Financial advisor Bot

Final project

Table of contents

1 Intro

I chose to specialise in **Artificial Intelligence and Machine Learning** not only because it is one of the most impactful and rapidly evolving areas in technology today, but also because I have always been fascinated by the idea that intelligent systems can learn through interaction, adapt to change, and improve through experience.
A clear example of this concept is found in one of the most remarkable technological achievements of recent years: **self-driving cars.** It is extraordinary how principles of learning and adaptation have been used to master such a complex challenge.

Among all the AI techniques I studied, **Reinforcement Learning (RL)** and **Genetic Algorithms (GA)** stood out to me as the most engaging. Both share an underlying concept of continual adaptation, learning from trial and error, which, in my view, mirrors not only intelligent behaviour but also the dynamics of real-world problem-solving.

While all machine learning concepts are valuable, I find the underlying mechanisms such as **backpropagation**, **loss functions**, and **reward-driven optimisation** especially interesting: in my opinion they represent a more dynamic and efficient way to enable continuous improvement compared to traditional classification tasks that rely heavily on static and huge datasets (*is this a picture of a dog or of a cat?*).
As computational resources grow and AI systems become more autonomous, I believe the industry is naturally progressing toward Agent and **Agentic AI,** intelligent agents capable of acting, reasoning, and improving autonomously in real-time environments (like the toy robot Scientist topic we have been studying).

During my study of Reinforcement Learning, I also explored real-world applications beyond coursework and discovered that **financial decision-making and trading** are among the most common and challenging RL use cases.
The idea that an agent could learn optimal trading or portfolio strategies through repeated interaction with market data intrigued me deeply.
When we began experimenting with **OpenAI Gym environments,** training agents to play *Atari Breakout* or to climb simulated mountains, I saw first-hand how policies evolve over time through experience and adaptation.

Later, when studying **Genetic Algorithms**, I realised that similar principles of adaptation could be applied at an even broader scale, for example evolving populations of solutions toward better performance over generations. Whether the goal is improving the efficiency of an airplane wing or evolving a simulated creature to reach the top of a mountain, the same logic can even be extended to the optimisation of **neural networks**.

Out of curiosity, I decided to explore how these ideas were applied in practice and began reading a book about **Machine Learning and Algorithmic Trading.** I was pleasantly surprised to find many of the concepts studied throughout the course, particularly Reinforcement Learning and Genetic Algorithms, discussed in real-world financial contexts. This confirmed that the methods I was most passionate about have tangible and valuable applications in the industry.

The financial environment is, by nature, unpredictable and noisy (I'd say even chaotic sometimes). That challenge intrigued me: could an intelligent agent detect meaningful / hidden patterns in what appears to be randomness?

One analogy related to Artificial Intelligence that I once read, and still find particularly powerful, is:

"Machine Learning is like giving an algorithm a crumpled piece of paper and asking it to un-crumple it."

That image perfectly captures the essence of uncovering order from disorder through iterative learning, one of the main reasons I chose the **Financial Advisor Bot** template for my final project: it represents a challenging but motivating attempt to "un-crumple" the complexity of financial decision-making using adaptive, learning-based systems.

Therefore selecting the **Financial Advisor Bot** as my final project was an intuitive and logical decision. It allows me to apply the AI and ML concepts I am most passionate about (Reinforcement Learning, Genetic Algorithms) within a context that is both academically challenging and practically relevant and it also presents an exciting opportunity to integrate another of my interests: **Natural Language Processing (NLP)**. The ability to interpret complex model outcomes and communicate them in clear, human-understandable language is increasingly important, especially in financial applications where transparency and user trust are essential.
Building a system that not only *learns* but also *explains* its reasoning therefore feels like a natural and meaningful extension of my studies.

