic decision-making problems by learning through interactions with an unknown environment, thus providing two significant advantages - *portfolio scalability* and *market model independence*. In quantitative finance, trading is essentially making dynamic decisions, namely ***to decide where to trade, at what price, and what quantity***, over a highly stochastic and complex market. As a result, DRL provides a natural toolkit for automated trading. 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.

However, implementing a DRL-driven trading strategy is nowhere near as easy. The code development and debugging processes are arduous and error-prone. Training environments, managing intermediate trading states, organizing market data, and standardizing outputs for evaluation metrics - these steps are standard in implementation yet time-consuming, especially for beginners. Therefore, we created a beginner-friendly library with fine-tuned standard DRL algorithms.

FinRL 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-throughs will help users explore the functionalities;
- **Reproducibility:** Our library shall guarantee reproducibility to ensure transparency and provide users with confidence in what they have done.

We present a three-layered **FinRL** library that streamlines the development of trading strategies. FinRL provides standard building blocks that allow strategy builders to configure market datasets as virtual environments, train deep neural networks as trading agents, analyze trading performance via extensive backtesting, and incorporate essential market frictions.

On the lowest level is an *environment layer*, which simulates the financial market environment using actual historical data from major indices with various environmental attributes such as closing price, shares, trading volume, technical indicators etc.

The *agent layer* in the middle provides fine-tuned standard DRL algorithms such as DQN, DDPG, Adaptive DDPG, Multi-Agent DDPG, PPO, SAC, A2C, and TD3, etc.), commonly used reward functions and standard evaluation baselines to alleviate the debugging workloads and promote the reproducibility. The agent interacts with the environment through properly defined reward functions on the state and action spaces.

The top is an *application layer* that includes various finance applications, here we demonstrate three use cases, namely *multiple stock trading*, *portfolio allocation,* and *cryptocurrency trading*.

A. Architecture of FinRL Framework

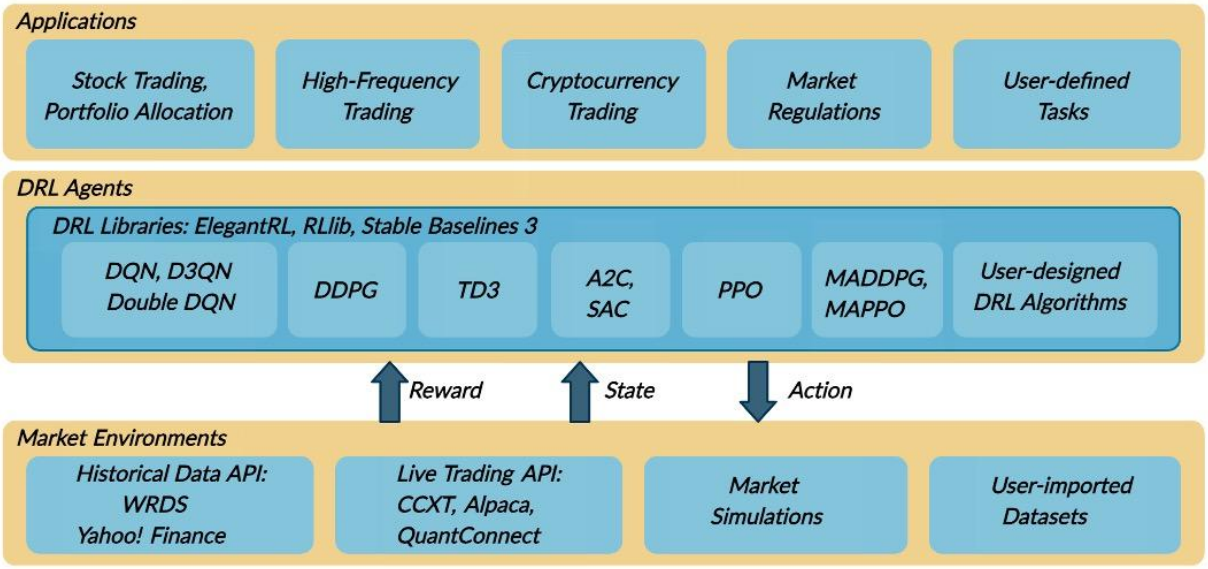


Figure 1: An overview of the FinRL framework. It consists of three layers: application layer, DRL agent layer, and environment layer for various finance markets

An overview of the FinRL library is shown in Fig. 1, and its features are summarized as follows:

- **Three-layer architecture:** The three layers of the FinRL library are market environments, DRL trading agent, and trading applications. 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 rewards. The lower layer provides APIs for the upper layer, making the lower layer transparent to the upper layer.
- **Modularity:** Each layer includes several modules, and each module defines a separate function. One can select specific modules from any layer to implement their trading task. Furthermore, updating existing modules is possible.
- **Simplicity, Applicability and Extendibility:** Specifically designed for automated 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 market simulator that replicates the market and provides backtesting support that incorporates 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.

B. Environment: Market Simulator

Due to the stochastic and interactive nature of the automated trading tasks, a financial task is modeled as a Markov Decision Process (MDP) problem. The training process involves observing price change, taking action, and calculating the reward to have the agent adjust its strategy accordingly. The trading agent will derive a trading strategy with the maximized rewards as time proceeds by interacting with the market environment.

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.

Standard and User-Imported Datasets

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.

C. DRL Agents: ElegantRL

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 self-driving cars, gaming, natural language processing (NLP), 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.

Overall, the **Contributions of FinRL** are summarized as follows:

- 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 and used to establish a benchmark for the quantitative finance community. Standard backtesting and evaluation metrics are also provided for easy and effective performance evaluation.

With FinRL Library, implementation of powerful DRL driven trading strategies is made an accessible, efficient and delightful experience.

2.2. FinRL Tutorials

FinRL: Multiple Stock Trading

To begin with, we would like to explain the logic of stock trading using Deep Reinforcement Learning. We use Dow 30 constituents as an example throughout this tutorial, because they are popular stocks.

A lot of people are terrified by the word "Deep Reinforcement Learning", actually, you can just think of 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 this 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, where h_max is a predefined parameter that is set 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 amounts 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 the 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.

Fig. 2 shows the logic chart for multiple stock trading and an example for demonstration purpose:

Time t	Balance	h_max	
	$ 600,000.00	100	

	Balance - (Buy Amount - Sell Amount)	$	600,000.00
-	Amount to Trade	$	60,831.88
=	Update Balance	$	539,168.12

Time t+1	Balance
	$ 539,168.12

Ticker	Shares Held	Close Price	Dollar Amount	Shares to Trade (DRL Trader)	Dollar amount to Trade	Updated Shares	Close Price at t+1	Dollar Amount at t+1
AAPL	100	$ 38.25	$ 3,824.94	-100	$ (3,824.94)	0	$ 35.83	$ -
AXP	100	$ 92.64	$ 9,264.33	10	$ 926.43	110	$ 95.44	$ 10,498.76
BA	100	$ 314.65	$ 31,464.51	-10	$ (3,146.45)	90	$ 302.10	$ 27,189.05
CAT	100	$ 119.30	$ 11,930.26	10	$ 1,193.03	110	$ 121.05	$ 13,315.39
CSCO	100	$ 40.38	$ 4,038.21	50	$ 2,019.10	150	$ 40.94	$ 6,141.00
CVX	100	$ 100.66	$ 10,066.45	30	$ 3,019.93	130	$ 102.09	$ 13,272.00
DD	100	$ 73.62	$ 7,361.71	-30	$ (2,208.51)	70	$ 73.94	$ 5,175.89
DIS	100	$ 107.65	$ 10,765.43	-100	$ (10,765.43)	0	$ 109.23	$ -
GS	100	$ 164.59	$ 16,459.10	20	$ 3,291.82	120	$ 168.41	$ 20,209.02
HD	100	$ 164.86	$ 16,486.42	50	$ 8,243.21	150	$ 169.29	$ 25,393.73
IBM	100	$ 104.46	$ 10,445.89	60	$ 6,267.53	160	$ 107.12	$ 17,139.92
INTC	100	$ 44.82	$ 4,481.96	30	$ 1,344.59	130	$ 45.16	$ 5,871.10
JNJ	100	$ 120.96	$ 12,095.92	30	$ 3,628.78	130	$ 120.26	$ 15,633.61
JPM	100	$ 92.93	$ 9,292.85	20	$ 1,858.57	120	$ 95.05	$ 11,406.13
KO	100	$ 43.98	$ 4,398.35	30	$ 1,319.50	130	$ 44.00	$ 5,720.29
MCD	100	$ 167.73	$ 16,772.65	100	$ 16,772.65	200	$ 171.69	$ 34,337.91
MMM	100	$ 178.09	$ 17,809.44	50	$ 8,904.72	150	$ 178.03	$ 26,704.38
MRK	100	$ 71.88	$ 7,188.41	30	$ 2,156.52	130	$ 71.73	$ 9,325.16
MSFT	100	$ 98.60	$ 9,860.21	10	$ 986.02	110	$ 99.52	$ 10,947.05
NKE	100	$ 72.74	$ 7,273.81	-50	$ (3,636.91)	50	$ 71.45	$ 3,572.58
PFE	100	$ 38.01	$ 3,801.23	30	$ 1,140.37	130	$ 36.95	$ 4,803.35
PG	100	$ 86.64	$ 8,664.32	-30	$ (2,599.29)	70	$ 87.44	$ 6,120.83
RTX	100	$ 64.80	$ 6,479.66	50	$ 3,239.83	150	$ 64.00	$ 9,600.13
TRV	100	$ 112.36	$ 11,236.43	100	$ 11,236.43	200	$ 113.25	$ 22,650.92
UNH	100	$ 236.41	$ 23,641.15	-30	$ (7,092.34)	70	$ 233.10	$ 16,317.04
V	100	$ 131.26	$ 13,126.45	100	$ 13,126.45	200	$ 134.37	$ 26,873.09
VZ	100	$ 51.49	$ 5,149.29	50	$ 2,574.65	150	$ 51.68	$ 7,751.52
WBA	100	$ 63.17	$ 6,317.01	-30	$ (1,895.10)	70	$ 64.96	$ 4,547.34
WMT	100	$ 90.31	$ 9,031.39	10	$ 903.14	110	$ 91.48	$ 10,062.25
XOM	100	$ 61.59	$ 6,158.63	30	$ 1,847.59	130	$ 63.20	$ 8,216.46
Total			$ 314,886.42	Amount to Trade	$ 60,831.88		Stock Dollar Amount	$ 378,795.88
		+	$ 600,000.00			+	Balance (t+1)	$ 539,168.12
Portfolio Value at t		=	$ 914,886.42			=	V(t+1)	$ 917,964.00
						-	V(t)	$ 914,886.42
						=	Reward	$ 3,077.58
						We have a positive reward by taking the actions from DRL Trader!		

Figure 2: An example of the logic chart for multiple stock trading

Multiple stock trading: as the number of stocks increase, the dimension of the data will increase, the state and action space will grow exponentially. So stability and reproducibility are very essential.

This tutorial is focusing on one of the use cases in our paper: **Multiple Stock Trading**. We use one Jupyter notebook to include all the necessary steps. For more details and code about this example, **please visit here.**

Part 1. 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.

Part 2. Getting Started- 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-Foundation/FinRL-Library.git
```

Import Packages:

packages.py Raw

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

from finrl.config import config
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
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)
```

Part 3. Download Data

FinRL uses several data processors, e.g., **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:

DownloadData.py Raw

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

	date	open	high	low	close	volume	tic
0	2009-01-02	3.067143	3.251429	3.041429	2.773207	746015200.0	AAPL
1	2009-01-02	18.570000	19.520000	18.400000	15.800624	10955700.0	AXP
2	2009-01-02	42.799999	45.560001	42.779999	33.680935	7010200.0	BA
3	2009-01-02	44.910000	46.980000	44.709999	32.514400	7117200.0	CAT
4	2009-01-02	16.410000	17.000000	16.250000	12.786087	40980600.0	CSCO
5	2009-01-02	74.230003	77.300003	73.580002	48.043262	13695900.0	CVX
6	2009-01-02	21.605234	22.060680	20.993229	14.527276	13251000.0	DD
7	2009-01-02	22.760000	24.030001	22.500000	20.597496	9796600.0	DIS
8	2009-01-02	84.019997	87.620003	82.190002	72.844467	14088500.0	GS
9	2009-01-02	23.070000	24.190001	22.959999	18.007103	14902500.0	HD
10	2009-01-02	83.889999	87.589996	83.889999	59.605556	7558200.0	IBM
11	2009-01-02	14.690000	15.250000	14.470000	10.527306	52208200.0	INTC
12	2009-01-02	60.130001	61.000000	59.040001	42.265537	11638900.0	JNJ
13	2009-01-02	31.190001	31.639999	30.469999	23.515011	32494900.0	JPM
14	2009-01-02	22.700001	23.000000	22.520000	14.135621	16355800.0	KO
15	2009-01-02	62.380001	64.129997	62.200001	44.201206	8652700.0	MCD
16	2009-01-02	57.549999	59.389999	57.520000	42.614807	5313900.0	MMM
17	2009-01-02	30.459999	31.120001	30.309999	20.276968	12207400.0	MRK
18	2009-01-02	19.530001	20.400000	19.370001	15.471161	50084000.0	MSFT
19	2009-01-02	12.737500	13.400000	12.577500	8.974419	12028800.0	NKE
20	2009-01-02	16.963947	17.362429	16.793169	10.938112	30274000.0	PFE
21	2009-01-02	61.689999	62.970001	61.060001	43.361279	11135700.0	PG
22	2009-01-02	33.643803	34.764004	33.373192	25.853655	7452800.0	RTX
23	2009-01-02	45.259998	45.910000	44.130001	33.668175	3279700.0	TRV
24	2009-01-02	26.700001	27.719999	26.540001	23.298557	4885900.0	UNH
25	2009-01-02	13.230000	13.427500	13.060000	12.244266	13199200.0	V
26	2009-01-02	32.000774	32.601021	31.466175	18.187479	14848900.0	VZ
27	2009-01-02	24.650000	25.670000	24.610001	19.187819	5266700.0	WBA
28	2009-01-02	55.980000	57.509998	55.779999	42.877636	16054800.0	WMT
29	2009-01-02	80.059998	82.110001	78.900002	53.483833	35803700.0	XOM

Part 4. 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:

FeatureEngineer.py

```
# 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()
```

	date	open	high	low	close	volume	tic	macd	rsi_30	cci_30	dx_30	turbulence
0	2009-01-02	3.067143	3.251429	3.041429	2.773207	746015200.0	AAPL	0.0	100.0	66.666667	100.0	0.0
1	2009-01-02	18.570000	19.520000	18.400000	15.800624	10955700.0	AXP	0.0	100.0	66.666667	100.0	0.0
2	2009-01-02	42.799999	45.560001	42.779999	33.680935	7010200.0	BA	0.0	100.0	66.666667	100.0	0.0
3	2009-01-02	44.910000	46.980000	44.709999	32.514400	7117200.0	CAT	0.0	100.0	66.666667	100.0	0.0
4	2009-01-02	16.410000	17.000000	16.250000	12.786087	40980600.0	CSCO	0.0	100.0	66.666667	100.0	0.0
5	2009-01-02	74.230003	77.300003	73.580002	48.043262	13695900.0	CVX	0.0	100.0	66.666667	100.0	0.0
6	2009-01-02	21.605234	22.060680	20.993229	14.527276	13251000.0	DD	0.0	100.0	66.666667	100.0	0.0
7	2009-01-02	22.760000	24.030001	22.500000	20.597496	9796600.0	DIS	0.0	100.0	66.666667	100.0	0.0
8	2009-01-02	84.019997	87.620003	82.190002	72.844467	14088500.0	GS	0.0	100.0	66.666667	100.0	0.0
9	2009-01-02	23.070000	24.190001	22.959999	18.007103	14902500.0	HD	0.0	100.0	66.666667	100.0	0.0

Users can add their technical indicators: 'macd', 'boll_ub', 'boll_lb', 'rsi_30', 'dx_30', 'close_30_sma', 'close_60_sma'

Train & Trade Data Split: In real life trading, the model needs to be updated periodically using rolling windows. In this article, we just split the data into train and trade sets.

Part 5. Build Environment

FinRL uses a **EnvSetup** class to set up an 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

"""
```

State Space and Action Space Calculation:

The action space is just the number of unique stocks **30**. The state space is **181** in this example.

calculate _space.py Raw

```
stock_dimension = len(train.tic.unique())
state_space = stock_dimension
```

Initialize an environment class:

InitializeEnv.py Raw

```
# Initialize env:
env_setup = EnvSetup(stock_dim = stock_dimension,
                     state_space = state_space,
                     initial_amount = 1000000,
                     tech_indicator_list = config.TECHNICAL_INDICATORS_LIST)

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

User-defined Environment: a simulation environment class.The environment for training and trading is different from that in the multiple stock trading case.

Training v.s. Trading: turbulence index is used as a risk aversion signal after the actions generated by the DRL algorithms. Turbulence index should not be included in training, because it is not a part of model training, so only a trading environment should include the risk aversion signal.

FinRL provides a blueprint for **training and trading** environments in multiple stock trading.

Part 6. Implement 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.

```
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
    train_SAC()
        the implementation for SAC algorithm
    DRL_prediction()
        make a prediction in a test dataset and get results
    """
```

Model Training:

ModelTraining.py

```
env_train = env_setup.create_env_training(data = train,
                                          env_class = StockPortfolioEnv)
agent = DRLAgent(env = env_train)

print("==============Model Training===========")
now = datetime.datetime.now().strftime('%Y%m%d-%Hh%M')
a2c_params_tuning = {'n_steps':5,
                     'ent_coef':0.005,
                     'learning_rate':0.0003,
                     'verbose':0,
                     'timesteps':50000}
model_a2c = agent.train_A2C(model_name = "A2C_{}".format(now), model_params = a2c_params_tuning)
```

We use Soft Actor-Critic (SAC) for multiple stock trading, because it is one of the most recent state-of-art algorithms. SAC is characterized by its stability.

Trading:

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

tradingenv.py

```
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)
```

```
previous_total_asset:1000000
end_total_asset:1387593.0471878196
total_reward:387593.0471878196
total_cost:  7276.527191085817
total trades:  6582
Sharpe:  1.0670255446734498
```

Part 7. Backtesting Performance

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

BackTest.py

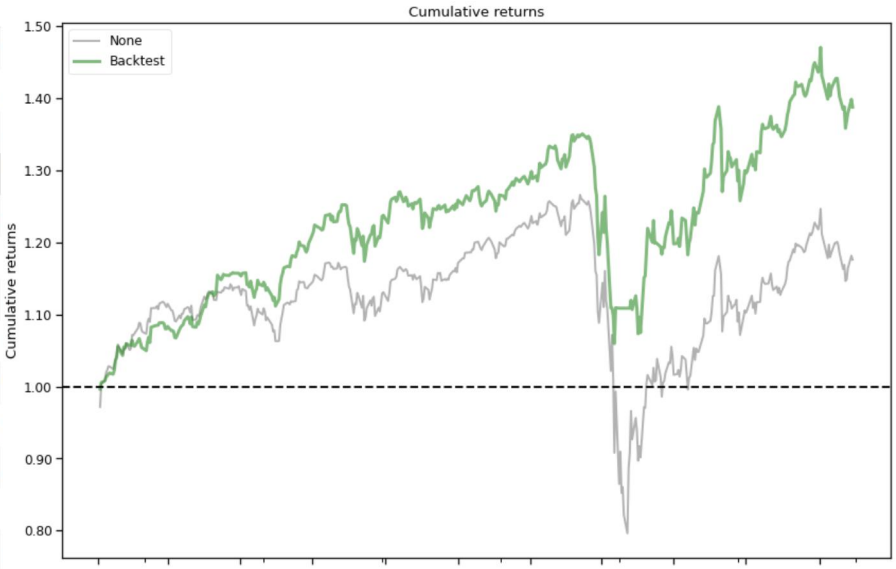
```
from pyfolio import timeseries
DRL_strat = backtest_strat(df_daily_return)
perf_func = timeseries.perf_stats
perf_stats_all = perf_func( returns=DRL_strat,
                            factor_returns=DRL_strat,
                              positions=None, transactions=None, turnover_denom="AGB")
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)
```

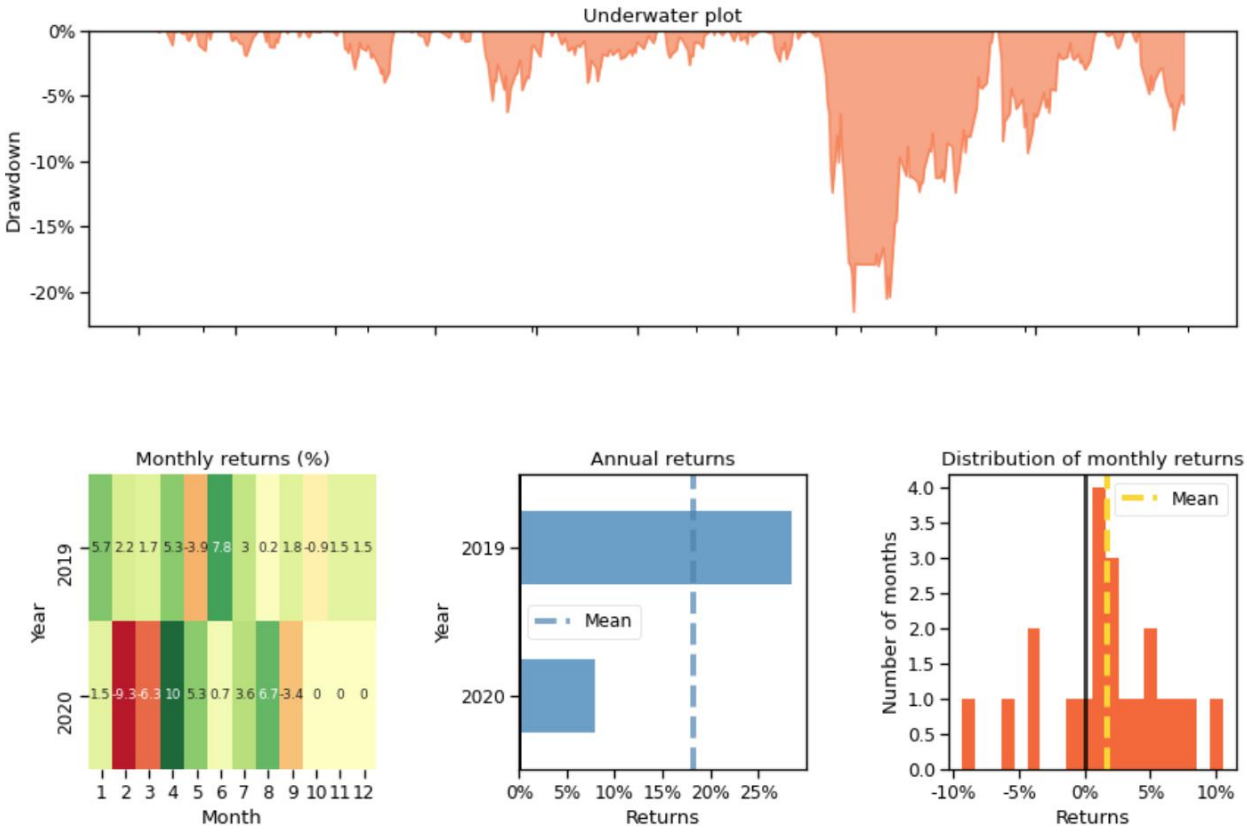
The left table is the stats for **backtesting performance**, the right table is the stats for **Index (DJIA) performanc**e.