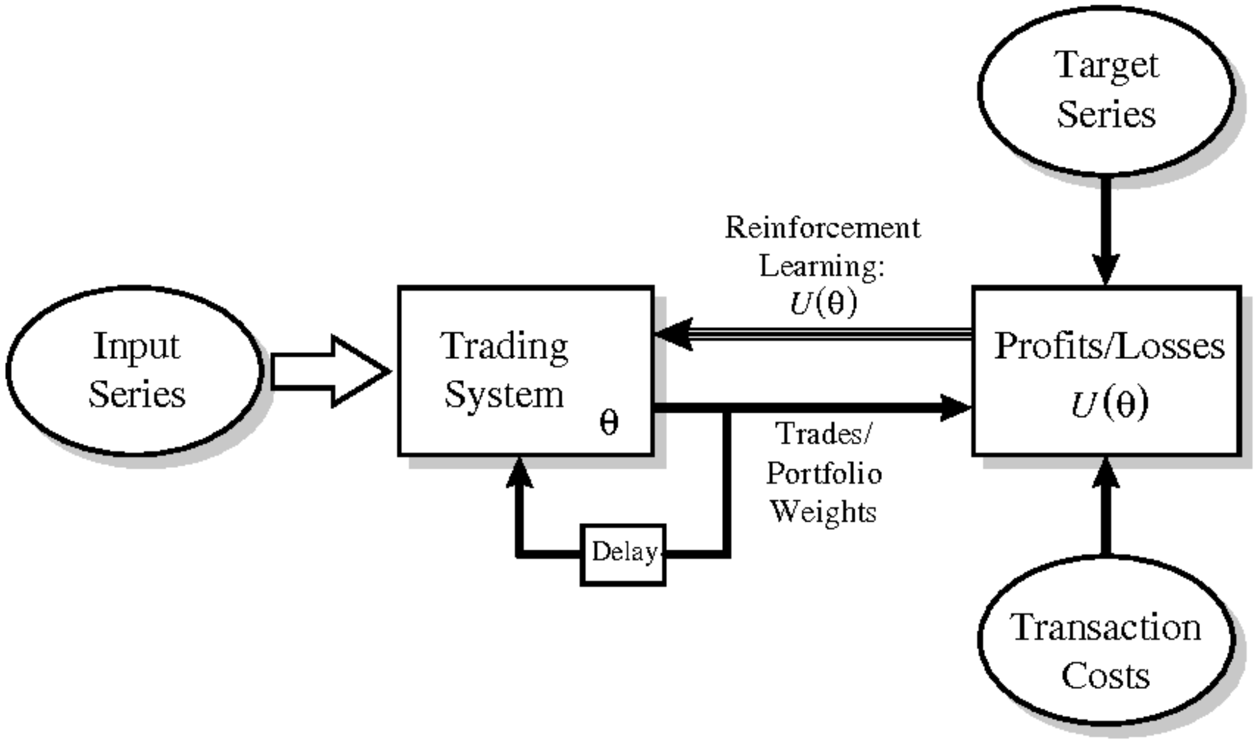
This project will enable me to explore how AI agents can learn adaptive investment strategies, optimise their performance through evolutionary processes, and communicate their insights effectively to human users.

2 Literature review

For the literature review, I studied key research applying Reinforcement Learning (RL) and Genetic Algorithms (GA) to financial trading and portfolio optimisation. These two techniques were central to my studies during the degree, and they form the theoretical foundation of my project. By analysing how prior work has modelled trading as a sequential decision-making problem, designed reward functions, addressed market volatility, and integrated evolutionary optimisation, the aim for the review is to identify both effective strategies and related strengths and weaknesses. The selected studies range from early direct-reinforcement methods to recent deep RL, hybrid GA-RL, and multi-module architectures.

2.1 Moody & Saffell (2001)

The first example I encountered during my background and literature research dates back to the early 2000s, when **Moody and Saffell (2001)** presented one of the earliest RL approaches for trading, introducing a *direct reinforcement (DR)* framework in which an agent optimises portfolio performance without relying on value-function estimation. Their method formulates trading as a stochastic control problem and updates policy parameters through gradient ascent on a utility-based objective.


Target Series
Reinforcement Learning: $U(\theta)$
Input Series
Trading System θ
Profits/Losses $U(\theta)$
Trades/ Portfolio Weights
Delay
Transaction Costs

A key contribution of their work is the explicit avoidance of the recursive Bellman equation to circumvent the “curse of dimensionality.” Instead of predicting long-term value estimates as in Q-learning, their system optimises a risk-adjusted performance measure directly (the differential Sharpe ratio). This insight reflects an essential principle in financial RL: reward functions must incorporate volatility, risk, and transaction costs to remain viable beyond theoretical settings.

The paper also highlights early recognition of domain-specific constraints in trading systems, including the need for smooth, stable policy updates and the importance of reward shaping. Although computationally modest by modern standards, Moody and Saffell established conceptual foundations for later policy-gradient and model-free approaches in finance.

2.2 Deep Reinforcement Learning and Portfolio Optimisation - Jiang, Xu and Liang (2017)

A major development came with Jiang, Xu and Liang (2017), who introduced the **Ensemble of Identical Independent Evaluators (EIIE)** architecture for multi-asset portfolio allocation. Their system uses a collection of small neural networks (one per asset) that process historical price data and jointly output portfolio-weight adjustments. This structure is modular and linearly scalable: the model can incorporate new assets simply by adding additional evaluators.

The reward function is defined as portfolio growth at each time-step, enabling faster training and a mathematically clean formulation. However, the authors acknowledge that their assumptions, such as perfect liquidity, absence of slippage, and deterministic price execution, make the reported profitability over-optimistic compared to real markets.

I find it useful to compare these approaches with the ones I studied during my degree. In this case, I observed that the **Atari DQN agent** I implemented uses **one network** to predict action values for a single agent in a single environment, while **Jiang et al.'s model** uses **multiple identical networks** operating in parallel (each focused on one asset) and a shared top layer that merges their evaluations into portfolio actions.
In simple terms, DQN learns “what move to make next,” while EIIE learns “how much to invest in each asset.”

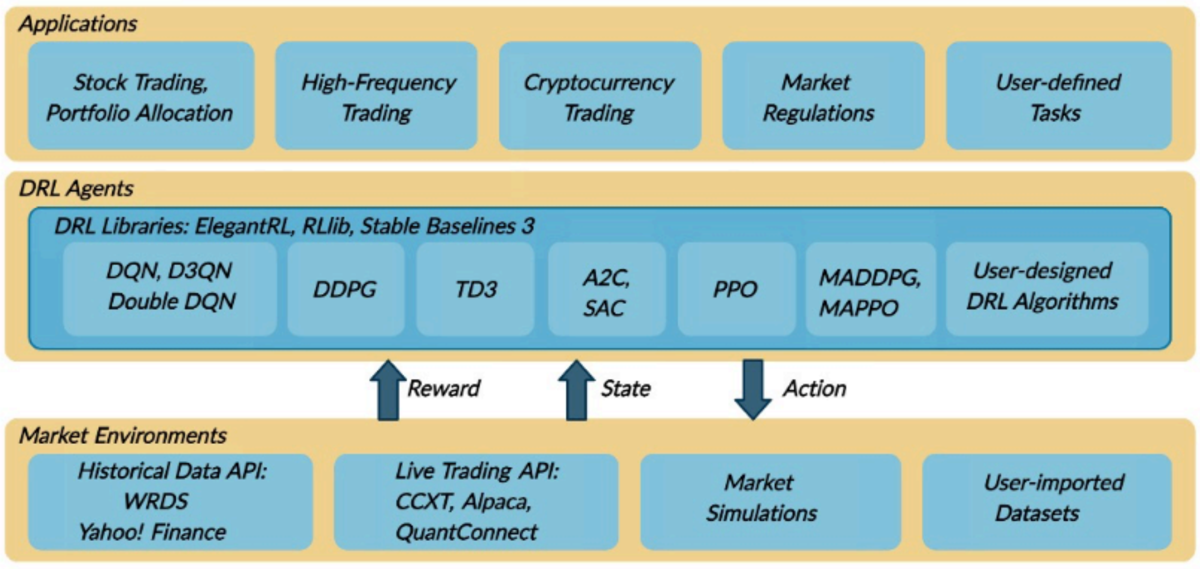
When comparing this solution not only with DQN but also with **Moody & Saffell (2001)**, I also noticed important differences in how **rewards** are defined:

- **Moody & Saffell** optimised a *risk-adjusted utility function* (the differential Sharpe ratio) , effectively teaching the agent to balance return and volatility.
- **Atari DQN** used a *delayed reward* system because you only know if the strategy works after multiple steps (e.g., finishing a level).
- **Jiang et al.**, by contrast, used an explicitly defined and immediate reward function based on portfolio growth at each step.