Annual return	0.206361	Annual return	0.097239
Cumulative returns	0.387593	Cumulative returns	0.175892
Annual volatility	0.193942	Annual volatility	0.287173
Sharpe ratio	1.067026	Sharpe ratio	0.468543
Calmar ratio	0.957894	Calmar ratio	0.262198
Stability	0.637709	Stability	0.010261
Max drawdown	-0.215432	Max drawdown	-0.370862
Omega ratio	1.231145	Omega ratio	1.110729
Sortino ratio	1.504066	Sortino ratio	0.641939
Skew	NaN	Skew	NaN
Kurtosis	NaN	Kurtosis	NaN
Tail ratio	0.944479	Tail ratio	0.807113
Daily value at risk	-0.023613	Daily value at risk	-0.035647

Plots:

Start date	2019-01-03
End date	2020-09-29
Total months	20
	Backtest
Annual return	20.688%
Cumulative returns	38.759%
Annual volatility	19.394%
Sharpe ratio	1.07
Calmar ratio	0.96
Stability	0.64
Max drawdown	-21.543%
Omega ratio	1.23
Sortino ratio	1.50
Skew	-0.46
Kurtosis	8.46
Tail ratio	0.94
Daily value at risk	-2.361%





Conclusion

For more details about **Multiple Stock Trading,** please visit **Blog** and **Code**.

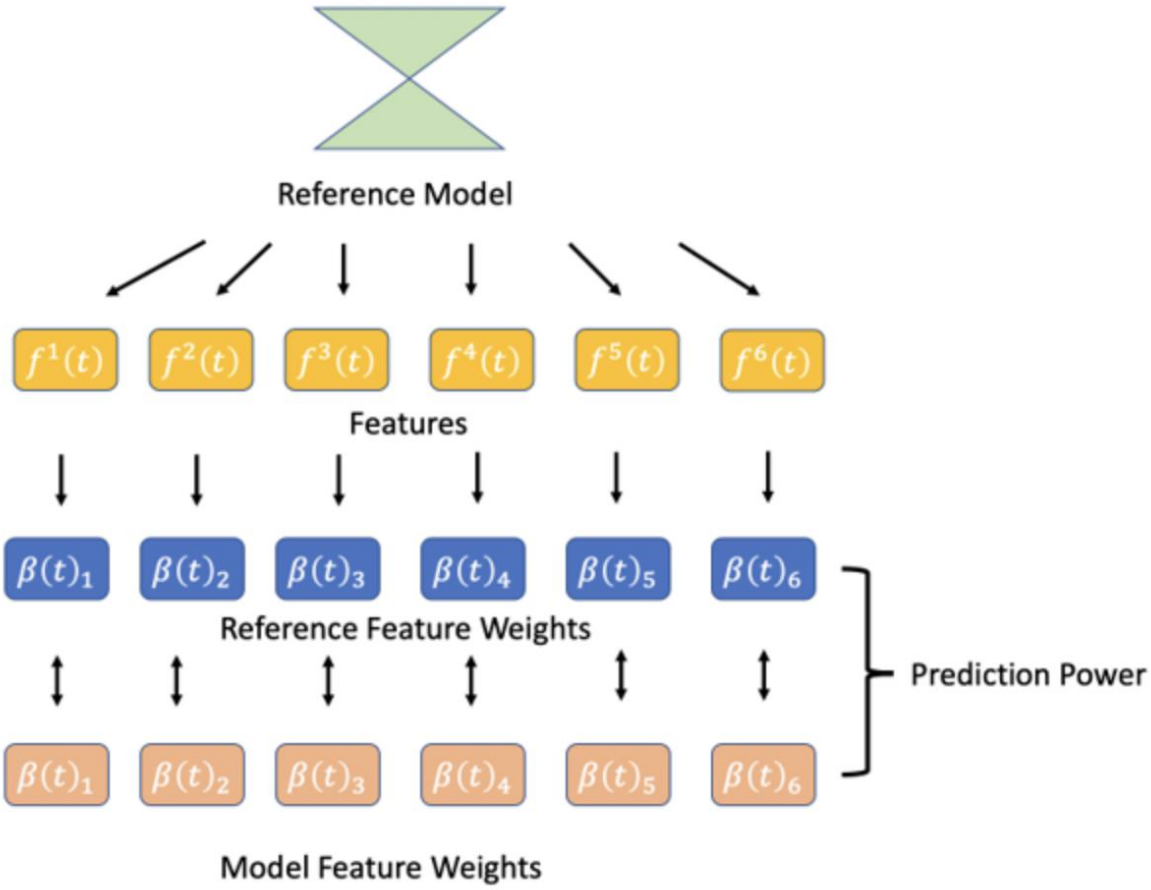
We also developed examples and tutorials for **Portfolio Allocation.** For more details about **Portfolio Allocation,** please visit **Blog** and **Code**.

Explainable FinRL: An Empirical Approach

First of all, our work aims to provide an empirical approach to explain the portfolio management task on the basis of FinRL settings.

We propose an empirical approach to explain the strategies of DRL agents for the portfolio management task:

- First, we study the portfolio management strategy using **feature weights**, which quantify the relationship between the reward (say, portfolio return) and the input (say, features). In particular, we use the coefficients of **a linear model in hindsight** as the **reference feature weights**.
- Then, for the deep reinforcement learning strategy, we use **integrated gradients** to define the **feature weights**, which are the coefficients between reward and features under a linear regression model
- Finally, we quantify the **prediction power** by calculating the **linear correlations** between the coefficients of a DRL agent and the reference feature weights, and similarly for conventional machine learning methods. Moreover, we consider both the single-step case and multiple-step case.



Part 1. Portfolio Management Task

Consider a portfolio with ý risky assets over d time slots, the portfolio management task aims to maximize profit and minimize risk.

- The price relative vector $y(t) \in R^N$ is defined as the element-wise division of $p(t)$ by $p(t-1)$: $y(t) \triangleq [\frac{p_1(t)}{p_1(t-1)}, \frac{p_2(t)}{p_2(t-1)}, \dots, \frac{p_N(t)}{p_N(t-1)}]^T, t = 1, ...d$, where $p(0) \in R^N$ is the vector of opening prices at $t = 1$ and $p(t) \in R^N$ denotes the closing prices of all assets at time slot $t = 1, ...,d$.
- Let $w(0) \in R^N$ denotes the portfolio weights, which is updated at the beginning of time slot t
- The rate of portfolio return is $w(t)^T y(t) - 1$, and the logarithmic rate of portfolio return is $\ln(w(t)^T y(t))$.
- The risk of a portfolio is defined as the variance of the rate of portfolio return: $w(t)^T \Sigma(t) \ w(t)$, where $\Sigma(t) = \text{Cov}(y(t)) \in R^{N \times N}$ is the covariance matrix of the stock returns at the end of time slot t.
- Our goal to find a portfolio weight vector $w^*(t) \in R^N$ such that

  $w^*(t) \triangleq \text{argmax}_{w(t)} \ w(t)^T y(t) - c \ w(t)^T \Sigma(t) \ w(t)$, s.t. $\sum_{i=1}^N \boldsymbol{w_i}(t) = 1, \boldsymbol{w_i}(t) \in [0, 1], t = 1, ..., T$, where c is the risk aversion parameter

Part 2. The DRL Agent Settings For Portfolio Management Task

Similar to the tutorial FinRL: Multiple Stock Trading, we model the portfolio management 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 an agent can take at a state. In our task, the action $w(t) \in R^N$ corresponds to the portfolio weight vector decided at the beginning of time slot t and should satisfy the constraints: firstly, the each element is between 0 and 1, secondly the summation of all elements is 1.
- Reward function: The reward function $r(s(t), w(t), s(t + 1))$ is the incentive for an agent to learn a profitable policy. We use the logarithmic rate of portfolio return: $\ln(w(t)^T y(t))$ as the reward, where $y(t) \in R^N$ is the price relative vector.
- State: describes an agent's perception of a market. The state at the beginning of time slot t is $s(t) = [f^1(t), ..., f^K(t), \Sigma(t)] \in R^{N \times (N+K)}, t = 1, ...,d$.
- DRL Algorithms: We use two popular deep reinforcement learning algorithms: Advantage Actor Critic (A2C) and Proximal Policy Optimization (PPO).
- Environment: Dow Jones 30 constituent stocks during 01/01/2009 to 09/01/2021

Part 3. The Data Preparation

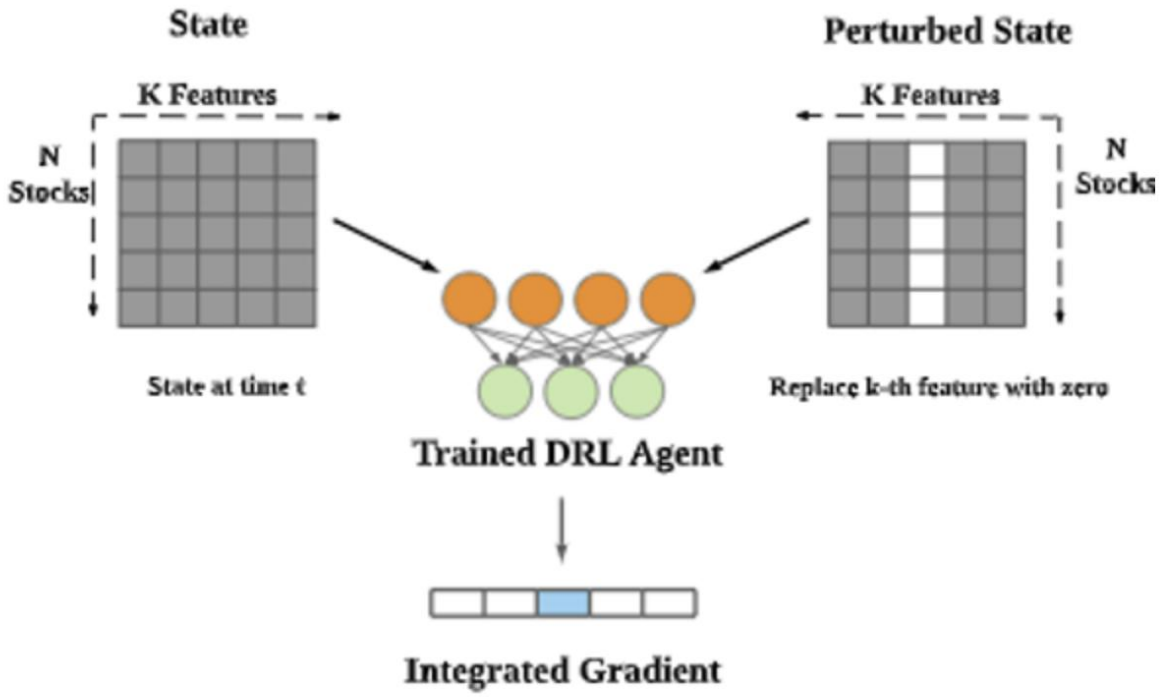
The data preparation step includes importing python packages, downloading data and feature engineering. Our work is based on the tutorial FinRL: Multiple Stock Trading, we use the same settings with it.

Part 4. The Reference Model & Feature Weights

We use a linear model in hindsight as a reference model. For a linear model in hindsight, a demon would optimize the portfolio with actual stock returns and the actual sample covariance matrix. It is the upper bound performance that any linear predictive model would have been able to achieve.

We use the regression coefficients to define the **reference feature weights** as $\beta(t) := [\beta(t)_1, \beta(t)_2, \ldots, \beta(t)_K] \in R^K$, where $\beta(t)_k = \Sigma^N_{i=1} \beta_k(t) \cdot f^k(t)_i$, $\beta_k(t)$ is the coefficient in the linear model:

$$w^*(t) \odot y(t) = \beta_0(t) \cdot [1, \ldots, 1]^T + \beta_1(t) \cdot f^1(t) + \ldots + \beta_K(t) \cdot f^K(t) + \epsilon(t)$$



Feature weights of a trained DRL agent.

Part 5. The Feature Weights For DRL Agents

We use integrated gradients to define the feature weights for DRL agents in portfolio management tasks.

$$IG(x)_i := (x_i - x'_i) \times \int_{z=0}^{1} \frac{\partial F(x' + z\,(x - x'))}{\partial x_i} dz,$$

where $x \in R^N$ is the input and $F(\cdot)$ is the DRL model. Likewise, we use linear regression coefficients to help understand DRL agents:

$$w^{DRL}(t) \odot y(t) = c_0(t) \cdot [1, \ldots, 1]^T + c_1(t) \cdot f^1(t) + \ldots + c_K(t) \cdot f^K(t) + \epsilon(t).$$

Lastly, we define the feature weights of DRL agents in portfolio management task using integrated gradients and the regression coefficients.

$$M^{\pi}(t) := [\, M^{\pi}(t)_1, \ldots, M^{\pi}(t)_K \,],$$

where $M^{\pi}(t)_k := \Sigma^N_{i=1} IG(f^k(t))_i \approx \Sigma^N_{i=1} f^k(t)_i \cdot \Sigma^{\infty}_{i=1} \gamma^l \frac{\partial E[w^{DRL}(t+l)^T y(t+l) \mid s^{k,i}(t),\, w(t)]}{\partial f^k(t)_i}$

$$= \Sigma^N_{i=1} f^k(t)_i \cdot \Sigma^{\infty}_{i=1} \gamma^l E[c_k(t+l) \frac{\partial f^k(t+l)_i}{\partial f^k(t)_i} \mid s^{k,i}(t),\, w(t)]$$

Part 6. The Feature Weights For Machine Learning Methods

We use conventional machine learning methods as comparison.

- Firstly, it uses the features as input to predict the stock returns vector.
- Secondly, it builds a linear regression model to find the relationship between the portfolio return vector q and features.
- Lastly, it uses the regression coefficients b to define the feature weights as follows.

We define the feature weights for machine learning methods as

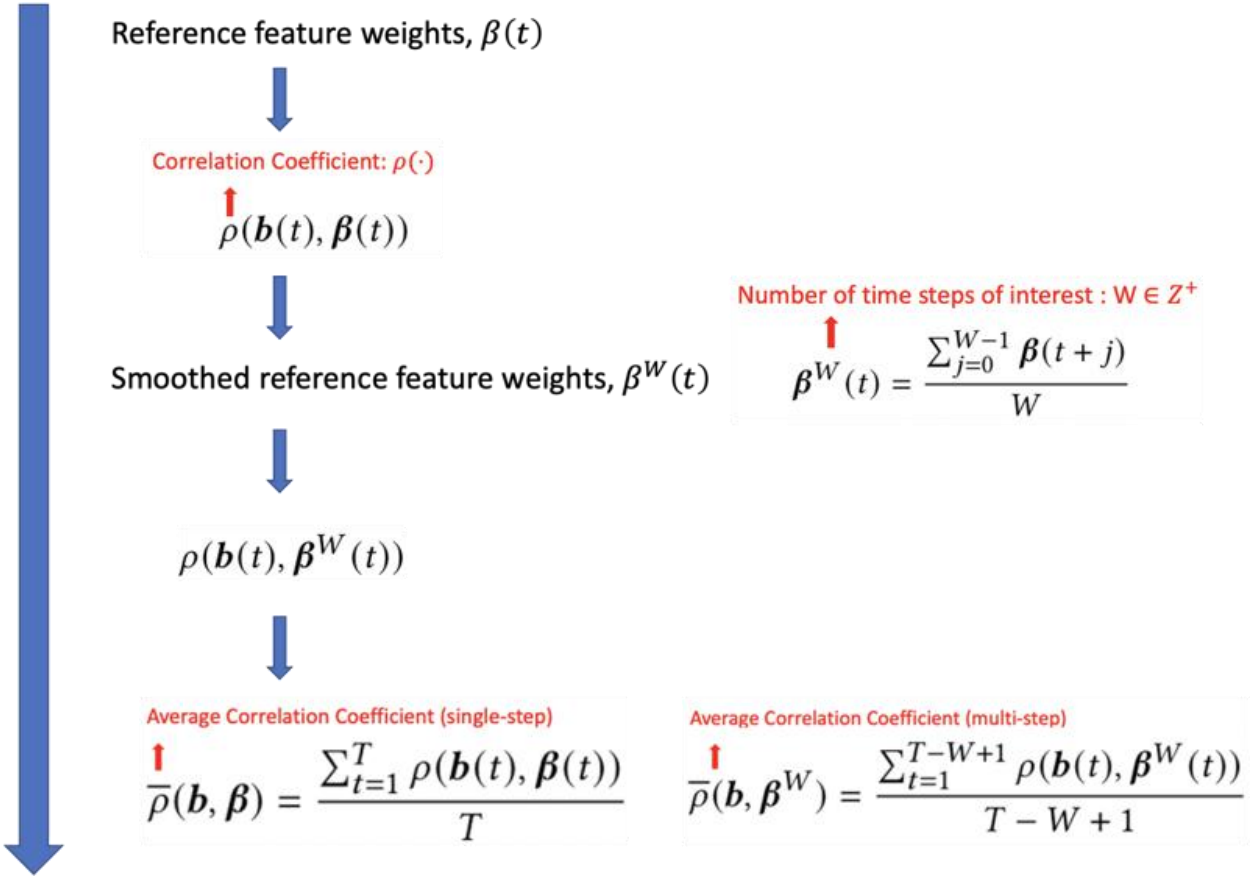
b(t) := $[b(t)_1, b(t)_2, \ldots, b(t)_K] \in R^K$, where $b(t)_k = \Sigma^N_{i=1} b_k(t) \cdot f^k(t)_i$, $b_k(t)$ is the coefficient in the linear model:

$$w^{ML}(t) \odot y(t) = b_0(t) \cdot [1, \ldots, 1]^T + b_1(t) \cdot f^1(t) + \ldots + b_K(t) \cdot f^K(t) + \epsilon(t)$$

Part 7. The Prediction Power

Both the machine learning methods and DRL agents take profits from their prediction power. We quantify the prediction power by calculating the **linear correlations** between the feature weights of a DRL agent

and the reference feature weights and similarly for machine learning methods. Furthermore, the machine learning methods and DRL agents are different when predicting the future. The machine learning methods rely on single-step prediction to find portfolio weights. However, the DRL agents find portfolio weights with a long-term goal. Then, we compare two cases, **single-step prediction** and **multi-step prediction**.



Part 8. Experiment & Conclusions

Our experiment environment is as follows:

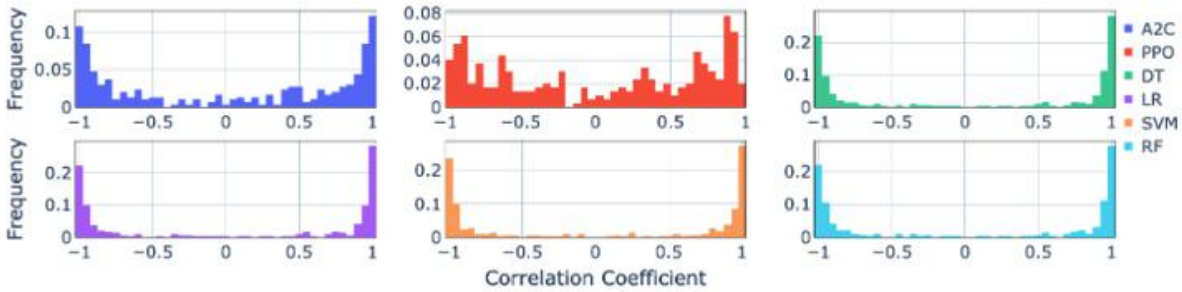
- Algorithms: PPO, A2C, SVM, Decision Tree, Random Forest, Linear Regression
- Data: Dow Jones 30 constituent stocks, accessed at 7/1/2020. We used the data from 1/1/2009 to 6/30/2020 as a training set and the data from 7/1/2020 to 9/1/2021 as a trading set.
- We used four technical indicators as features: MACD, CCI, RSI, ADX
- Benchmark: Dow Jones Industrial Average (DJIA)

The experiment result shows below:

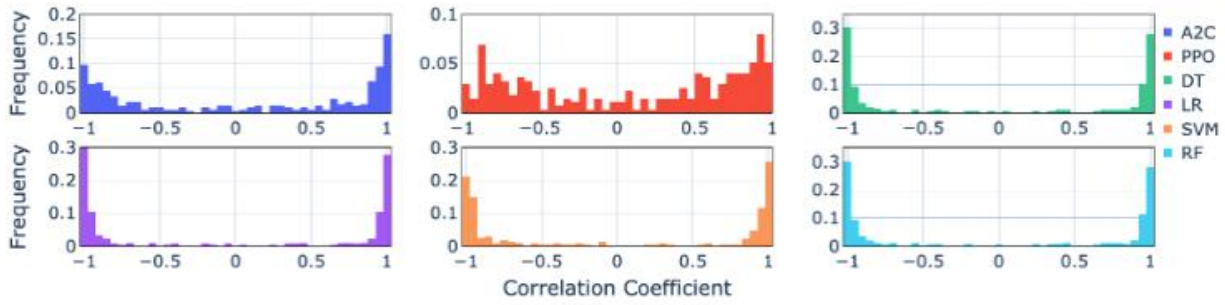
We firstly compare the portfolio performance among the algorithms

(2020/07/01-2021/09/01)	PPO	A2C	DT	LR	RF	SVM	DJIA
Annual Return	**35.0%**	34 %	10.8%	17.6%	6.5%	26.2%	31.2%
Annual Volatility	**14.7%**	14.9 %	40.1%	42.4%	41.2 %	16.2 %	14.1 %
Sharpe Ratio	**2.11**	2.04	0.45	0.592	0.36	1.53	2.0
Calmar Ratio	**4.23**	4.30	0.46	0.76	0.21	2.33	3.5
Max Drawdown	-8.3%	**-7.9%**	-23.5%	-23.2%	-30.7 %	-11.3 %	-8.9 %
Ave. Corr. Coeff. (single-step)	0.024	0.030	**0.068**	0.055	0.052	0.034	N/A
Ave. Corr. Coeff. (multi-step)	**0.09**	0.078	-0.03	-0.03	-0.015	-0.006	N/A

We find that the DRL methods performed best among all and we seek to explain this empirically using our proposed method. So we measured the prediction power and showed their histogram as below.



Single-Step Prediction

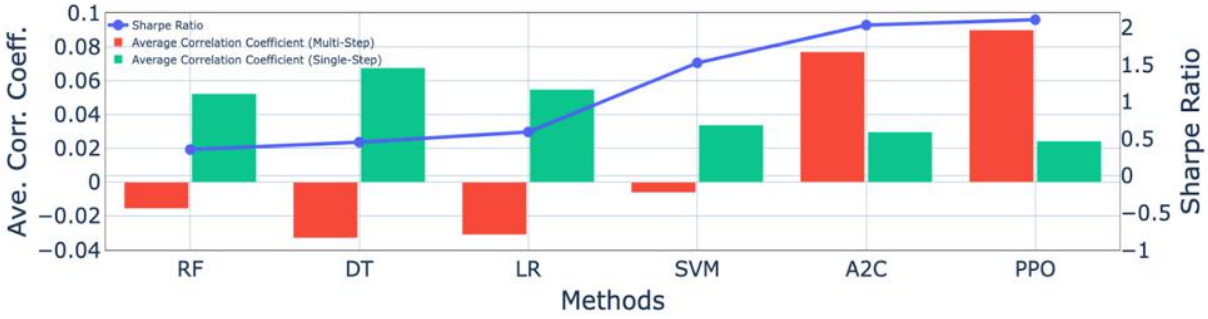


Multi-Step Prediction

To understand the pattern in the histogram, we use the statistical test to measure it.

	Z-statistics (single-step)	Z-statistics (multi-step)
PPO	0.6	**2.16*****
A2C	0.51	**1.58****
DT	**1.28****	-0.59
LR	1.03	-0.55
RF	0.98	-0.28
SVM	0.64	-0.11

We found that the DRL agents showed the most significant prediction multi-step power. We proposed to use it to empirically explain the difference in portfolio performance by comparing it with Sharpe ratio.



We find that:

- The DRL agent using PPO has the highest Sharpe ratio:2.11 and highest average correlation coefficient (multi-step): 0.09 among all the others.
- The DRL agents' average correlation coefficients (multi-step) are significantly higher than their average correlation coefficients (single-step).
- The machine learning methods' average correlation coefficients (single-step) are significantly higher than their average correlation coefficients (multi-step).

- The DRL agents outperform the machine learning methods in multi-step prediction power and fall behind in single-step prediction power.

Overall, a higher mean correlation coefficient (multi-step) empirically indicates a higher Sharpe ratio. For more details**,** please visit **Blog** and **Code**.

2.3. FinRL-Meta Overview

FinRL-Meta is a universe of market environments for data-driven financial reinforcement learning. Users can use FinRL-Meta as the metaverse of their financial environments.

1. FinRL-Meta separates financial data processing from the design pipeline of DRL-based strategy and provides open-source data engineering tools for financial big data.
2. FinRL-Meta provides hundreds of market environments for various trading tasks.
3. FinRL-Meta enables multiprocessing simulation and training by exploiting thousands of GPU cores.

Also called **Neo_FinRL**: **N**ear real-market **E**nvironments f**o**r data-driven **F**inancial **R**einforcement **L**earning.

This part is based on our paper: **FinRL-Meta: Data-Driven Deep Reinforcement Learning in Quantitative Finance**, presented at NeurIPS Workshop on Data-Centric AI. Our codes are available on **Github** and our paper is available on **arXiv.**

Part 1. Overview

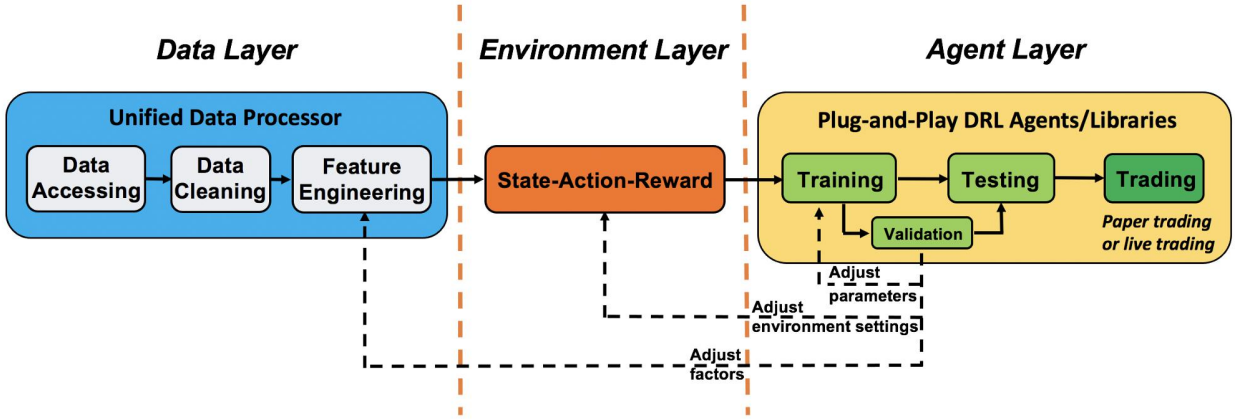
In quantitative finance, market simulators play important roles in studying the complex market phe-nomena and investigating financial regulations. Compared to traditional simulation models, deep reinforcement learning (DRL) has shown huge potential in building financial market simula-tors through multi--agent systems. However, due to the highly complex and dynamic nature of real-world markets, **raw historical financial data** often involve **large noise** and may not **reflect the future of markets**, degrading the fidelity of DRL-based market simulators.

To support different trading tasks, we need to train multiple agents using various environments. This requires a **diverse RL-based market environment**. The current work targets at **developing trading strategies** instead of **market simulation.** Yet, no prior work focuses on building the financial market RL environments like OpenAI Gym did for Atari games RL environments.

We present a **FinRL-Meta framework** that builds a universe of market environments for data-driven financial reinforcement learning.

1. FinRL-Meta separates **financial data processing** from the design pipeline of DRL-based strategy and provides **open-source data engineering tools** for financial big data.
2. FinRL-Meta provides **hundreds of market environments for various trading tasks.**
3. FinRL-Meta enables **multiprocessing simulation** and training by exploiting thousands of **GPU cores**.

Part 2. Proposed FinRL--Meta Framework



Overview of FinRL-Meta

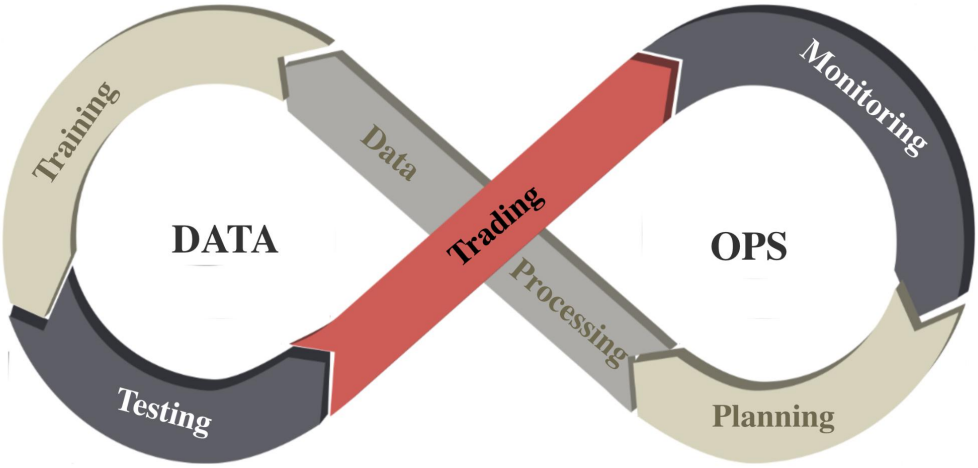
We utilize a layered structure for DRL in finance. FinRL-Meta consists of three layers: **data layer, environment layer, and agent layer**. Each layer executes its functions and is relatively **independent**. Meanwhile, layers interact through end-to-end interfaces to implement the complete workflow of algorithm trading.

This specialized structure realizes the **extensibility** of FinRL-Meta. 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**.

Part 3. DataOps for Data-Driven DRL in Finance

DataOps is a series of principles and practices to improve the quality and reduce the cycle time of data science. It inherits the ideas of Agile development, DevOps, and lean manufacturing and applies them to the data science and machine learning field. Many implementations of DataOps have been developed in companies and organizations to improve the **quality and efficiency** of data science, analytics and machine learning tasks.

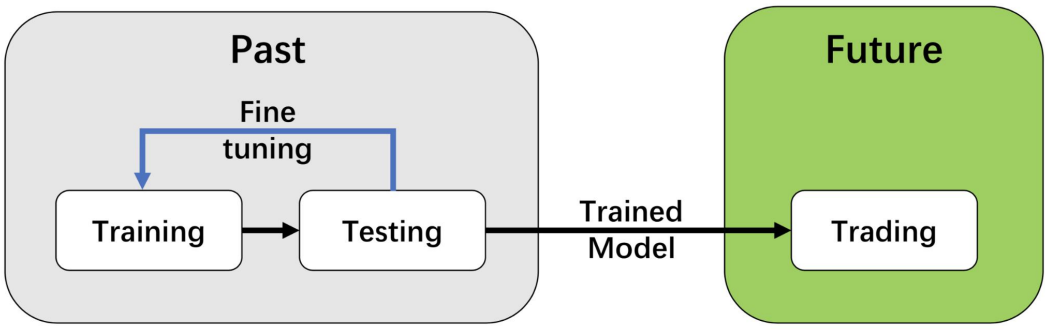
However, the methodology of **DataOps has not been applied to DRL research in quantitative finance**. Most researchers in financial DRL access data, clean data, and extract factors in a **case-by-case manner**, which involves **heavy manual work and may not guarantee the data quality.**



DataOps in FinRL-Meta

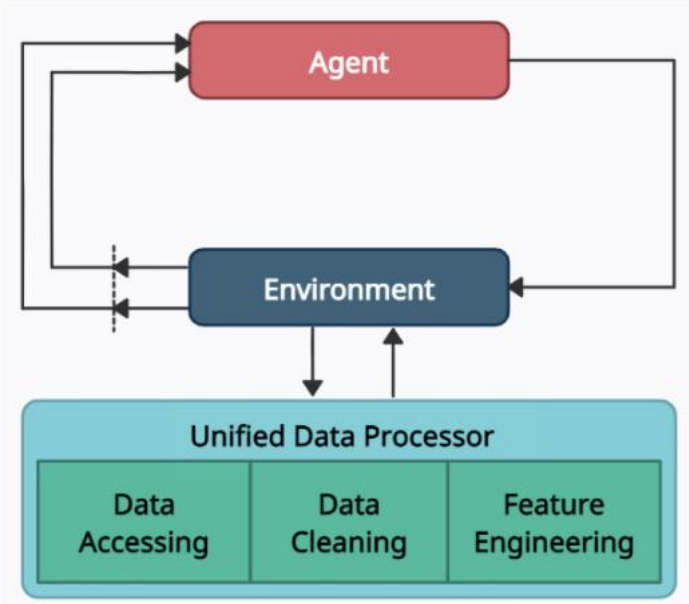
We follow the [DataOps paradigm] in the data layer.

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.



The training-testing-trading pipeline.

Part 4. Data-Centric AI and Unified Data Processor



Overview of automated trading using deep reinforcement learning with DataOps paradigm

In the data layer, we use a unified data processor to access data, clean data, and extract features. The supported platforms and their attributes are shown in the following Table (actively updating):

Platforms	Type	Supported Range and Frequency	Request Limits	Raw Data	Preprocessed Data	Need Account
Yahoo! Finance	US Securities	Depends on frequency, 1min	2,000/hour	OHLCV, df	Prices & Indicators, np.array	No
CCXT	Cryptocurrency	Depends on specific API, 1min	API-specific	OHLCV, df	Prices & Indicators, np.array	No
WRDS.TAQ	US Securities	2003-now, 1ms	5 requests at the same time	Intraday Trades, df	Prices & Indicators, np.array	Yes
Alpaca	US Stocks, ETFs	2015-now, 1min	Account-specific	OHLCV, df	Prices & Indicators, np.array	Yes
JoinQuant	CN Securities	2005-now, 1min	3 requests at the same time	OHLCV, df	Prices & Indicators, np.array	Yes
RiceQuant	CN Securities	2005-now, 1ms	Account-specific	OHLCV, df	Prices & Indicators, np.array	Yes
QuantConnect	US Securities	1998-now, 1s	NA	OHLCV, df	Prices & Indicators, np.array	Yes

Step 1: Data Accessing

We connect data APIs of different platforms and unify them in the **FinRL--Meta** data processor. Users can access data from various sources given the start date, end date, stock list, time frequency, and so on.

```
class DataProcessor():
    def __init__(self, data_source, **kwargs):
        if data_source == 'alpaca':

            try:
                API_KEY= kwargs.get('API_KEY')
                API_SECRET= kwargs.get('API_SECRET')
                APCA_API_BASE_URL= kwargs.get('APCA_API_BASE_URL')
                self.processor = Alpaca(API_KEY, API_SECRET, APCA_API_BASE_URL)
                print('Alpaca successfully connected')
            except:
                raise ValueError('Please input correct account info for alpaca!')

        elif data_source == 'wrds':
            self.processor = Wrds()

        elif data_source == 'yahoofinance':
            self.processor = YahooFinance()

        elif data_source =='binance':
            self.processor = Binance()

        else:
            raise ValueError('Data source input is NOT supported yet.')

    def download_data(self, ticker_list, start_date, end_date,
                      time_interval) -> pd.DataFrame:
        df = self.processor.download_data(ticker_list = ticker_list,
                                          start_date = start_date,
                                          end_date = end_date,
                                          time_interval = time_interval)
        return df

    def clean_data(self, df) -> pd.DataFrame:
        df = self.processor.clean_data(df)

        return df

```

```
    def add_technical_indicator(self, df, tech_indicator_list) -> pd.DataFrame:
        self.tech_indicator_list = tech_indicator_list
        df = self.processor.add_technical_indicator(df, tech_indicator_list)

        return df

    def add_turbulence(self, df) -> pd.DataFrame:
        df = self.processor.add_turbulence(df)

        return df

    def add_vix(self, df) -> pd.DataFrame:
        df = self.processor.add_vix(df)

        return df

    def df_to_array(self, df, if_vix) -> np.array:
        price_array,tech_array,turbulence_array = self.processor.df_to_array(df,
                                                  self.tech_indicator_list,
                                                  if_vix)
        #fill nan with 0 for technical indicators
        tech_nan_positions = np.isnan(tech_array)
        tech_array[tech_nan_positions] = 0

        return price_array, tech_array, turbulence_array

    def run(self, ticker_list, start_date, end_date, time_interval,
            technical_indicator_list, if_vix):
        data = self.processor.download_data(ticker_list, start_date, end_date, time_interval)
        data = self.processor.clean_data(data)
        data = self.processor.add_technical_indicator(data, technical_indicator_list)
        if if_vix:
            data = self.processor.add_vix(data)
        price_array, tech_array, turbulence_array = self.processor.df_to_array(data, if_vix)
        tech_nan_positions = np.isnan(tech_array)
        tech_array[tech_nan_positions] = 0

        return price_array, tech_array, turbulence_array
```

Step 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 the FinRL--Meta data processor, we automate the data cleaning process.
The cleaning processes of NaN data are usually different for various time frequencies. For **Low-frequency data**, except for a 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 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 (open, high, low, and close prices; volume) 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 were 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.

Step 3: Feature Engineering

Feature Engineering is the last part of the data layer. In this part, we automate the calculation of technical indicators by connecting the **Stockstats** or **TA-Lib** library with our data processor. Users can quickly add indicators from the Stockstats library, or add user-defined features.

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 all supported. Users can also quickly add indicators from other libraries, or add other user-defined features directly.

2.4. FinRL-Meta Tutorials

Stock Trading Task

We select the 30 constituent stocks in Dow Jones Industrial Average (DJIA), accessed at the beginning of our testing period. We use the Proximal Policy Optimization (PPO) algorithm of ElegantRL, Stable-baselines3 and RLlib, respectively, to train agents and use the DJIA index as the baseline.

- **Backtesting:** We use 1-minute data from 06/01/2021 to 08/15/2021 for training and data from 08/16/2021 to 08/31/2021 for validation (backtesting).
- **Paper Trading:** Then we retrain the agent using data from 06/01/2021 to 08/31/2021 and conduct paper trading from 09/03/2021 to 09/16/2021. The historical data and real-time data are accessed from the Alpaca's database and paper trading APIs.