Their framework outperformed several traditional portfolio strategies, however Jiang et al. also acknowledged important **limitations**:

- Their system assumes no transaction delays or market impact, which is unrealistic for live trading.
- It does not handle long-term dependencies beyond a few timesteps.
- The model's success depends heavily on the stability of historical correlations, which is a major issue in volatile markets.

Reading this paper helped me appreciate how far RL has evolved since the early *direct reinforcement* methods. **Jiang et al.** extended the concept into a deep-learning framework capable of handling multiple assets simultaneously, combining scalability, modularity, and the ability to incorporate domain-specific constraints such as transaction costs.
To me, this work also reinforced one of the biggest challenges in applying deep RL to finance: **overfitting**. Financial markets are inherently **non-stationary**, meaning that yesterday's optimal policy can quickly become obsolete, an issue that any adaptive trading system must (meaning 'should be aware and try to') address or at least consider.



2.3 FinRL – A Standardised Deep Reinforcement-Learning Framework for Financial Trading

After reviewing early Reinforcement Learning approaches, I wanted to explore what current researchers actually use to build trading agents. This search led me to **FinRL (Liu et al., 2021)**, an open-source library that provides ready-to-use environments, algorithms, and evaluation tools for financial reinforcement learning (FinRL is basically what **OpenAI Gym** is for games: a research platform that standardises environments, datasets, and reward functions so that different RL models can be compared under consistent scenarios).

One of FinRL's main goals is not to invent a new trading algorithm but to **unify the workflow** of reinforcement learning in finance. It wraps data sources such as *Yahoo Finance* and *Alpaca*, preprocesses price and fundamental data into market "states" and allows users to train standard RL agents, such as DQN, under consistent settings.

The environment simulates a trading session in which an agent observes features (e.g., prices, holdings, and cash balance) and chooses actions (buy, sell, or hold). After each step, the environment returns a reward (typically the change in portfolio value).
I immediately noticed the familiarity with the **Atari DQN** structure I studied during the course: the same action loop, replay buffer, and episodic learning cycle based on state observation (rewards are no longer game scores but **portfolio returns**, often adjusted for risk using techniques, like the **Sharpe ratio).**

Overall, I found FinRL to be an **excellent teaching and research tool**. Its modular design and clear documentation made it easy for me to relate to the frameworks I used during my coursework. The library integrates multiple reinforcement-learning paradigms (e.g. value-based, policy, etc.) and directly connects RL with financial evaluation metrics.

Despite its strengths, FinRL also illustrates again the main limitations of most current RL research in finance: although FinRL can simulate transaction costs, liquidity, order execution, and slippage are often ignored.

When comparing with Moody & Saffell (2001), I think FinRL generalises the concept of reinforcement learning from a single-agent system to a **broader framework** supporting many algorithms (some value-based like DQN, others policy-gradient based) whilst Moody and Saffell trained a single recurrent model to optimise a risk-adjusted objective.
In contrast to Jiang et al. (2017), FinRL does not propose a new neural topology such as EIIE: it implements existing ones within a controlled environment and it lacks Jiang et al.'s focus on multi-asset scalability and explicit transaction-cost handling.

Finally, compared with the **Atari DQN** I studied, FinRL formalises the same reinforcement-learning principles but applies them to a far noisier and less predictable environment. The agent still seeks to maximise cumulative reward through exploration and exploitation, but here the reward is tied to *real-world financial performance* rather than a deterministic score. This makes training more unstable and heavily dependent on **hyperparameter tuning** - an area where **Genetic Algorithms (GA)** could offer valuable support by automatically evolving parameters or reward functions, which is one of the directions I intend to explore further.

2.4 Hybrid Genetic-Algorithm and Deep Q-Learning Approaches (Ding et al., 2019; Zeng et al., 2023)

Given the volatility and non-stationarity of financial markets, several researchers have explored combining RL with Genetic Algorithms to enhance adaptability. Two notable examples are Ding et al. (2019) and Zeng et al. (2023), who introduce hybrid frameworks that evolve RL parameters using evolutionary search.

Ding et al. (2019) treat each DQN agent as an individual within a GA population. After each episode, agents are evaluated on cumulative and risk-adjusted returns, and the top performers undergo crossover and mutation to produce new policies. Over generations, this evolutionary process refines hyperparameters and stabilises the learning behaviour.

Zeng et al. (2023) extend this idea by evolving hyperparameters such as learning rate, exploration ratio, and discount factor dynamically throughout training. Their method reduces manual tuning and improves convergence in noisy environments.
These hybrid designs are conceptually compelling for several reasons:

- RL learns within a generation through experience replay and policy updates.
- GA evolves across generations by exploring broader design spaces.
- The combination supports both exploitation (learning from data) and exploration (searching for new strategies).

The main drawbacks are computational cost (training populations of DQN agents is resource-intensive) and reduced explainability. Nevertheless, these studies illustrate a promising direction for building more adaptive and robust financial decision systems, addressing limitations observed in purely RL-based frameworks such as FinRL.

2.5 ProTrader-RL (2024): Multi-Module Reinforcement Learning for Stock Trading

One of the most recent and relevant studies identified during my research is *ProTrader-RL* (2024), a framework designed to emulate the behaviour and reasoning process of professional traders by dividing the decision-making task into specialised modules.
The system is composed of **four main components**:

1. **Data Preprocessing**
2. **Buy Knowledge RL**
3. **Sell Knowledge RL**
4. **Stop-Loss Rule**

Each of these plays a distinct role that mirrors the workflow of human traders.
In their experiments, the authors reported strong and stable results, consistently high returns and Sharpe ratios with low maximum drawdown (MDD) across different market conditions outperforming prior reinforcement-learning-based trading systems.

Unlike earlier frameworks such as Jiang et al.'s (2017) EIIE model, which focused on allocating assets based on historical prices, **ProTrader-RL does not restrict its state**

inputs to specific stocks or fixed financial indicators. Instead, it processes a broader spectrum of market data, representing the approach that professional traders typically adopt. The system primarily relies on **technical indicators and price series**, assuming that these features dominate short-term trading behaviour. However, the modular design leaves room to incorporate other signals, such as sentiment or macroeconomic factors.

Technically, ProTrader-RL is implemented within the **OpenAI Gym** environment and the **policy network** follows a **Deep Neural Network (DNN)** architecture with three hidden layers (69–40–2 units), used identically for both actor and critic networks. As reported in the '*Pro Trader RL: Reinforcement learning framework for generating trading knowledge by mimicking the decision-making patterns of professional traders*' article (it's mentioned in the reference) a total of sixty-nine variables are initially normalised and passed into the **two reinforcement-learning environments**. The reward function can be configured to maximise not only returns but also **risk-adjusted metrics** such as the Sharpe or Sortino ratios, allowing the agent to learn more stable and consistent trading strategies.

Compared with the approaches I studied earlier, I do believe that ProTrader-RL introduces several conceptual improvements. For example Moody & Saffell (2001) handled trading as a single-agent optimisation problem, while Jiang et al. (2017) extended this to multi-asset allocation: ProTrader-RL explicitly "splits" trading into multiple agents (aka "knowledge modules"), each optimising a different decision layer. This architecture not only reduces the learning complexity of each component but also mirrors human cognitive division of tasks (buying, selling, and risk control).
Also, from a systems perspective, this multi-module structure offers scalability and interpretability, characteristics that are often missing in several RL models such as FinRL's standard environments.

For my own project, this study is particularly inspiring: I intend to extend the **multi-agent collaboration concept / architecture** by integrating **parallel multiprocessing** (I will discuss this in more detail in the next section) and adding a **Sentiment Analysis agent** to validate Buy signals based on real-world news tone.

Furthermore, I plan to use **Genetic Algorithms** to evolve the hyperparameters of each RL module automatically, something not yet addressed in ProTrader-RL, which could make the overall system more adaptive and less sensitive to manual configuration.

In summary, ProTrader-RL (2024) represents an important evolution and milestone, it really aligns closely with my project's goal of building a transparent, adaptable, and multi-agent financial advisor.