In the backtesting stage (plot in 5-minute), both ElegantRL agent and Stable-baselines3 agent outperform DJIA in annual return and Sharpe ratio. The ElegantRL agent achieves an **annual return of 22.425%** and a **Sharpe ratio of 1.457**. The Stable-baselines3 agent achieves an **annual return of 32.106%** and a **Sharpe ratio of 1.621**. In the paper trading stage, the results are **consistent** with the backtesting results. Both the ElegantRL agent and the Stable-baselines3 agent **outperform the baseline.**

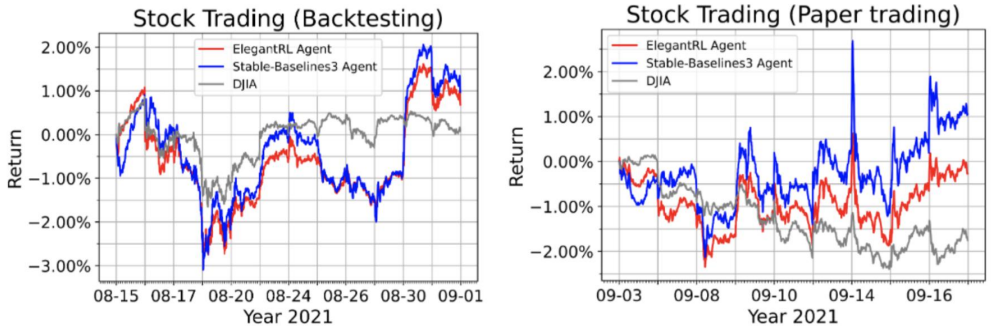


Figure 2: Cumulative returns (5-minute) of stock trading in backtesting and paper trading.

	ElegantRL [9]	**Stable-baselines3** [10]	**DJIA**
Cumul. return	0.968% / -0.652%	1.335% / 0.191%	0.099% / -1.56%
Annual return	22.425% / -16.746%	32.106% / 5.492%	2.108% / -35.522%
Annual volatility	15.951% / 14.113%	19.871% / 15.953%	9.196% / 9.989%
Sharpe ratio	1.457 / -1.399	1.621 / 0.447	0.289 / -4.894
Max drawdown	-2.657% / -1.871%	-2.932% / -1.404%	-1.438% / -2.220%

Table 2: Performance of backtesting (red) and paper trading (blue) for stock trading.

Cryptocurrency Trading Task

We select top 10 market cap cryptocurrencies, the top 10 market cap cryptocurrencies as of Oct 2021 are: Bitcoin (BTC), Ethereum (ETH), Cardano (ADA), Binance Coin (BNB), Ripple (XRP), Solana (SOL), Polkadot (DOT), Dogecoin (DOGE), Avalanche (AVAX), Uniswap (UNI). We use the PPO algorithm of ElegantRL to train an agent and use the Bitcoin (BTC) price as the baseline.

- **Backtesting:** We use 5-minute data from 06/01/2021 to 08/14/2021 for training and data from 08/15/2021 to 08/31/2021 for validation (backtesting).

- **Paper Trading:** Then we retrain the agent using data from 06/01/2021 to 08/31/2021 and conduct paper trading from 09/01/2021 to 09/15/2021. The historical data and real-time data are accessed from Binance.

In the backtesting stage (plot in 5-minute), the ElegantRL agent outperforms the benchmark (BTC price) in most performance metrics. It achieves an annual return of 360.823% and a Sharpe ratio of 2.992. The ElegantRL agent also outperforms the benchmark (BTC price) in the paper trading stage, which is consistent with the backtesting results.

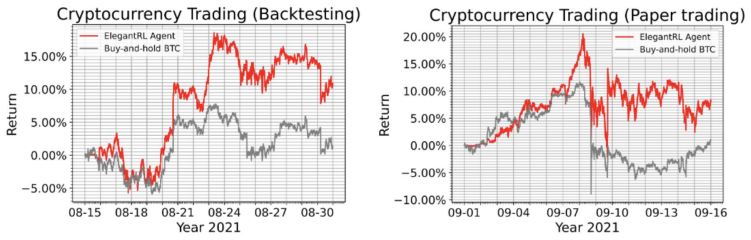


Figure 3: Cumulative returns (5-minute) of cryptocurrency trading in backtesting and paper trading.

	ElegantRL [9]	**BTC buy and hold**
Cumul. return	10.857% / 4.844%	1.332% / -1.255%
Annual return	360.823% / 121.380%	21.666% / 5.492%
Annual volatility	59.976% / 65.857%	47.410% / 57.611%
Sharpe ratio	2.992 / 1.608	0.657 / -0.113
Max drawdown	-6.396% / -10.474%	-7.079% / -14.849%

Table 3: Performance of backtesting (red) and paper trading (blue) for cryptocurrency trading.

Cryptocurrency Trading Exercises

Two colab notebooks are present for the reader to practice on: one with the answers and one without. Try to do the exercises by yourself. This should allow the reader to build a backtested agent.

Exercises:
https://drive.google.com/file/d/17FZCQVHCX57bDHTd7GwD2t74HPDVi2uf/view?usp=sharing

Answers:
https://drive.google.com/file/d/1R8qKvrXHM81UQp1fRcSvsOf1Ppd27cnh/view?usp=sharing

Trade Execution and Liquidation

1. Introduction

Trade execution is when a buy or sell order gets fulfilled. In order for a trade to be executed, an investor who trades using a brokerage account would first submit a buy or sell order, which then gets sent to a broker. On behalf of the investor, the broker would then decide which market to send the order to. Once the order is in the market and it gets fulfilled, only then can it be considered executed.

Metrics to measure Trade Execution:

- Profit and Loss (PnL): the overall profit obtained during the transaction;
- Implementation Shortfall: The difference between the total return of the trading algorithm and the total return of all transactions in the first place;
- Sharp ratio: Average return divided by standard deviation of return.

Common trading strategies:

- Time-Weighted Average Price (TWAP): Execute evenly in time;
- Volume-Weighted Average Price (VWAP): It is also a problem worthy of study to execute evenly on the transaction volume, and to obtain the corresponding execution plan accurately;
- Submit and Leave (SnL): place a fixed sell order, and then wait for it to be filled. If the final deal is not completed, sell it at the full market price;
- Almgren-Chriss: Executed according to the optimal analytical strategy under a certain price movement assumption.

2. Almgren and Chriss model

The problem of an optimal liquidation strategy is investigated by using the Almgren-Chriss market impact model on the background that the agents liquidate assets completely in a given time frame. The impact of the stock market is divided into three components: unaffected price process, permanent impact, and temporary impact. The stochastic component of the price process exists, but is eliminated from the mean-variance. The price process permits linear functions of permanent and temporary price. Therefore, the model serves as the trading environment such that when agents make selling decisions, the environment would return price information.

- The price process of the Almgren and Chriss model is as follows:

$$P_k = P_{k-1} + \sigma \cdot \tau^{1/2} \xi_k - \tau \cdot g(n_k/\tau),\ k = 1,...,N$$

  where σ represents the volatility of the stock, ξ_k are random variables with zero mean and unit variance, $g(v)$ is a function of the average rate of the trading, $v = n_k/\tau$ during time interval t_{k-1} to t_k, n_k is the number of shares to sell during time interval t_{k-1} to t_k, N is the total number of trades and $\tau = T/N$.
- Inventory process:

$$x_{tk} = X - \Sigma_{j=1}\, n_j,$$

  where x_{tk} is the number of shares remaining at time t_k, with $x_T = 0$.
- Linear permanent impact function:

$$g(v) = \gamma \cdot v,$$

where $v = n_k/\tau$.

- Temporary impact function:

$$h(n_k/\tau) = \varepsilon \cdot sgn(n_k) + (\eta/\tau) \cdot n_k$$

where a reasonable estimate of ε is the fixed costs of selling, and η depends on internal and transient aspects of the market microstructure.

- Parameters σ, γ, η, ε, time frame T, number of trades N are set at t = 0.

3. Liquidation as a MDP Problem

Consider the stochastic and interactive nature of the trading market, we model the stock trading process as a Markov decision process, which is specified as follows:

- State **s = [r,m,l]**: a set that includes the information of the log–return $r \in R_+^D$, where D is the number of days of log-return, and the remaining number of trades m normalized by the total number of trades, the remaining number of shares l, normalized by the total number of shares. The log-returns capture information about stock prices before time t_k, where k is the current step. It is important to note that in real world trading scenarios, this state vector may hold more variables.
- Action **a**: we interpret the action a_k as a fractional share. In this case, the actions will take continuous values in between 0 and 1.
- Reward **R(s,a)**: to define the reward function, we use the difference between two consecutive utility functions. The utility function is given by:

$$U(x) = E(x) + \lambda \cdot V(x),$$

$$E(x) = \Sigma_{k=1} \tau \cdot x_k \cdot g(n_k/\tau) + \Sigma_{k=1}\, n_k \cdot h(n_k/\tau),$$

$$V(x) = \sigma^2 \cdot \Sigma_{k=1} \tau \cdot x_k^2,$$

where λ is the risk aversion level, and x is the trading trajectory or the vector of shares remaining at each time step k. After each time step, we compute the utility using the equations for E(x) and V(x) from the Almgren and Chriss model for the remaining time and inventory while holding the parameter λ constant. Denotes the optimal trading trajectory computed at time t by x_t^*, we define the reward as:

$$R_t = U_t(x_t^*) - U_{t+1}(x_{t+1}^*).$$

- Policy $\pi(s)$: The liquidation strategy of stocks at state s. It is essentially the distribution of selling fraction a at state s.
- Action-value function $Q_\pi(s,a)$: the expected reward for using action a at state s, following policy π.

4. Multi-agent deep reinforcement learning for liquidation strategy analysis

- States **s = [r,m,l]** : in a multi-agent environment, the state vector should have information about the remaining stocks of each agent, the state vector at time t_k would be:

$$[r_{k-D},\ldots,r_{k-1},r_k,m_k,l_{1,k},\ldots,l_{N,k}].$$

$r_k = \log(P_k / P_{k-1})$ is the log-return at time t_k.

$m_k = N_k/N$ is the number of trades remaining at time t_k normalized by the total number of trades. $l_{j,k} = x_{j,k} /x_j$ is the remaining number of shares for agent j at time t_k normalized by the total number of shares.

- Action **a**: using the interpretation in Section \ref{DPL:liquidatoin}, we can determine the number of shares to sell for each agent at every time step using:

$$n_{j,k} = a_{j,k} \cdot x_{j,k},$$

where $x_{j,k}$ is the number of remaining shares at time t_k for agent j.

- Reward **R(s,a)**: denotes the optimal trading trajectory computed at time t for agent j by $x_{j,k}^*$ we define the reward as:

$$R_{j,t} = U_{j,t}(x_{j,t}^*)-U_{j,t+1}(x_{j,t+1}^*).$$

- Observation **O**: Each agent only observes limited state information. In other words, in addition to the environment information, each agent only knows its own remaining shares, but not other agents' remaining shares. The observation vector at time t_k for agent j is:

$$O_{j,k} = [r_{k-D},...,r_{k-1},r_k,m_k,l_{j,k}].$$

5. Demo: Multi-Agent Reinforcement Learning for Liquidation Strategy Analysis

```
import utils

# Get the default financial and AC Model parameters
financial_params, ac_params = utils.get_env_param()
```

```
financial_params
```

Financial Parameters

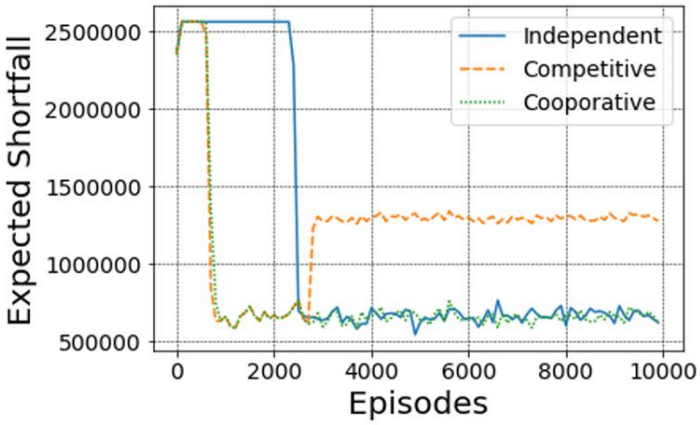
Annual Volatility:	12%	Bid-Ask Spread:	0.125
Daily Volatility:	0.8%	Daily Trading Volume:	5,000,000

```
ac_params
```

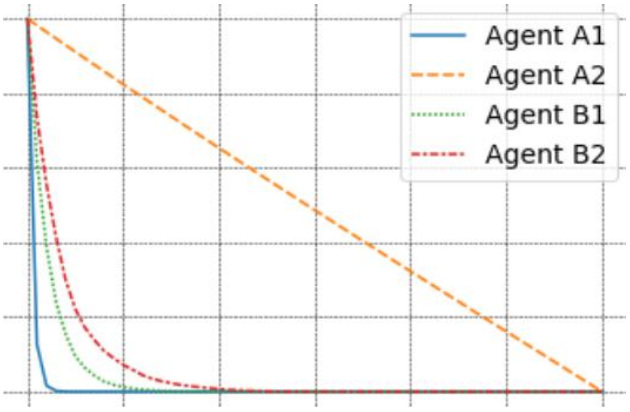
Almgren and Chriss Model Parameters

Total Number of Shares for Agent1 to Sell:	500,000	Fixed Cost of Selling per Share:	$0.062
Total Number of Shares for Agent2 to Sell:	500,000	Trader's Risk Aversion for Agent 1:	1e-06
Starting Price per Share:	$50.00	Trader's Risk Aversion for Agent 2:	0.0001
Price Impact for Each 1% of Daily Volume Traded:	$2.5e-06	Permanent Impact Constant:	2.5e-07
Number of Days to Sell All the Shares:	60	Single Step Variance:	0.144
Number of Trades:	60	Time Interval between trades:	1.0

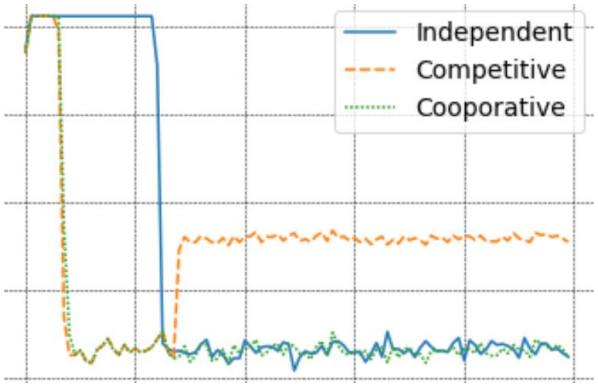
```
Episode [100/1300]      Average Shortfall for Agent1: $1,172,575.47
Episode [100/1300]      Average Shortfall for Agent2: $1,183,832.73
Episode [200/1300]      Average Shortfall for Agent1: $1,281,148.59
Episode [200/1300]      Average Shortfall for Agent2: $1,281,025.43
Episode [300/1300]      Average Shortfall for Agent1: $1,281,250.00
Episode [300/1300]      Average Shortfall for Agent2: $1,281,250.00
Episode [400/1300]      Average Shortfall for Agent1: $1,281,250.00
Episode [400/1300]      Average Shortfall for Agent2: $1,281,250.00
Episode [500/1300]      Average Shortfall for Agent1: $1,281,250.00
Episode [500/1300]      Average Shortfall for Agent2: $1,281,250.00
Episode [600/1300]      Average Shortfall for Agent1: $1,281,250.00
Episode [600/1300]      Average Shortfall for Agent2: $1,281,250.00
Episode [700/1300]      Average Shortfall for Agent1: $1,227,339.91
Episode [700/1300]      Average Shortfall for Agent2: $1,253,734.24
Episode [800/1300]      Average Shortfall for Agent1: $415,623.02
Episode [800/1300]      Average Shortfall for Agent2: $433,944.85
Episode [900/1300]      Average Shortfall for Agent1: $314,968.49
Episode [900/1300]      Average Shortfall for Agent2: $317,854.76
Episode [1000/1300]     Average Shortfall for Agent1: $318,731.56
Episode [1000/1300]     Average Shortfall for Agent2: $317,495.71
Episode [1100/1300]     Average Shortfall for Agent1: $329,135.85
Episode [1100/1300]     Average Shortfall for Agent2: $333,255.71
Episode [1200/1300]     Average Shortfall for Agent1: $300,993.44
Episode [1200/1300]     Average Shortfall for Agent2: $301,320.57
Episode [1300/1300]     Average Shortfall for Agent1: $294,413.69
...
Average Implementation Shortfall for Agent1: $829,225.39


Average Implementation Shortfall for Agent2: $833,877.00
```



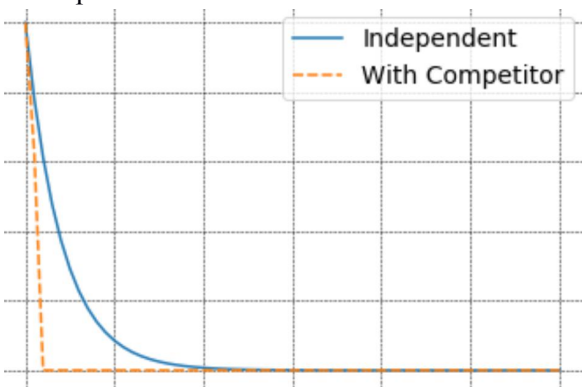
Comparison of expected implementation shortfalls: there are three agents A, B_1 and B_2. The expected shortfall of agent A is higher than the sum of two expected shortfalls B_1 and B_2.



Trading trajectory: comparing to their original trading trajectories, their current trading trajectories are closer to each other when they are trained in a multi-agent environment.



Cooperative and competitive relationships: if two agents are in a cooperative relationship, the total expected shortfall is not better than training with independent reward functions. If two agents are in a competitive relationship, they would first learn to minimize expected shortfall, and then malignant competition leads to significant implementation shortfall increment.



Trading trajectory: comparing to independent training, introducing a competitor makes the host agent learn to adapt to new environment and sell all shares of stock in the first two days

6. Profit-oriented Trade Execution Methods

[1] Jeong G, Kim H Y. Improving financial trading decisions using deep Q-learning: Predicting the number of shares, action strategies, and transfer learning. Expert Systems with Applications, 2019, 117: 125-138.
[2] Deng Y, Bao F, Kong Y, et al. Deep direct reinforcement learning for financial signal representation and trading[J]. IEEE transactions on neural networks and learning systems, 2016, 28(3): 653-664.
[3] Zhang Z, Zohren S, Roberts S. Deep reinforcement learning for trading. The Journal of Financial Data Science, 2020, 2(2): 25-40.

[4] Wei H, Wang Y, Mangu L, et al. Model-based reinforcement learning for predictions and control for limit order books. arXiv preprint arXiv:1910.03743, 2019. Zhang Chuheng: [Reinforcement Learning187] Model-based RL for LOB
[5] Shen Y, Huang R, Yan C, et al. Risk-averse reinforcement learning for algorithmic trading. 2014 IEEE Conference on Computational Intelligence for Financial Engineering & Economics (CIFEr). IEEE, 2014: 391-398. Additional Conference Controlling risk, discusses the DQN approach with risk control.

Conclusion

We followed the DataOps paradigm and developed a FinRL-Meta framework. FinRL-Meta provides open-source data engineering tools and hundreds of market environments with multiprocessing simulation.

2.5. ElegantRL Overview

ElegantRL is designed for researchers and practitioners with finance-oriented optimizations.

1. ElegantRL implements **state-of-the-art DRL algorithms** from scratch, including both discrete and continuous ones, and provides user-friendly tutorials in Jupyter Notebooks.
2. The ElegantRL performs DRL algorithms under the **Actor-Critic framework**
3. The ElegantRL library enables researchers and practitioners to pipeline the disruptive "design, development and deployment" of DRL technology.

2.6. ElegantRL Code Structure

Part 1. Overview: File Structure and Functions

The file structure of ElegantRL is shown in Fig. 1:

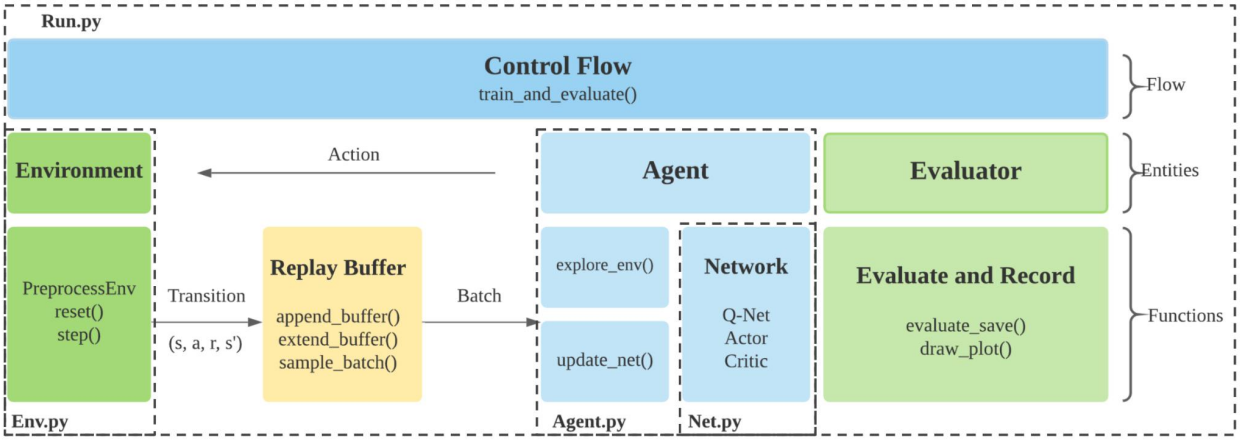


Figure 1. An agent in Agent.py uses networks in Net.py and is trained in Run.py by interacting with an environment in Env.py.

Env.py: it contains the environments with which the agent interacts.

- A PreprocessEnv class for gym-environment modification.
- A self-created stock trading environment as an example for user customization.

Net.py: There are three types of networks:

- Q-Net,
- Actor Network,
- Critic Network,

Each includes a base network for inheritance and a set of variations for different algorithms.

Agent.py: it contains agents for different DRL algorithms.

Run.py: it provides basic functions for the training and evaluating process:

- Parameter initialization,
- Training loop,
- Evaluator.

As a high-level overview, the relations among the files are as follows. Initialize an environment in Env.py and an agent in Agent.py. The agent is constructed with Actor and Critic networks in Net.py. In each training step in Run.py, the agent interacts with the environment, generating transitions that are stored into a Replay Buffer. Then, the agent fetches transitions from the Replay Buffer to train its networks. After each update, an evaluator evaluates the agent's performance and saves the agent if the performance is good.

Part 2. Implementations of DRL Algorithms

This part describes **DQN-series algorithms and DDPG-series algorithms**, respectively. Each DRL algorithm agent follows a hierarchy from its base class.

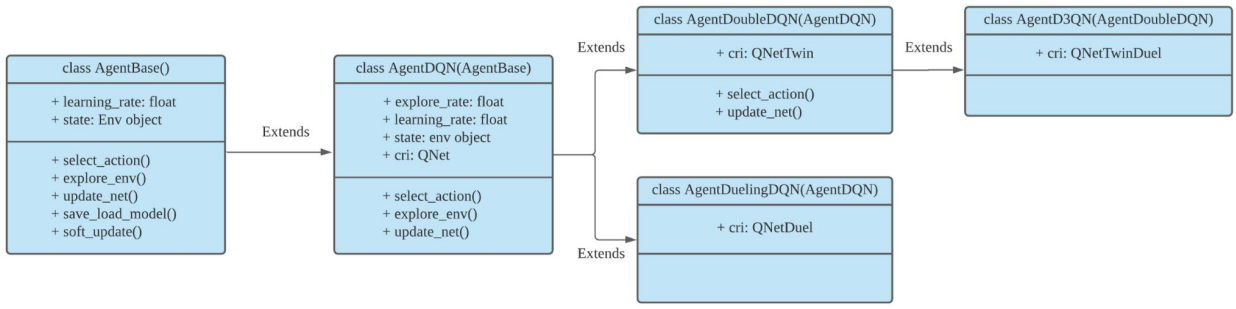


Figure 2. The inheritance hierarchy of DQN-series algorithms.

As shown in Fig. 2, the inheritance hierarchy of the DQN-series algorithms is as follows:

- **AgentDQN:** a standard DQN agent.
- **AgentDoubleDQN:** a Double-DQN agent with two Q-Nets for reducing overestimation, inheriting from AgentDQN.
- **AgentDuelingDQN:** a DQN agent with a different Q-value calculation, inheriting from AgentDQN.
- **AgentD3QN:** a combination of AgentDoubleDQN and AgentDuelingDQN, inheriting from AgentDoubleDQN.

```
class AgentBase:
    def init(self);
    def select_action(states); # states = (state, …)
    def explore_env(env, buffer, target_step, reward_scale, gamma);
    def update_net(buffer, max_step, batch_size, repeat_times);
    def save_load_model(cwd, if_save);
    def soft_update(target_net, current_net);


class AgentDQN:
    def init(net_dim, state_dim, action_dim);
    def select_action(states); # for discrete action space
    def explore_env(env, buffer, target_step, reward_scale, gamma);
    def update_net(buffer, max_step, batch_size, repeat_times);
    def save_or_load_model(cwd, if_save);


class AgentDuelingDQN(AgentDQN):
    def init(net_dim, state_dim, action_dim);


class AgentDoubleDQN(AgentDQN):
    def init(self, net_dim, state_dim, action_dim);
    def select_action(states);
    def update_net(buffer, max_step, batch_size, repeat_times);


class AgentD3QN(AgentDoubleDQN): # D3QN: Dueling Double DQN
    def init(net_dim, state_dim, action_dim);
```

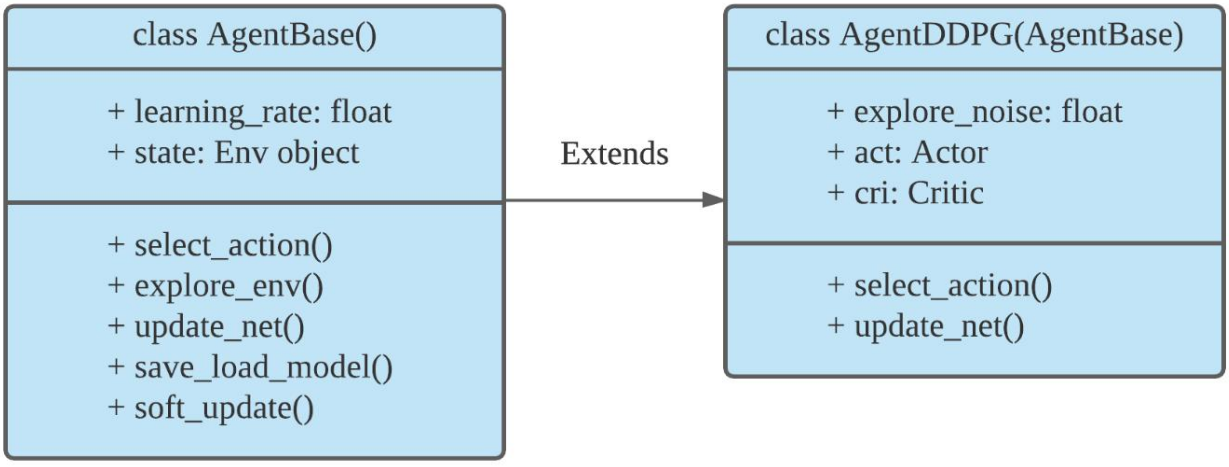


Figure 3. The inheritance hierarchy of DDPG-series algorithms.

As shown in Fig. 3, the inheritance hierarchy of the DDPG-series algorithms is as follows

- **AgentBase:** a base class for all Actor-Critic agents.
- **AgentDDPG:** a DDPG agent, inheriting from AgentBase.

```
class AgentBase:
    def init(self);
    def select_action(states); # states = (state, …)
    def explore_env(env, buffer, target_step, reward_scale, gamma);
    def update_net(buffer, max_step, batch_size, repeat_times);
    def save_load_model(cwd, if_save);
    def soft_update(target_net, current_net);


class AgentDDPG(AgentBase):
    def init(net_dim, state_dim, action_dim);
    def select_action(states);
    def update_net(buffer, max_step, batch_size, repeat_times);
```

Applying such a hierarchy in building DRL agents effectively improves **lightweightness and effectiveness**. Users can easily design and implement new agents in a similar flow.

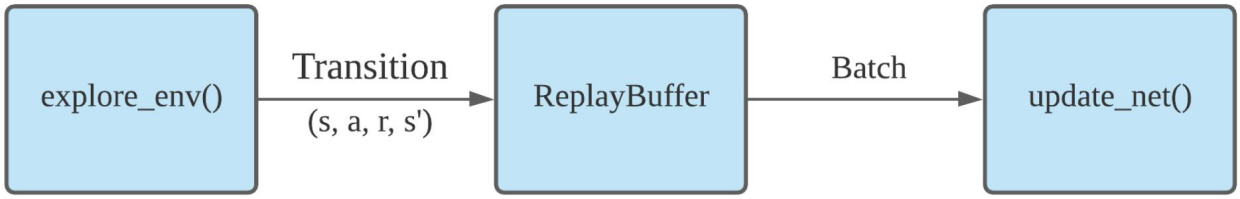


Figure 4. The data flow of training an agent.

Basically, an agent has two fundamental functions, and the data flow is shown in Fig.4:

- **explore_env():** it allows the agent to interact with the environment and generates transitions for training networks.
- **update_net():** it first fetches a batch of transitions from the Replay Buffer, and then train the network with backpropagation.

Part 3. Training Pipeline

Two major steps to train an agent:

1. Initialization:

- **hyper-parameters args.**
- **env = PreprocessEnv() :** creates an environment (in the OpenAI gym format).
- **agent = AgentXXX() :** creates an agent for a DRL algorithm.
- **evaluator = Evaluator() :** evaluates and stores the trained model.
- **buffer = ReplayBuffer() :** stores the transitions.

2. Then, the training process is controlled by a while-loop:

- **agent.explore_env(…):** the agent explores the environment within target steps, generates transitions, and stores them into the ReplayBuffer.
- **agent.update_net(…):** the agent uses a batch from the ReplayBuffer to update the network parameters.
- **evaluator.evaluate_save(…):** evaluates the agent's performance and keeps the trained model with the highest score.

The while-loop will terminate when the conditions are met, e.g., achieving a target score, maximum steps, or manually breaks.

Part 4. Testing Example: BipedalWalker-v3

BipedalWalker-v3 is a classic task in robotics that performs a fundamental skill: moving. The goal is to get a 2D biped walker to walk through rough terrain. BipedalWalker is considered to be a difficult task in the continuous action space, and there are only a few RL implementations that can reach the target reward.

Step 1: Install ElegantRL

```
pip install git+https://github.com/AI4Finance-LLC/ElegantRL.git
```

Step 2: Import Packages

- **ElegantRL**
- **OpenAI Gym:** a toolkit for developing and comparing reinforcement learning algorithms.
- **PyBullet Gym:** an open-source implementation of the OpenAI Gym MuJoCo environments.

```
from elegantrl.run import *

from elegantrl.agent import AgentGaePPO

from elegantrl.env import PreprocessEnv

import gym

gym.logger.set_level(40) # Block warning
```

Step 3: Specify Agent and Environment

- **args.agent:** firstly chooses a DRL algorithm, and the user is able to choose one from a set of agents in agent.py
- **args.env:** creates and preprocesses an environment, and the user can either customize own environment or preprocess environments from OpenAI Gym and PyBullet Gym in env.py.

```
args = Arguments(if_on_policy=False)

args.agent = AgentGaePPO() # AgentSAC(), AgentTD3(), AgentDDPG()

args.env = PreprocessEnv(env=gym.make('BipedalWalker-v3'))

args.reward_scale = 2 ** -1 # RewardRange: -200 < -150 < 300 < 334

args.gamma = 0.95

args.rollout_num = 2 # the number of rollout workers (larger is not
always faster)
```

Step 4: Train and Evaluate the Agent

The training and evaluating processes are inside the function **train_and_evaluate__multiprocessing(args)**, and the parameter is **args**. It includes two fundamental objects in DRL:

- **agent,**

- **environment (env).**

And the parameters for training:

- **batch_size,**
- **target_step,**
- **reward_scale,**
- **learning rate**
- **gamma, etc.**

Also the parameters for evaluation:

- **break_step,**
- **random_seed, etc.**

```
train_and_evaluate__multiprocessing(args) # the training process will
terminate once it reaches the target reward.
```

Step 5: Testing Results

After reaching the target reward, we generate the frame for each state and compose frames as a video result. From the video, the walker is able to move forward constantly.

```
for i in range(1024):
    frame = gym_env.render('rgb_array')
    cv2.imwrite(f'{save_dir}/{i:06}.png', frame)

    states = torch.as_tensor((state,), dtype=torch.float32,
device=device)
    actions = agent.act(states)
    action = actions.detach().cpu().numpy()[0]
    next_state, reward, done, _ = env.step(action)


    if done:
        state = env.reset()
    else:
        state = next_state
```

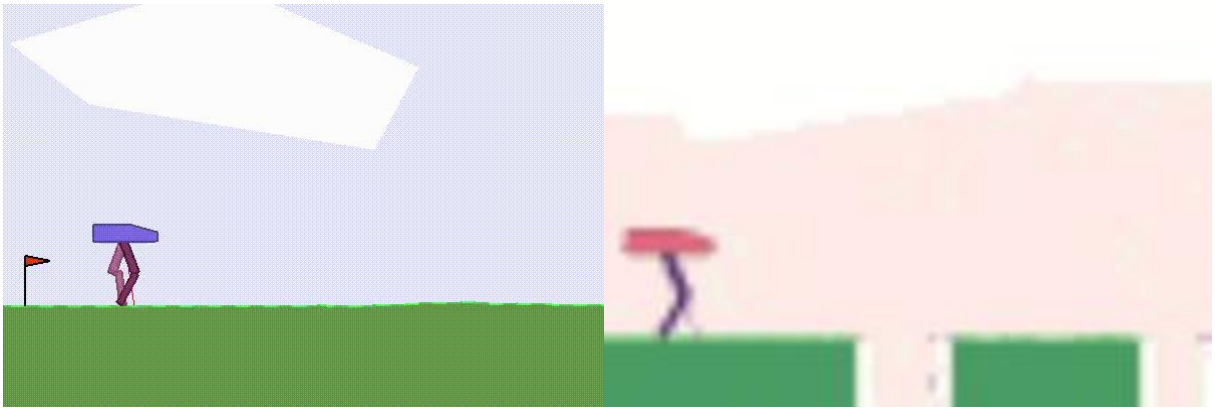


Figure 5. (left) An agent with random actions. (right) A PPO agent in ElegantRL.

Check out the Colab codes for this BipedalWalker-v3 demo.

Conclusion

ElegantRL is developed for researchers and practitioners with the following advantages:

- Lightweight: the core codes <1,000 lines (check elegantrl/tutorial), using PyTorch (train), OpenAI Gym (env), NumPy, Matplotlib (plot).
- Efficient: in many testing cases, we find it more efficient than Ray RLlib.
- Stable: much more stable than Stable Baselines 3. Stable Baselines 3 can only use a single GPU, but ElegantRL can use 1~8 GPUs for stable training.

ElegantRL implements the following model-free deep reinforcement learning (DRL) algorithms:

- DDPG, TD3, SAC, PPO, PPO (GAE),REDQ for continuous actions
- DQN, DoubleDQN, D3QN, SAC for discrete actions
- QMIX, VDN; MADDPG, MAPPO, MATD3 for multi-agent environment

For the details of DRL algorithms, please check out the educational webpage OpenAI Spinning Up.

We also developed demos such as: **ElegantRL Demo: Stock Trading Using DDPG.** For more details about it**,** please visit **Part 1 and Part 2**.

3. Senior/Capstone Projects

The questions to answer before embarking on the project: (These are our Interview Screening Questions)

1. What is a Markov Decision Process? Give a definition of MDP. What is a policy? Give a definition of the Bellman Equation. What are some of the ways to learn optimal policy?
2. Can you walk through how to learn optimal policy with MDP through Neural Network? How do we represent those states and policies in terms of NN?
3. What is the difference between Reinforcement learning & Deep Reinforcement Learning? What is the difference between Q-learning & Deep Q-learning?
4. How to implement DQN?
5. We want to apply DRL to finance (FinRL). Can you find some recent applications of deep reinforcement learning in financial markets, for example, automated trading, portfolio allocation, cryptocurrency (BTC), hedging, high-frequency (minute level or tick data)? What are the critic-based approach, actor-based approach, or actor-critic approach algorithms?
6. Can you explain the evolution line of Deep Reinforcement Learning algorithms? DRL is always about coming up with state-of-the-art performance algorithms (DQN, DDPG, Policy Gradient, A2C, PPO, TD3, SAC, etc. the advantages and disadvantages for each algorithm, for example, if you think DQN < DDPG < TD3 < SAC, explain the reasons)

FinRL

FinRL is the first open-source framework to demonstrate the great potential of applying deep reinforcement learning in quantitative finance. We help practitioners establish the development pipeline of trading strategies using **deep reinforcement learning (DRL). A DRL agent learns by continuously interacting with an environment in a trial-and-error manner, making sequential decisions under uncertainty, and achieving a balance between exploration and exploitation.**

1. Technical Indicators & Statistical Features

Using *feature extraction* methods to obtain attributes that can be used in *states*. For example, the TA-Lib library (https://github.com/mrjbq7/ta-lib, already used by FinRL) provides standard calculations of technical indicators. The reason classic technical analysis works at all is probably only the self-fulfilling prophecy. From that point of view, it might be a good idea to start with the popular indicators. In trading, candlestick patterns are often used for trading decisions, too. You could define features for those patterns (libraries like TA-lib provide such functions) or provide the model the features that describe those patterns like upper and lower shadow of the candle, real body of the candle and full length of the candle.

Also the so-called hl2, hlc3, ohlc4 prices and orderbook data are typically used in trading and might be valuable features.

Here are some examples about feature extraction methods and indicator groups, you are more than welcome to try other methods as well.

Feature extraction methods: Principal Components Analysis (PCA), Independent Component Analysis (ICA), Linear Discriminant Analysis (LDA), Locally Linear Embedding (LLE), t-distributed Stochastic Neighbor Embedding (t-SNE), Autoencoders

Indicator Groups include: Overlap Studies, Momentum Indicators, Volume Indicators, Volatility Indicators, Price Transform, Cycle Indicators, Pattern Recognition

There are also many statistical & mathematical features that can be considered. Cointegration (between assets / Augmented Dickey–Fuller test, Johansen test, Engle-Granger test), correlation (between assets / Pearson), variance, deviation, entropy, logarithmic return, fast fourier transform and wavelet transform to just name some. You will find more inspiration here:

- https://tsfresh.readthedocs.io/en/latest/text/list_of_features.html
- https://github.com/fraunhoferportugal/tsfel

Hidden states with Hidden Markov model (https://hmmlearn.readthedocs.io/en/0.2.0/auto_examples/plot_hmm_stock_analysis.html)

Another approach to generate features can be clustering. Two examples using clustering:

Ding, Fengqian & Luo, Chao. (2020). An Adaptive Financial Trading System Using Deep Reinforcement Learning With Candlestick Decomposing Features. IEEE Access.

Clustering to find support & resistance levels:

https://towardsdatascience.com/using-k-means-clustering-to-create-support-and-resistance-b13fdeeba12

2. Rewards Functions

The reward fed to the RL agent is completely governing its behavior, so a wise choice of the reward shaping function is critical for good performance. There are quite a number of rewards one can choose from or combine, from risk-based measures to profitability or cumulative return, number of trades per interval, etc. The RL framework accepts any sort of reward: the denser, the better. (Millea, A. Deep Reinforcement Learning for Trading—A Critical Survey. Data 2021, 6, 119. https://doi.org/10.3390/data6110119)

An overview over certain reward functions can be found in the paper Sadighian, J. (2020). Extending Deep Reinforcement Learning Frameworks in Cryptocurrency Market Making arXiv:2004.06985:

- *PnL-based Rewards (Unrealized PnL, Unrealized PnL with Realized Fills, Asymmetrical Unrealized PnL with Realized Fills, Asymmetrical Unrealized PnL with Realized Fills and Ceiling, Realized PnL Change)*
- *Goal-based Rewards (Trade Completion)*
- *Risk-based Rewards (Differential Sharpe Ratio)*

The Deflated Sharpe Ratio from M López De Prado is worth considering too. It corrects for two leading sources of performance inflation: Non-Normally distributed returns and selection bias under multiple testing.(https://papers.ssrn.com/sol3/papers.cfm?abstract_id=2465675)

The reward function is the incentive mechanism for an agent to learn a better action. There are many forms of the reward function. FinRL supports user-defined reward functions to include
risk factor or transaction cost term such as in Deep hedging, Deep Direct Reinforcement Learning for Financial Signal Representation and Trading, and Deep reinforcement learning for trading.

In this section, you are required to try all the above-mentioned reward functions, then compare the results. You are also required to create your own reward function to achieve the best performance as you can. A combination of different reward functions might be a valuable approach. Besides the above-mentioned ideas, you might want to consider:

- The immediate and long-term reward
- Incentivizing actions / Penalizing (too long) holding/time without an open position.
- Optimal total of trades (a rough estimation on how often the model should trade - 1D timeframe vs. 5m timeframe should probably be traded with a different frequency) and penalizing too big deviation from this optimum.
- Reward in relation to buy-and-hold. You could penalize if it doesn't beat buy-and-hold metrics (of the day).

3. Alternative Datasets

Financial-News / Sentiment analysis.

Financial news is frequently discussed and plays an important role in evaluating financial markets. In this section, we would like to leverage the financial news datasets and dive deeper into how such information can help FinRL to make better decisions. We provide three financial datasets as follows: Daily Financial News for 6000+ Stocks, US Financial News Articles, and Daily News for Stock Market. You are required to add such information as input features into the FinRL model and compare the result. A keyword for this is (News) Sentiment Analysis.

Company fundamentals

Company fundamentals can be valuable information sources too. Like financial statements, market capitalization, SEC filings, press releases (dates), etc.

Social Media data/sentiment

Social media activities can be a valuable indicator for models too. For example Reddit subscribers, Twitter followers, positive/negative mentions of the stock (sentiment analysis), etc. When using these data one should be aware that those activities can be distorted by social media bots. A reliable data source that has measures in place to filter such distortions is important.

Cryptocurrency specific data

Cryptocurrencies offer some data that are unique to them. On-Chain data are related to transactions, mining activities, and network stats. Another data point can be developer activity on Github or the number of listings on exchanges for example.

You will find many data aggregator services offering those alternative historical data in real-time. This list could go on forever. The creativity on alternative data used for trading has no limits and you will even find the use of more seemingly unrelated data usage like astrology data, weather data, and sun activity being applied in trading.

4. Seasonality Information

Providing the model information of seasonality can open up the possibility for the model to discover patterns. Data could be day of the week, day of the month, month, year, week of the year, hour, and holidays (Christmas, Black Friday, etc.)

5. Volatility Features

There are several established functions to estimate the volatility of an asset:

- Realized
- Parkinson
- Garman-Klass
- Roger-Satchell
- Garman-Klass-Yang-Zhang
- Yang-Zhang

See https://www.kaggle.com/yamqwe/crypto-prediction-volatility-features for implementations.

6. Feature Importance / Feature Selection

To improve performance of the model this part is essential. More input features often make the decision task for the model more challenging. This is known as the curse of dimensionality. So it's essential to only keep important features. There are different techniques that can be applied to determine the right set of features.

Overview of different approaches:

- https://jundongl.github.io/scikit-feature/algorithms.html

- https://www.kdnuggets.com/2021/06/feature-selection-overview.html
- Solorio-Fernández, S., Carrasco-Ochoa, J.A. & Martínez-Trinidad, J.F. A review of unsupervised feature selection methods. *Artif Intell Rev* **53,** 907–948 (2020). https://doi.org/10.1007/s10462-019-09682-y
- Liu, Kunpeng, u. a. *Efficient Reinforced Feature Selection via Early Stopping Traverse Strategy*. 2021. arXiv:2109.14180
- Fan, Wei, u. a. *AutoFS: Automated Feature Selection via Diversity-aware Interactive Reinforcement Learning*. 2020. arXiv:2008.12001
- Zhao, Xiaosa, u. a. *Simplifying Reinforced Feature Selection via Restructured Choice Strategy of Single Agent*. 2020. arXiv:2009.09230
- Wang, Xiaoyang, u. a. *Robusta: Robust AutoML for Feature Selection via Reinforcement Learning*. 2021. arXiv:2101.05950

7. Cross-Validation

"There are many different ways one can do cross-validation, and it is the most critical step when building a good machine learning model which is generalizable when it comes to unseen data." - Thakur, A. (2020). Approaching (almost) any machine learning problem.

Cross-Validation is essential to prevent overfitting. There are certain techniques suited for time series data / financial data explained here:
https://medium.com/@samuel.monnier/cross-validation-tools-for-time-series-ffa1a5a09bf9

Great library which implements two cross-validation algorithms suitable to evaluate machine learning models based on time series datasets: https://github.com/sam31415/timeseriescv

8. Synthetic Price Series for Initial Training and/or Testing

Training first on rather simple idealized synthetic prices before feeding real data might be beneficial to learn the agent the "basics". Also, it's great for testing functionality.

- Sine wave
- Trend curves
- Random walk
- Different types of autocorrelation
- Adding different degrees of noise/trend
- Recurring patterns

You will find existing libraries/examples available for these tasks. For example:

- https://github.com/Nike-Inc/timeseries-generator
- https://github.com/TimeSynth/TimeSynth
- https://github.com/stefan-jansen/synthetic-data-for-finance

- https://towardsdatascience.com/time-series-analysis-creating-synthetic-datasets-cf008208e014

9. Different Approaches, Actions and Environments

Currently, the actions / environments focus on trading actions buy, sell and hold. There are more possibilities to explore:

- Only buy or sell.
- Only day-trading. Two approaches:
    - Force a sell at the end of the day and let the model find just entries
    - Force an open position at the start of the day and let the model find just the exit
- Use classic stop-loss and / or take-profit approaches (trailing, ATR based, etc.) along the model's trading decisions.
- Risk / volatility based position-sizing along the models decisions.
- Detecting / classifying market regimes / market conditions with DRL (or as feature) (https://www.twosigma.com/articles/a-machine-learning-approach-to-regime-modeling/ and Horvath, Blanka, u. a. *Clustering Market Regimes using the Wasserstein Distance*. 2021. arXiv:2110.11848)
- Pump and Dump / anomaly detection. (Nilsen, Andreas Isnes. *Limelight: real-time detection of pump-and-dump events on cryptocurrency exchanges using deep learning*. MS thesis. UiT Norges arktiske universitet, 2019.)
- statistical arbitrage also known as pairs trading
- portfolio optimization

Those concepts also reflect trading best practices.

10. Reproduce Experimental Results of Existing Papers

In the AI for finance society, there are many valuable works we would like you to study as well. Such works will be good assets for you to enhance your basic skills in understanding others' research work and to inspire your research ideas in future studies. Attached are research papers we recommend you reproduce:

Cryptocurrency portfolio management with deep reinforcement learning

A deep reinforcement learning framework for the financial portfolio management problem
Github repo: https://github.com/ZhengyaoJiang/PGPortfolio

Deep Reinforcement Learning for Trading

Sun, Shuo, u. a. *DeepScalper: A Risk-Aware Deep Reinforcement Learning Framework for Intraday Trading with Micro-level Market Embedding*. 2021. arXiv:2201.09058

Moraes Sarmento, Simão, und Nuno C. G. Horta. *A Machine Learning Based Pairs Trading Investment Strategy*. Springer, 2021.

FinRL-Meta

FinRL-Meta is a universe of market environments for data-driven financial reinforcement learning. Users can use FinRL-Meta as the metaverse of their financial environments.

Why FinRL-Meta?

- To reduce the simulation-reality gap: existing works use backtesting on historical data, while the real performance may be quite different when applying the algorithms to paper/live trading.
- To reduce the data pre-processing burden, so that quants can focus on developing and optimizing strategies.
- To provide benchmark performance and facilitate fair comparisons, providing a standardized environment will allow researchers to evaluate different strategies in the same way. Also, it would help researchers to better understand the "black-box" nature (deep neural network-based) of DRL algorithms.

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).
- Transparency: a virtual env that is invisible to the upper layer
- Flexibility and extensibility: Inheritance might be helpful here

We plan to build a **multi-agent based market simulator** that consists of over ten thousands of agents, namely, a FinRL-Meta. First, FinRL-Meta aims to build a universe of market environments, like the Xland environment and planet-scale climate forecast by DeepMind. To improve the performance for large-scale markets, we will employ GPU-based massive parallel simulation as Isaac Gym. Moreover, it will be interesting to explore the deep evolutionary RL framework to simulate the markets. **Our final goal is to provide insights into complex market phenomena and offer guidance for financial regulations through FinRL-Meta.**

1. Momentum Strategy with Deep Learning in Chinese Commodity Market

Goal: Study trading strategies with state-of-the-art DRL techniques trained on the FinRL platform. Specifically we want to improve momentum strategies by detecting trend reversals using deep reinforcement learning techniques.

The student will have the opportunity to gain market insights and real world quant trading experience.

Market:

1. Daily market data for all futures traded on CFFEX, SHFE, DCE, INE starting 04/01/2020, till now.
2. Tick-by-tick data for all above futures.

Both datasets in the period can be shared in FinRL-Meta.

Base lines:

1. A buy-and-hold strategy
2. Traditional momentum strategy

Key Steps:

1. Describe the problem in the deep reinforcement learning language
2. Choose the appropriate algorithm
3. Train and test the model, and possibly iterate through 1, 2, 3.

Performance Backtesting:

1. For inter-day strategy, the train period will be around 1.5 years, the validation period will be around 0.5 years, and the backtest period will be around 0.5 years.
2. For intraday strategy, the train period will be around 2 months, and the validation period will be about 1 month, and the backtest will be about 1 month, all on rolling windows.
3. We will look at performance on individual assets and performance on asset portfolios based on certain selection.

References:

1. “A Trading Strategy Using Deep Learning and Change point Detection”
2. “Deep Direct Reinforcement Learning for Financial Signal Representation and Trading”

2. Integrate Data Ingestion and Trading Actions

Open ended questions:

How to integrate data ingestion and trading actions with more popular trading platforms such as Metatrader (https://www.metatrader5.com/en)? REST API for prediction endpoint and broker data ingestion (batch/streaming)? Can the models be utilized by traders in robots (expert advisors) and trading indicators?

We would like you to think about these questions and come up with the implementation plans.

3. Expand Support for a Number of FinRL Dataprocessors and Brokers for Trading Operations.

Users can use DataProcessor in data_processor.py. Take **Binance** as an example.

```
DP = DataProcessor('binance')
   ticker_list = ['BTCUSDT', 'ETHUSDT', 'ADAUSDT', 'BNBUSDT']
   start_date = '2021-09-01'
   end_date = '2021-09-20'
   time_interval = '5m'
   technical_indicator_list = ['macd', 'rsi', 'cci', 'dx']  # self-defined technical indicator list is NOT supported yet
   if_vix = False
   price_array, tech_array, turbulence_array = DP.run(ticker_list, start_date, end_date,
                                                      time_interval, technical_indicator_list,
                                                      if_vix, cache=True)
```

We would like you to try out other markets and fill out the results.

4. FinRL-Meta Real-world Enhancement

In order to achieve the goal of FinRL-Meta, a universe of market environments for data-driven financial reinforcement learning. We encourage you to explore the following questions and add the corresponding features:

Provide signals to the traders

In real-world implementation, traders need to track FinRL's performance with certain instruments over time. Providing valid signals is a key enhancement for FinRL-Meta.

Millisecond data frequency support

Some brokers provide data at tick or millisecond frequency - how can FinRL be scaled to be able to provide trading operations at this frequency?

Futures contracts support

As a universe of market environments, FinRL-Meta will take the Futures contracts into account. How to manage Futures contracts and calculation of variation margin at the end of each trading day?

5. Incorporate the real-world market environment

1. How to incorporate the real-world market environment such as dealing with bid-ask spreads, slippage, etc. and providing trading operations to limit risk e.g. fixed and trailing loss-stops, take profit, pending and limit orders?

2. How to manage orders after it has been formed and sent to a trade server, e.g. the order can undergo the following stages:

a. Started — the order correctness has been checked, but it hasn't been yet accepted by the broker;
b. Placed — a dealer has accepted the order;
c. Partially filled — the order is filled partially;
d. Filled — the entire order is filled;
e. Canceled — the order is canceled by the client;
f. Rejected — the order is rejected by a dealer;
g. Expired — the order is canceled due to its expiration.

3. How can FinRL enable Netting (single open positions on a symbol) or Hedging (multiple open positions), with trades in the same direction or opposite.

4. How to incorporate fundamental analysis in FinRL?

5. How to conduct various experiments, record them, and systematically compare and contrast the results, and then fine tune the models?

6. How to determine whether technical indicators help or hinder FinRL predictions?
Users can set the technical indicator list as an empty list, train the model, and test the performance. If the return is higher than that case with a defined list, it helps, otherwise, no help.

6. Technical enhancement for FinRL-Meta

1. To support the model for different traders, transfer learning can certainly help to make it happen. How to transfer learning between models run by different traders?

Some hands on examples for transfer learning:
https://www.tensorflow.org/tutorials/images/transfer_learning

2. How to scale data ingestion, cleansing, transformation to features using DataOps?

3. Enable Fin-RL to use a data lake and data pipeline to be able to ingest batch or streaming data, and structured or unstructured data e.g. social media sentiment. Incorporate a data pipeline with e.g. bronze, silver, gold zones for data cleansing, transformation, feature creation etc.

4. How to scale ML model management, through development, testing, preproduction and production, whilst managing continuous training in production to cope with data drift/concept drift?

5. The success of FinRL in the financial markets depends on its reliability, predictability and of course profitability. How can FinRL be provided as a robust live service to traders (albeit in pilot mode), whilst allowing development to proceed in parallel? How to monitor and classify events as incidents? How to do root cause analysis? How to manage resultant change of FinRL, automated testing, and release of that change? How to roll back any changes that are not working? How to manage capacity? How to manage the financial performance of FinRL-as-a-Service? How to manage security of the service? What are the KPIs that would be involved in creating a live service? How to manage requests from users into development?

6. How to enable the trader to measure, display and evaluate, and optimise performance of FinRL? Presume all measurements need to be against a baseline? All in terms the trader, not just data scientist, can understand.
A trader's dashboard and reporting might include:

a. Gross Profit — the sum of all profitable trades in terms of money;
b. Gross Loss — the sum of all losing trades in terms of money;
c. Total Net profit — the financial result of all trades;
d. Profit Factor — the ratio of gross profit and gross loss in percents;
e. Expected Payoff — average return of one deal.
f. Balance Drawdown Absolute — difference between the initial deposit and the minimal level below initial deposit throughout the whole history of the account. AbsoluteDrawDown = InitialDeposit - MinimalBalance
g. Balance Drawdown Maximal — difference in deposit currency between the highest local balance value and the next lowest account balance value. MaximumDrawDown = Max[Local High - Next Local Low]
h. Balance Drawdown Relative — difference in percentage terms between the highest local balance value and the next lowest account balance value. RelativeDrawdown = Max[(Local High - Next Local Low)/Local High * 100)]
i. Total trades — the total amount of executed trades (the trades that resulted in a profit or loss);
j. Short Trades (won %) — number of trades that resulted in profit obtained from selling a financial instrument, and percentage of profitable short trades;
k. Long Trades (won %) — number of trades that resulted in profit obtained from purchasing a financial instrument, and percentage of profitable long trades;
l. Profit Trades (% of total) — the amount of profitable trades and their percentage in the total trades;

m. Loss trades (% of total) — the amount of losing trades and their percentage in the total trades;
n. Largest profit trade — the largest profit of all profitable trades;
o. Largest loss trade — the largest loss of all loss-making trades;
p. Average profit trade — the average profit value per a trade (the total of profits divided by the number of winning trades);
q. Average loss trade — the average loss value per a trade (the total of losses divided by the number of losing trades);
r. Maximum consecutive wins ($) — the longest series of winning trades and their total profit;
s. Maximum consecutive losses ($) — the longest series of losing trades and their total loss;
t. Maximal consecutive profit (count) — the maximum profit of a series of profitable trades and the amount of profitable trades in this series;
u. Maximal consecutive loss (count) — the maximum loss of a series of losing trades and the amount of losing trades in this series;
v. Average consecutive wins — the average number of winning trades in profitable series;
w. Average consecutive losses — the average number of losing trades in losing series.

7. Cloud solution for FinRL

To improve the performance for large-scale markets, we will employ GPU-based massive parallel simulation as Isaac Gym. Moreover, it will be interesting to explore the deep evolutionary RL framework to simulate the markets. For more information, please visit FinRL_Podracer.

ElegantRL

ElegantRL is lightweight, efficient and stable, for researchers and practitioners.

- Lightweight: The core codes <1,000 lines (check elegantrl/tutorial), using PyTorch (train), OpenAI Gym (env), NumPy, Matplotlib (plot).
- Efficient: performance is comparable with Ray RLlib.
- Stable: as stable as Stable Baseline 3.

Currently, model-free deep reinforcement learning (DRL) algorithms:
DDPG, TD3, SAC, A2C, PPO, PPO(GAE) for continuous actions
DQN, DoubleDQN, D3QN for discrete action

1. Mastering PPO Algorithms

PPO algorithms are widely used deep RL algorithms nowadays and are chosen as baselines by many research institutes and scholars. In this section, we would like you to get hands-on experience in PPO algorithms. We prepared the documentation Mastering PPO Algorithms for you to understand the concept of PPO and how we implement PPO in our ElegantRL framework. Your job in this section will be to try to fully understand the PPO concept and reproduce PPO algorithms.

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Alg. 1: The PPO-Clip algorithm.

2. RL for MILP with Applications in Finance

Besides existing RL methods used in the financial area, there are many other functional methods which can be applied in the financial market as well. Here we take one example:
Reinforcement learning for integer programming: Learning to cut. We believe that such a method has the potential to achieve better performance if applied to the financial market as well.

In this section, the project can be divided into two phases: 1. Reproduce RL for MILP. 2. Apply RL for MILP into the ElegantRL framework.

You are also welcome to implement other works that you may think it's a good fit for financial tasks. You will need to indicate the reason you select it as well.

3. High-Performance RL for Finance

In this section, we would like you to try different datasets and illustrate the result.

NASDAQ-100 index constituents: Its 100 constituents are characterized by high technology, high growth and non-financial. Training in such an environment gets agents to capture the financial trends of technology.

Dow Jones Industrial Average (DJIA) index constituents: DJIA is the most cited market indicator for overall market performance. It is made up of 30 constituents that best represent their industries respectively.

Standard & Poor's 500 (S&P 500) index constituents: The S&P 500 index consists of 500 largest U.S. publicly traded companies.

HSI Index constituents: Hang Seng Index (HSI) is the most widely quoted indicator of the performance of the Hong Kong stock market. HSI constituent securities are grouped into Finance,Utilities, Properties and Commerce & Industry.

SSE 180 Index constituents: SSE 180 Index includes constituents with best representation of A shares listed at Shanghai Stock Exchange (SSE) with considerable size and liquidity. Listed companies are classified into 10 industries.

CSI 300 Index constituents: CSI 300 Index consists of the 300 largest and most liquid A-share stocks listed on Shenzhen Stock Exchange or on SSE; the performance of constituents very much reflect the overall performance of China A-share market.

A recent paper for reference:
Li, Zechu, Xiao-Yang Liu, Jiahao Zheng, Zhaoran Wang, Anwar Walid, and Jian Guo. "FinRL-Podracer: High performance and scalable deep reinforcement learning for quantitative finance." ACM International Conference on AI in Finance (ICAIF), 2021.

4. Including Human Knowledge

Combining human knowledge and RL might be a valuable approach to improve the algorithms: Zhang, Peng, u. a. *KoGuN: Accelerating Deep Reinforcement Learning via Integrating Human Suboptimal Knowledge*. 2020. arXiv:2002.07418 / Annabestani, Mohsen, u. a. *A new soft computing method for integration of expert's knowledge in reinforcement learning problems*. 2021. arXiv:2106.07088

5. Hyperparameter Tuning of RL Algorithms

What are hyperparameters?

A machine learning or deep learning model has parameters and hyperparameters to aid the learning process. Parameters are the entities that build the model and hyperparameters are the entities that control the learning process. Hyperparameters also play a role in data augmentation. Like in Convolutional Neural networks, the rotation angle can be treated as a hyperparameter to optimize.

In linear regression, the weight (W) and biases (B) are parameters of the model as they build the model Y = WX + B. Hyperparameters like learning rate, batch size etc. aid in learning. In neural networks, each neuron has a weight and bias which are its parameters. So in vanilla models, the hyperparameters are

generally hard-coded and stationary throughout the training process and parameters by gradient descent get updated in each iteration. But hyperparameters can also be part of the learning process. Like in SAC (Soft-Actor Critic method Version-3), the entropy factor α is learnt from the data by specifying an objective function to tune it. This is called end-to-end training, as no external efforts are necessary, everything solely depends on the data after model instantiation.

Hyperparameters are the variables that control the learning process. In deep reinforcement learning (DRL), the common hyperparameters are learning rate, batch size, number network layers, exploration factor, clipping rate, etc. These hyperparameters are initialized at the beginning of the training process and can be dynamically controlled using a scheduler or else remain constant.

FinRL currently offers three DRL agents, RLlib from Ray, Stable Baselines3 and in-house package ElegantRL. We have separate Hyperparameter Optimization (HPO) pipelines for all three agents. All HPO pipelines consists of the following steps:

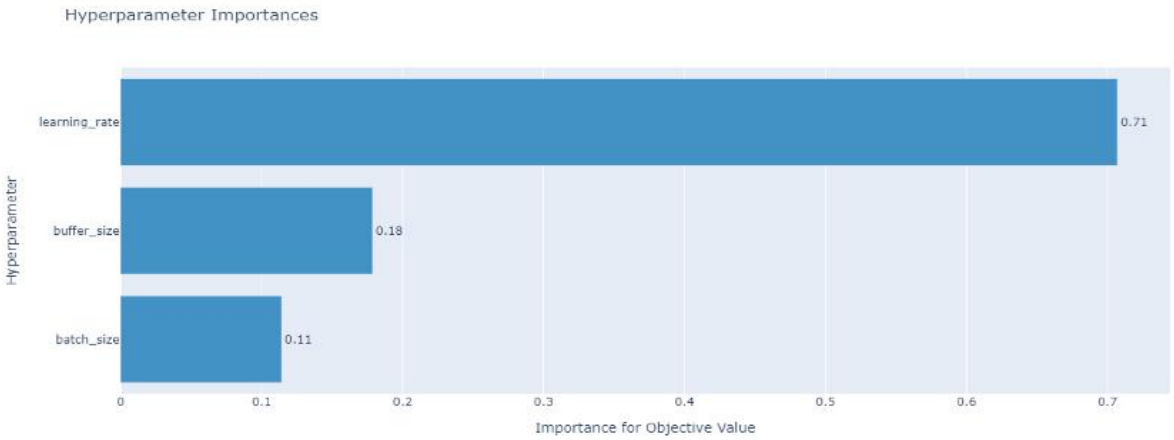
1. Pick training, validation and testing periods
2. Pick hyperparameters that you want to tune for your algorithm and define their domain space for example, loguniform, categorical, uniform

```
def sample_ddpg_params(trial:optuna.Trial):
 # Size of the replay buffer
 buffer_size = trial.suggest_categorical("buffer_size", [int(1e4), int(1e5), int(1e6)])
 learning_rate = trial.suggest_loguniform("learning_rate", 1e-5, 1)
 batch_size = trial.suggest_categorical("batch_size", [32, 64, 128, 256, 512])

 return {"buffer_size": buffer_size,
         "learning_rate":learning_rate,
         "batch_size":batch_size}
```

3. Define a metric for Bayesian optimization methods. In our testing we picked Sharpe ratio in the validation period to tune our hyperparameters
4. Define a searching technique. Some basic ones are Grid Search and Random. Other advanced methods are Tree-Parzen Estimator, Bayesian optimization, Population-Based methods. You can find more information about them here on Ray tune and Optuna.
5. Define a scheduler or pruner. These discard unpromising trials based on the chosen metric
6. Tune the hyperparameters and retrieve the best configuration

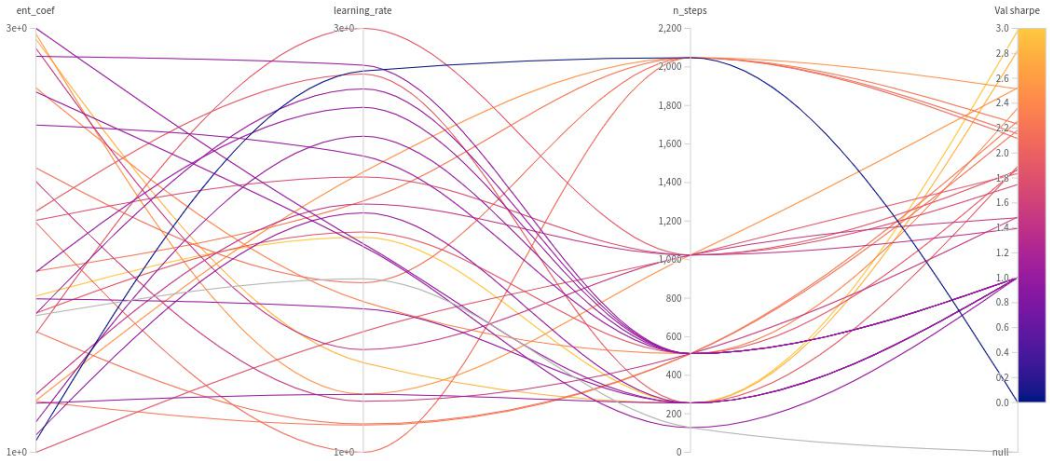
Tutorial and Guides:

- RLlib:
    - Ray tune
        - Blog post
        - Notebook
- Stable Baselines3
    - Optuna
        - Blog post
        - Notebook



Hyperparameter Feature Importance in Optuna

- Hyperparameter Sweep (Weights & Biases)
  - Blog post
  - Notebook
  - W&B report



Hyperparameter Sweep in Weights and Biases

- ElegantRL: not yet developed

Further Improvements

- Choosing an optimal metric apart from validation Sharpe ratio for sampler and scheduler
- Data augmentation in the training phase to account for changing distribution in testing phase and building robust models
- Population-Based methods for algorithms

Conclusion

DRL algorithms are sensitive to hyperparameters and seed values too. So proper tuning of hyperparameters can help us to squeeze out as much performance as possible from these algorithms. It comes with extra computation overhead, yet it is necessary to build robust trading agents.

End2end Proof-of-Concept (PoC) of DataOps and MLOps

Next, we present a proposal for a Proof-of-Concept of a production platform for the AI4Finance community. The target platform will support continuous data ingestion (batch/streaming and structure/unstructured), data transformation, feature creation, integration (CI), continuous delivery (CD), and continuous learning (CT) for machine learning (ML) systems in the AI4Finance ecosystem.

We will described the proposal in three aligned levels or architectures: conceptual (what are we proposing as a concept and why do we need it?), logical (process - how will it work?), physical (what technologies). Finally, we also describe the implementation plan.

Conceptual Architecture - Data and ML Operations

Machine learning has traditionally been approached from the perspective of individual scientific experiments performed by data scientists in isolation. However, as machine learning models become part of critical real-world solutions, we will need to change our perspective, not to recycle scientific principles, but to make them more accessible, reproducible, and collaborative. to recycle scientific principles, but to make them more accessible, reproducible, and collaborative. to recycle scientific principles, but to make them more accessible, reproducible, and collaborative.

In most real-world applications, the underlying data is constantly changing and therefore models need to be retrained or rebuilt to cope with feature drift. Business and regulatory business requirements and regulations can change rapidly, requiring a release cycle. This is where MLOps comes in to combine operational knowledge with machine learning and data science insights.

The aim of MLOps is to reduce technical friction so that the model moves from idea to production in the shortest possible time to market with the lowest possible risk.

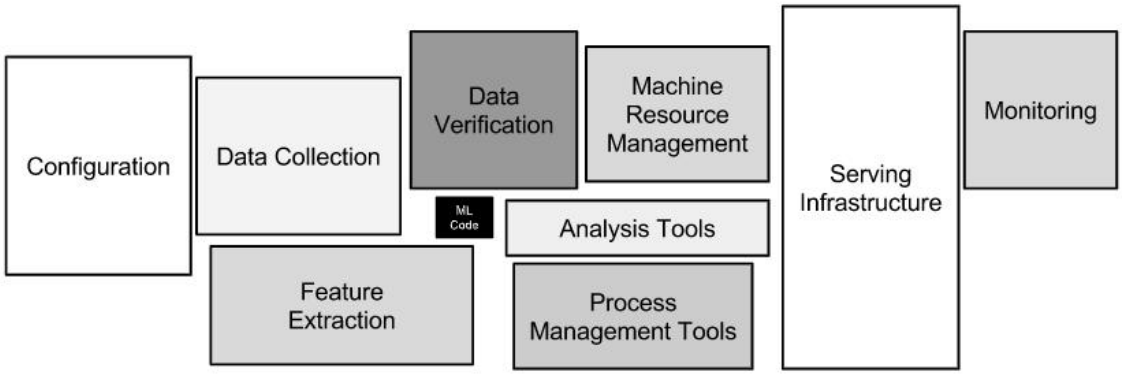


Figure X: Only a small fraction of real-world ML systems are composed of ML code, as the small black

box in the center shows. The surrounding infrastructure required is extensive and complex (Sculley et al. #).

Important notes about MLOps:

- The pipeline is the product, not only the model. Do not deploy the model, deploy the entire pipeline.
- To build the pipeline, split the system down into small and well-defined components.
- Model metrics will eventually degrade as the world changes, the pipeline must be ready for these changes.

Non-functional Requirements

Scalabilty? Yes.
Security? Yes, at least reserve APIs; keep it in mind when doing design.
Others? Cloud-native, MLOps, GPU Optimization

Logical Architecture

Pipeline

The proposed approach allows data scientists and ML engineers to quickly explore new solutions around feature engineering, model configuration and architecture, and hyperparameter selection, following CI/CD methodologies (Google). This pipeline contains different elements: source control, test and builds services, deployment services, model registry, feature store, ML metadata store, ML pipeline orchestrator.
Need to also how DataOps pipeline
Figure X: CI/CD and automated ML pipeline (Google).

Characteristics

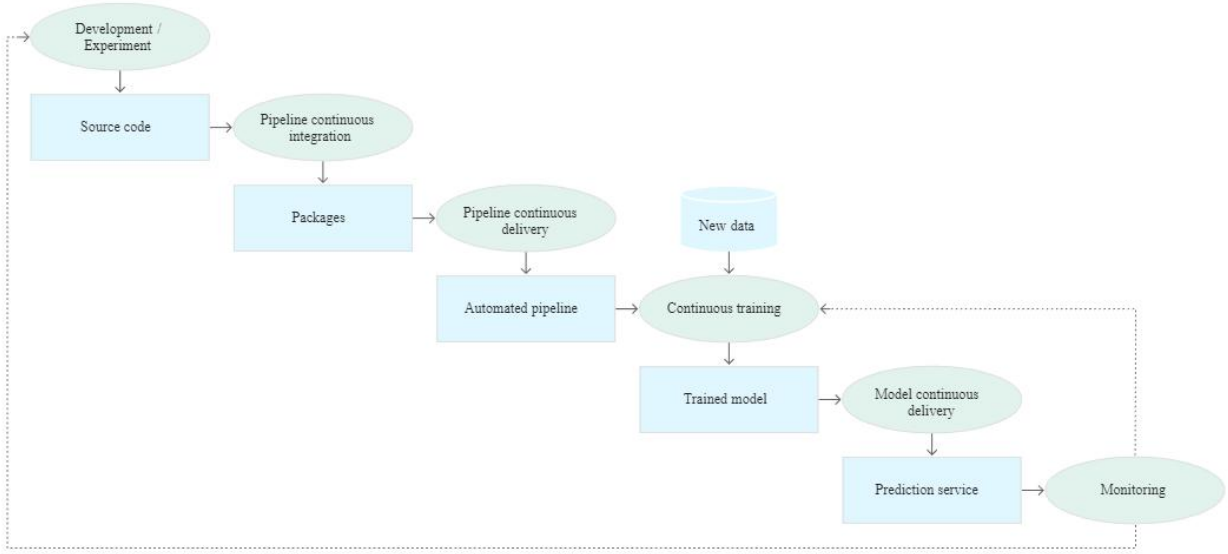


Figure X. Stages of the CI/CD automated ML pipeline (Google).

The process consists of the following stages

1. Development and experimentation: New ML algorithms and new modeling in which the experimentation steps are orchestrated are tested iteratively. The result of this stage is the source code of the ML pipeline steps which is then pushed to a source repository.
2. Continuous pipeline integration: Source code is built and various tests are run. The results of this stage are the pipeline components (packages, executables, and artefacts) that will be deployed at a later stage.
3. Continuous pipeline delivery: The artifacts produced by the CI stage are deployed to the target environment. The result of this stage is a deployed pipeline with the new model implementation.
4. Automatic activation: The pipeline is automatically executed in production based on a schedule or in response to a trigger. The result of this stage is a trained model that is sent to the model registry.
5. Continuous model delivery: The trained model is served as a prediction service for forecasts. The output of this stage is a prediction service of the deployed model.
6. Monitoring: Statistics are collected on model performance based on real data. The output of this stage is a trigger to run the pipeline or to run a new cycle of experiments.
7. The data analysis stage is still a manual process for data scientists before the pipeline starts a new iteration of the experiment. The model analysis stage is also a manual process.

Challenges

Please *check this paper: https://arxiv.org/pdf/2011.09926.pdf*

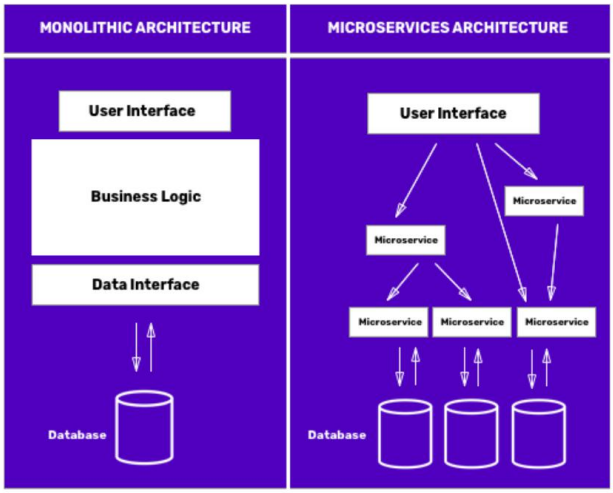
Deployment Stage	Deployment Step	Considerations, Issues and Concerns
Data management	Data collection	Data discovery
	Data preprocessing	Data dispersion Data cleaning
	Data augmentation	Labeling of large volumes of data Access to experts Lack of high-variance data
	Data analysis	Data profiling
Model learning	Model selection	Model complexity Resource-constrained environments Interpretability of the model
	Training	Computational cost Environmental impact
	Hyper-parameter selection	Resource-heavy techniques Hardware-aware optimization
Model verification	Requirement encoding	Performance metrics Business driven metrics
	Formal verification	Regulatory frameworks
	Test-based verification	Simulation-based testing
Model deployment	Integration	Operational support Reuse of code and models Software engineering anti-patterns Mixed team dynamics
	Monitoring	Feedback loops Outlier detection Custom design tooling
	Updating	Concept drift Continuous delivery
Cross-cutting aspects	Ethics	Country-level regulations Focus on technical solution only Aggravation of biases Authorship Decision making
	End users' trust	Involvement of end users User experience Explainability score
	Security	Data poisoning Model stealing Model inversion

Figure X: Challenges in MLops for different stages. Source: (Paleyes et al.)

Physical Architecture

Microservices and Machine Learning Models

As opposed to developing most or all of the application code in one place (monolith approach). Microservices-style development packages each application component into an individual piece, usually with a RESTful API endpoint for access. A production application developed as Microservices has communicating components, all developed and maintained separately. (Gage and Science #).



Tooling

To implement the pipeline, we will use a variety of tools that work independently of each other, and all of them will be implemented based on microservices, and each one will have a Docker image that will be realized and will only need to know what its inputs and outputs are, so that we can scale both horizontally and vertically.

Data Extraction

- Custom data loaders in Python
- ExampleGen (Tensorflow Extended). It consumes external files/services to generate Examples that will be read by other TFX components. It also provides consistent and configurable partition and shuffles the dataset for ML best practice.
- Custom Module in PyTorch (to explore yet)

Data Validation

- Custom modules in Python

- StatisticsGen (Tensorflow Extended). Generates feature statistics over both training and serving data, which can be used by other pipeline components. StatisticsGen uses Beam to scale to large datasets.
- SchemaGen (Tensorflow Extended). A SchemaGen pipeline component will automatically generate a schema by inferring types, categories, and ranges from the training data.
- ExampleValidator (Tensorflow Extended). The ExampleValidator pipeline component identifies anomalies in training and serving data. It can detect different classes of anomalies in the data.
- Custom Module in PyTorch (to explore yet)

Data Preparation

- Custom modules in Python.
- Transform (Tensorflow Extended). Run on top of ApacheBeam/Spark. It performs feature engineering on tf. Examples emitted from an ExampleGen component, using a data schema created by a SchemaGen component, and emits both a SavedModel as well as statistics on both pre-transform and post-transform data
- Custom Module in PyTorch (to explore yet)

Model Training

- Trainer (Tensorflow Extended)
- Tuner (Tensorflow Extended). It is in charge of the hyperparameters tuning for the model.
- PyTorch
- RayTune (Tensorflow Extended)
- Katib (Kubeflow)

Model Evaluation

- Custom modules in Python.
- Evaluator (Tensorflow Extended). It performs deep analysis on the training results for your models, to help you understand how your model performs on subsets of your data. The Evaluator also helps you validate your exported models, ensuring that they are "good enough" to be pushed to production.
- PyTorch (to explore yet)

Model Validation

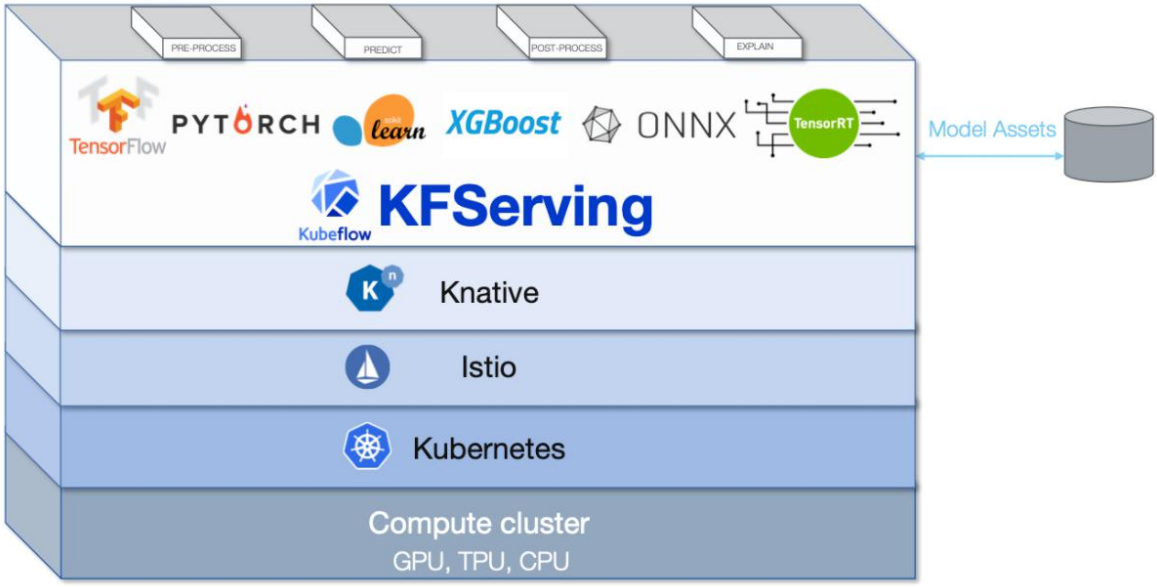
- InfraValidator (Tensorflow Extended). It is used as an early warning layer before pushing a model into production. The name "infra" validator came from the fact that it is validating the model in the actual model serving "infrastructure".
- Custom python modules

Model Deploy

- Pusher (Tensorflow Extended). The Pusher component is used to push a validated model to a deployment target during model training or re-training. Before the deployment, Pusher relies on one or more blessings from other validation components to decide whether to push the model or not.
    - Evaluator blesses the model if the newly trained model is "good enough" to be pushed to production.
    - (Optional but recommended) InfraValidator blesses the model if the model is mechanically servable in a production environment.

  A Pusher component consumes a trained model in SavedModel format, and produces the same SavedModel, along with versioning metadata.

- Tensorflow Serving - Tensorflow (REST and gRPC)
- TorchServe - PyTorch
- Seldon (to explore yet)
- KFserving - Kubeflow (to explore yet)



Monitoring (Time series Database + Dashboards)

- Prometheus + Grafana
- InfluxDB + Grafana

Pipeline Orchestration: **Kubeflow**

Kubeflow is an open source Kubernetes-native platform for developing, orchestrating, deploying, and running scalable and portable machine learning (ML) workloads. It is a cloud native platform based on Google's internal ML pipelines. The project is dedicated to making deployments of ML workflows on Kubernetes simple, portable, and scalable.

Kubernetes is an orchestration system for containers that is meant to coordinate clusters of nodes at scale, in production, in an efficient manner. Kubernetes works around the idea of Pods which are scheduling units (each pod containing one or more containers) in the Kubernetes ecosystem. These pods are distributed across hosts in a cluster to provide high availability. Kubernetes itself is not a complete solution and is intended to integrate with other tools such as Docker. A container image is a lightweight, standalone, executable package of a piece of software that includes everything needed to run it.

The goal of Kubeflow is to simplify the deployment of machine learning workflows to Kubernetes. The issue with using the Kubernetes API directly is that it is too low-level for most data scientists. A data scientist already has to know a number of techniques and technologies without the necessity of adding the complexities of the Kubernetes API to the list. The issues Kubeflow solves beyond just the core Kubernetes API are:

- Faster and more consistent deployment
- Better control over ports and component access for tighter security

- Protection against over-provisioning resources, saving costs
- Protection against tasks not being deallocated once complete, saving costs
- Workflow orchestration and metadata collection
- Centralized monitoring and logging
- Infrastructure to move models to production, securely and at scale

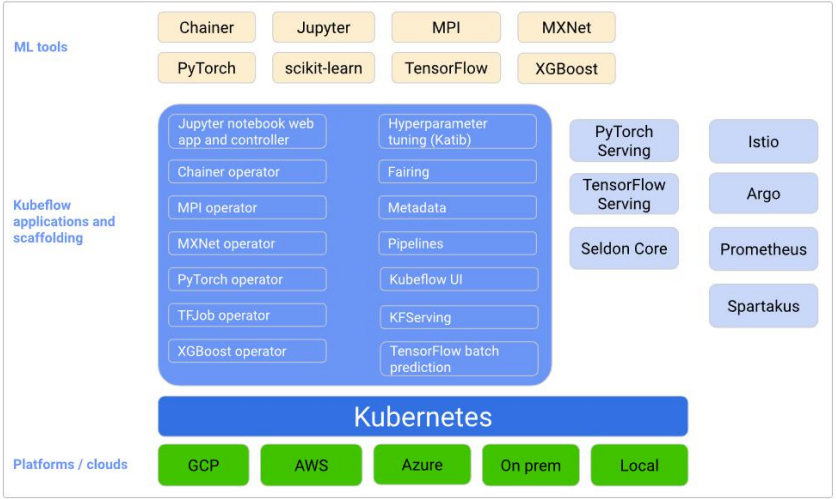


Figure X. Conceptual Overview of Kubeflow (https://www.kubeflow.org/docs/started/architecture/)

These components work together to provide a scalable and secure system for executing machine learning jobs (notebook-based jobs as well as non-notebook jobs). Given the rise of Kubernetes as an enterprise platform management system, it makes a lot of sense to have a way of managing our makes a lot of sense to have a way to manage our machine learning workloads in a similar way.

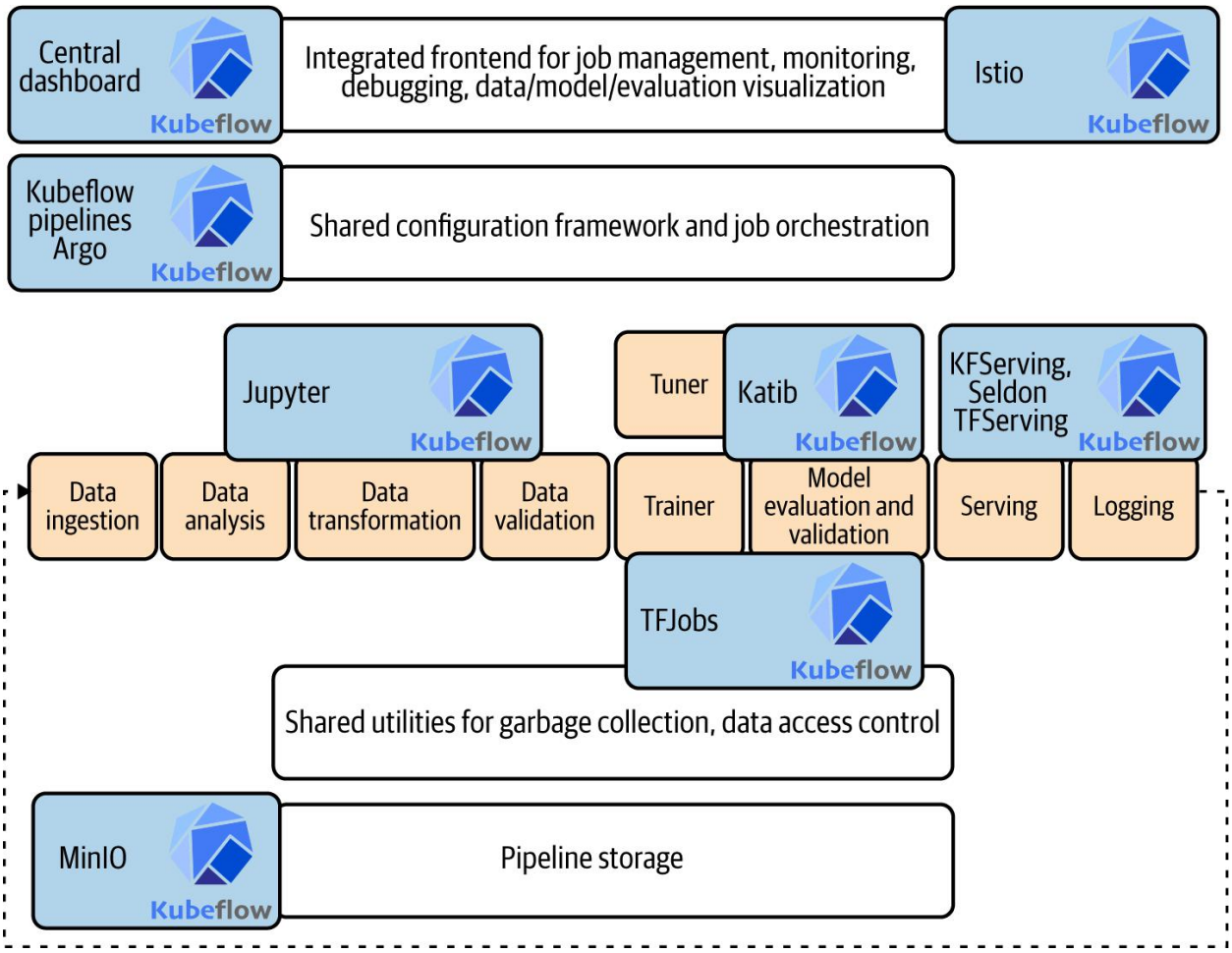
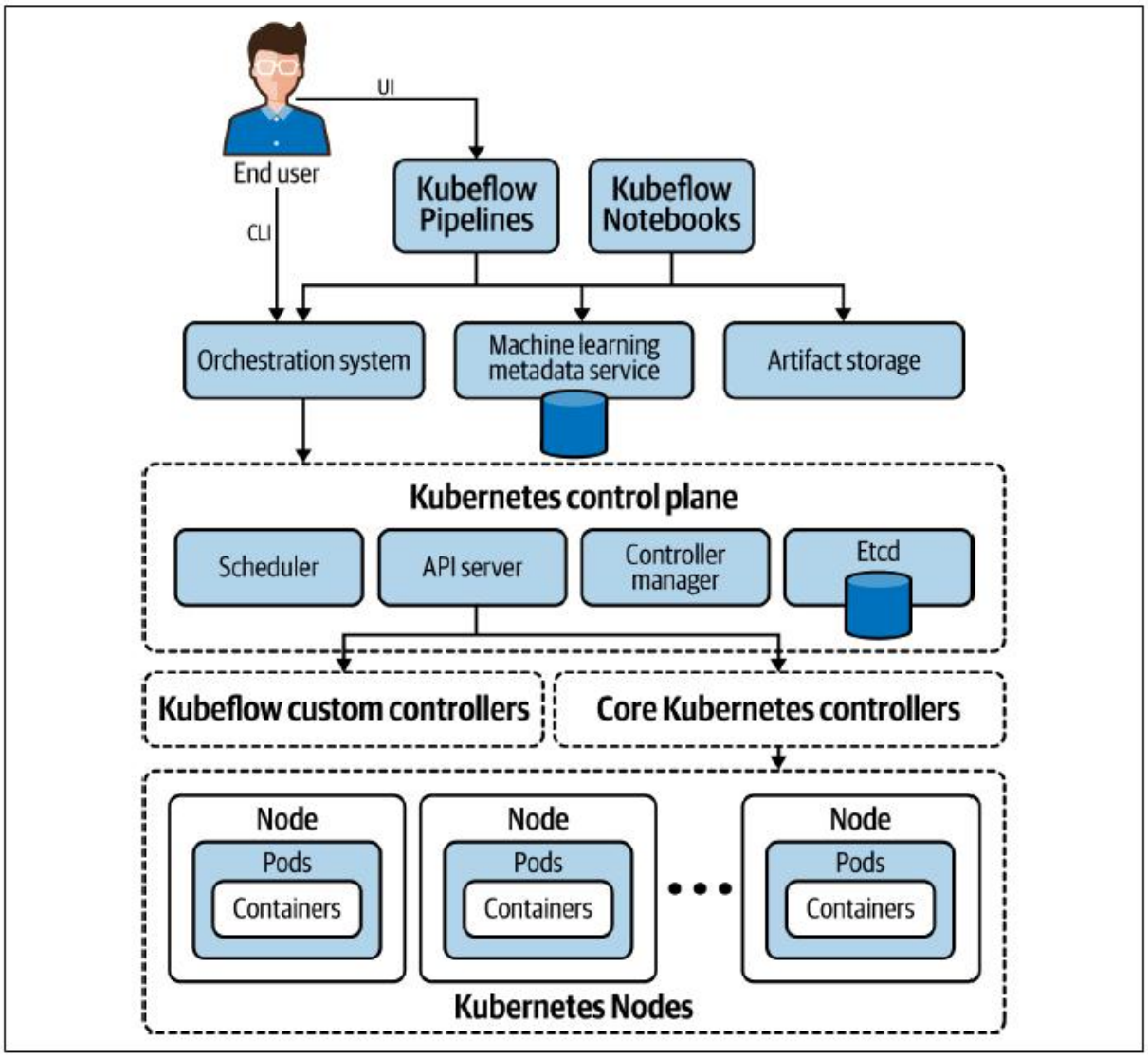


Figure X. Kubeflow components in the ML workflow

(https://www.oreilly.com/library/view/kubeflow-for-machine/9781492050117/ch03.html)

The high-level architecture of Kubeflow is presented in the next image. Kubeflow is built from multiple separate components in a service mesh to provide the full machine learning platform. Istio supports operations of the Kubeflow distributed microservice architecture by providing features such as secure communications, discovery, load balancing, failure recovery, metrics, and monitoring.


UI
End user
CLI
Kubeflow Pipelines
Kubeflow Notebooks
Orchestration system
Machine learning metadata service
Artifact storage
Kubernetes control plane
Scheduler
API server
Controller manager
Etcd
Kubeflow custom controllers
Core Kubernetes controllers
Node
Pods
Containers
Node
Pods
Containers
Node
Pods
Containers
Kubernetes Nodes

References

Google. “MLOps: Continuous delivery and automation pipelines in machine learning.” Google Cloud, 7 January 2020, https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning. Accessed 30 January 2022.

Paleyes, Andreis, et al. “Challenges in Deploying Machine Learning: a Survey of Case Studies.” arXiv, 18 November 2020, https://arxiv.org/abs/2011.09926. Accessed 30 January 2022.

Sculley, D., et al. “Hidden technical debt in machine learning systems.” Advances in Neural Information Processing Systems, 2015, pp. 2503-2511. NeurIPS.

Appendix I. Common Pitfalls to Avoid in Financial Machine learning

Look-ahead bias (future information)

Look-ahead bias is one of the big problems in financial deep learning. It occurs by the use of information or data in training or backtesting that would not have been known or available during the period being analyzed. It is always important to double-check whether future data will leak into the decision. This often happens when not being careful creating the features. A classic example is using the daily low or high as a feature in an intraday trading strategy. In reality, the daily low is not known until the end of the day.

Another example frequently introducing look-ahead is the VWAP technical indicator which is based on the volume of the day, this again changes during the day till it reaches its final value at the end of the day. If one is not careful implementing it, future data is being used. Another situation that might cause a look-ahead can be the selection of certain financial instruments based on your "now-knowledge". For example, selection of the current best performing instruments for your portfolio to train or backtest, can already introduce look-ahead and cause the model being biased to (long-term) uptrends. The best antidote is being careful and using forward-testing / paper-trading with live data. When working with missing-data, always forward fill.

Data-snooping bias / Overfitting

This is really hard to avoid, especially with DL. We won't go into detail here, as overfitting should be basic knowledge in the DL world. The best antidote is using a big enough sample and out-of-sample testing. Working with DL, you should already be familiar with the Training-Validation-Testing split. In the case of financial instruments, you also should additionally be aware of market regimes though. For example, cryptocurrencies have limited historic data available due to being relatively new instruments. The first bitcoin transaction was in 2009 and many cryptocurrencies were born even later. Having such little data increases the danger of overfitting. Certain time periods are affected by different market regimes like bull-market, bear-market and ranging / sideways-market. If you create your sample, you need to be aware of that. Otherwise, your model might be overfitted to those certain market conditions and fail in the other. There also can be overfitting to certain instruments, exchanges and timeframes. In theory, the perfect model would work similarly well when switching the traded instruments, used exchange or trading timeframes. In reality, it's often observed that this is not the case. This doesn't necessarily mean the model is bad. One might theorize that the model exploits certain market situations that only occur on those instruments, exchanges or timeframes. With Algorithmic trading accounting for around 60-73% of the overall United States equity trading (see https://www.mordorintelligence.com/industry-reports/algorithmic-trading-market) our model just might exploit another automatic trading systems trading decisions, with this other automatic trading system only being active on certain instruments. A model heavily depending on volume for its decision only might

work on the exchange it was trained on, as volume / liquidity / (automated) market-makers behind the scene differ from exchange to exchange. You see, overfitting is a hard topic to evaluate.

Transaction Costs & Slippage

No backtest performance is realistic without transaction costs and slippage. Depending on the financial instrument and trading exchange, those can be fixed transaction costs and / or percentages. Slippage can vary a lot. It differs from exchange to exchange and the instrument traded. Often, instruments with very low volume have bigger slippage.

Understanding performance measurement

It's important you make yourself familiar with the performance metrics. One common mistake is just looking at profit and not comparing the metrics to the buy-and-hold / equally-weighted benchmark. A successful trading system should always beat the benchmark. In what way depends on the investor's preferences. For example, not beating the buy-and-hold return, but having way less drawdown / risk can be totally fine and wanted by investors, while others prefer big gains accepting more risk. This part also is essential for the reward function of the model. Reading the above, you might already realize that just rewarding profit percentage might not be enough. The most popular ratio is the Sharpe ratio. While it already gives you substantially more information as it incorporates risk, it also has its downsides. Make yourself familiar with all the different metrics and what they can give you for information and what information about the performance they don't provide. A good starting point is understanding the difference between Sharpe, Sortino and Calmar Ratio. Quantstats (https://github.com/ranaroussi/quantstats) and Pyfolio (https://github.com/quantopian/pyfolio) are great tools for performance and risk analysis, providing you many metrics and graphs. Consider the right Annualization Factor for your metrics. Cryptocurrency is traded 24/7, so you need to use 365, while the stock market is closed on certain days.

This is a great paper extending and enlarging upon the above-mentioned pitfalls:

López de Prado, Marcos, The 10 Reasons Most Machine Learning Funds Fail (January 27, 2018). Journal of Portfolio Management, Forthcoming, Available at SSRN: https://ssrn.com/abstract=3104816 or http://dx.doi.org/10.2139/ssrn.3104816

Appendix II. FinRL Frequently Asked Questions

FAQ document

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

Outline

Section 1 Where to start?

- Read the paper that describes the FinRL library: Liu, X.Y., Yang, H., Chen, Q., Zhang, R., Yang, L., Xiao, B. and Wang, C.D., 2020. FinRL: A Deep Reinforcement Learning Library for Automated Stock Trading in Quantitative Finance. Deep RL Workshop, NeurIPS 2020. paper video
- Read the post related to the type of environment you want to work on (multi stock trading, portfolio optimization) https://github.com/AI4Finance-Foundation/FinRL , Section "News"
- Install the library following the instructions at the official Github repo: https://github.com/AI4Finance-Foundation/FinRL
- Run the Jupyter notebooks related to the type of environment you want to work on notebooks folder of the library : https://github.com/AI4Finance-Foundation/FinRL/tree/master/tutorials
- Enter on the AI4Finance slack: https://join.slack.com/t/ai4financeworkspace/shared_invite/zt-kq0c9het-FCSU6Y986OnSw6Wb5EkEYw

Section 2 What to do when you experience problems?

- If any questions arise, please follow this sequence of activities (it allows us to focus on the main issues that need to be solved, instead of repeatedly answering the same questions):
    1. Check if it is not already answered on this FAQ
    2. Check if it is not posted on the Github repo issues: https://github.com/AI4Finance-Foundation/FinRL-Library/issues
    3. Use the correct slack channel on the AI4Finance slack.

Section 3 Most frequently asked questions related to the FinRL Library

Subsection 3.1 Inputs and datasets

- Can I use FinRL for crypto? → We're developing this functionality
- Can I use FinRL for live trading? → We're developing this functionality
- Can I use FinRL for forex? → We're developing this functionality
- Can I use FinRL for futures? → not yet

- What is the best data source for free daily data → Yahoo Finance (through the yfinance library)
- What is the best data source for minute data → 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)
- Does FinRL support trading with leverage? → 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
- Can a sentiment feature be added to improve the model's performance? → yes, you can add it. Remember to check on the code that this additional feature is being fed to the model (state)
- Is there a good free source for market sentiment to use as a feature? → 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)

Subsection 3.2 Code and implementation

- Does FinRL support GPU training? → yes, it does
- The code works for daily data but gives bad results on intraday frequency → yes, because the current parameters are defined for daily data. You'll have to tune the model for intraday trading
- Are there different reward functions available? → not yet, but we're working on providing different reward functions and an easy way to code your own reward function
- Can I use a pre-trained model? → yes, but none is available at the moment. Sometimes in the literature you'll find this referred to as transfer learning
- What is the most important hyperparameter to tune on the models? → 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

- What are some libraries I could use to better tune the models? → there are several, such as: ray rllib and optuna. You'll have to implement them by yourself on the code, as this is not supported yet
- What DRL models can I use with FinRL? → all the DRL models on Stable Baselines 3. We tested the following models with success: A2C, A3C, DDPG, PPO, SAC, TD3, TRPO. You can also create your own model, using the OpenAI Gym structure
- The model is presenting strange results OR is not training → Please update to latest version (https://github.com/AI4Finance-Foundation/FinRL), 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)

Subsection 3.3 Model evaluation

- The model did not beat buy and hold (BH) with my data. Is the model or code wrong? → 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
- How does backtesting work in the library? → 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

- Which metrics should I use for evaluating the model? → 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
- Which models should I use as a baseline for comparison? → we recommend using buy and hold (BH), as it's 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

Subsection 3.4 Miscellaneous

- What is the development roadmap for the library? → this is available on our Github repo (https://github.com/AI4Finance-Foundation/FinRL)
- How can I contribute to the development? → 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)
- What are some good references before I start using the library? → please read Section 1 (Where to start?)
- What are some good RL references for people from finance? What are some good finance references for people from ML? → please read Section 4 (References for diving deep into Deep Reinforcement Learning (DRL) for finance)
- What new sota models will be incorporated on FinRL? → please check our development roadmap at our Github repo https://github.com/AI4Finance-Foundation/FinRL

Section 4 References for diving deep into Deep Reinforcement Learning (DRL)

Subsection 4.1 General resources

- OpenAI Spinning UP DRL, educational page for DRL:
  https://spinningup.openai.com/en/latest/
- Awesome-ai-in-finance
  https://github.com/georgezouq/awesome-ai-in-finance
- Curated list of practical financial machine learning tools and applications
  https://github.com/firmai/financial-machine-learning
- OpenAI Gym
- https://github.com/openai/gym
- Stable Baselines 3
  contains the implementations of all models used by FinRL
  https://github.com/DLR-RM/stable-baselines3
- Ray RLlib
  https://docs.ray.io/en/master/rllib.html
- Policy gradient algorithms
  https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html
- Fischer, T.G., 2018. Reinforcement learning in financial markets-a survey (No. 12/2018). **FAU Discussion Papers in Economics.** → a survey on the use of RL for finance
- Li, Y., 2018. Deep reinforcement learning. **arXiv preprint arXiv:1810.06339.** → an in-depth review of DRL and its main models and components
- Charpentier, A., Elie, R. and Remlinger, C., 2020. Reinforcement learning in economics and finance. **arXiv preprint arXiv:2003.10014.** → an in-depth review of uses of RL and DRL in finance
- Kolm, P.N. and Ritter, G., 2020. Modern perspectives on reinforcement learning in finance. Modern Perspectives on Reinforcement Learning in Finance (September 6, 2019). **The Journal of Machine Learning in Finance**, 1(1) → an in-depth review of uses of RL and DRL in finance
- Practical Deep Reinforcement Learning Approach for Stock Trading, **paper** and **codes**, Workshop on Challenges and Opportunities for AI in Financial Services, NeurIPS 2018.
- Hambly, B., Xu, R. and Yang, H., 2021. Recent Advances in Reinforcement Learning in Finance. **arXiv preprint arXiv:2112.04553**.

Subsection 4.2 Papers related to the implemented DRL models

Check the website: https://elegantrl.readthedocs.io/en/latest/index.html

- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D. and Riedmiller, M., 2013. Playing atari with deep reinforcement learning. ICLR 2013→ the first paper that proposed (with success) the combination of deep neural neworks and RL.
- Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G. and Petersen, S., 2015. Human-level control through deep reinforcement learning. **Nature,** 518(7540), pp.529–533 → an excellent review paper of important concepts on DRL
- Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D. and Wierstra, D., 2015. Continuous control with deep reinforcement learning. ICLR 2015 → paper that proposed the DDPG algorithm.
- Fujimoto, S., Hoof, H. and Meger, D., 2018, July. Addressing function approximation error in actor-critic methods. ICML **(**pp. 1587–1596). PMLR → paper that proposed the TD3 model
- Schulman, J., Wolski, F., Dhariwal, P., Radford, A. and Klimov, O., 2017. Proximal policy optimization algorithms. **arXiv preprint arXiv:1707.06347** → paper that proposed the PPO model
- Mnih, V., Badia, A.P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D. and Kavukcuoglu, K., 2016, June. Asynchronous methods for deep reinforcement learning. In **International conference on machine learning** (pp. 1928–1937). PMLR → paper that proposed the A3C model
- https://openai.com/blog/baselines-acktr-a2c/ --> description of the implementation of the A2C model
- Schulman, J., Levine, S., Abbeel, P., Jordan, M. and Moritz, P., 2015, June. Trust region policy optimization. **ICML** (pp. 1889–1897). PMLR → description of the implementation of the TRPO model.

Challenges of DataOps and MLOps

- Paleyes, A., Urma, R.G. and Lawrence, N.D., 2020. Challenges in deploying machine learning: a survey of case studies. arXiv preprint arXiv:2011.09926.