2.6 Parallel Multi-Module Reinforcement Learning for Stock Trading

Another study that caught my attention was the *Parallel Multi-Module Reinforcement Learning Algorithm for Stock Trading* (2024), which provides an elegant and scalable interpretation of reinforcement learning applied to financial markets.
The authors formulate stock trading as a **Markov Decision Process (MDP)**, in which an agent interacts with the market by buying or selling assets and receiving rewards based on portfolio performance. The overall objective is to **maximise the expected cumulative discounted return**, the same principle underpinning the reinforcement-learning theory studied during the AI module.
In fact, this framework immediately reminded me of the **DQN Agent** I implemented during the course. Both systems are grounded in the same theoretical foundation: the **Q-value function**, which estimates the long-term expected reward of taking action while training the Atari Breakout agent.

One difference comparing to the agent we have been using during the course is that this approach uses the **Double Deep Q-Network (DDQN)** algorithm: each agent is trained independently for multiple episodes (using 75 % of the dataset for training and 25 % for testing) with realistic transaction fees of 0.1 % for buys and 0.2 % for sells.

What makes this work particularly relevant is its **parallel multi-module architecture**: rather than relying on a single network, the authors run several DDQN agents in parallel, each focusing on a **specific stock** or trading subtask. This improves efficiency, increases exploration coverage, and reduces variance between runs. The group of agents is then aggregated into a unified global policy, producing more stable overall performance.

However, the authors recognise several **limitations** that are important to note:

1. The approach depends heavily on access to reliable fundamental data, which is unavailable for many assets such as cryptocurrencies.

2. It was tested only on **single-stock trading**, so extending it to multi-asset portfolio management remains an open research direction.

3. Explicit **risk management** components are not integrated into the RL process, meaning the agent may overfit to short-term profit maximisation.

The study offers important insights for building large-scale RL systems: parallel learners can collectively produce more stable global policies, reflecting a closer approximation of real market dynamics than single-agent approaches.

2.7 Literature considerations

The last two articles I have been studying, *ProTrader-RL (2024)* and the *Parallel Multi-Module RL* framework, have been very important and useful to me: I could really see the connections with what we studied and how they try to improve some concepts (such as the double DQN or the global policy). Also, comparing to the other researches, I think they highlight the advantages of decomposing trading into modular decision units, and of running agents in parallel to accelerate learning and improve robustness.

My project builds on these two complementary ideas. It aims to integrate a **multi-agent RL architecture** inspired by ProTrader-RL, enhanced with **parallel processing** to improve exploration efficiency, and extended with **Genetic Algorithms** to evolve hyperparameters and reward functions automatically. Additionally, I will incorporate a **sentiment-analysis agent** and a **Natural-Language layer** to make the system not only adaptive and data-driven but also transparent and human-understandable.

3 Design

The target of this project is to design and develop an AI-driven Financial Advisor Bot that learns adaptive investment strategies through a multi-agent reinforcement-learning (RL) architecture enhanced by Genetic Algorithm (GA) optimisation and an NLP-based layer to provide clear instructions to the end users.

The goal is to build a system capable not only of analysing financial data and making / providing buy or sell recommendations, but also of *evolving* its internal parameters to remain effective in changing market conditions and of *explaining* its reasoning in clear, human language.

The aim of the evaluation of the design (and development) is to assess whether the Financial Advisor Bot (a) processes financial data correctly, (b) learns trading behaviour that performs better than a simple benchmark, (c) adapts when Genetic Algorithms optimise its hyperparameters, and (d) provides clear, understandable explanations of its decisions through the NLP layer.

3.1 Project overview

Based on the ProTrader-RL concept, my project aims to emulate the workflow of professional traders using several specialised RL agents that collaborate within a unified environment.
Each agent focuses on a distinct aspect of decision-making (data processing, buying, selling, and sentiment analysis) while a **Genetic Algorithm (GA)** supervises their evolution to achieve long-term adaptability.

Four primary modules are defined:

1. Data Preprocessing Agent

Collects, cleans, and normalises market data (prices, volume, technical indicators) and computes a sentiment index from online financial news and social media. This sentiment score provides contextual information to the RL agents, improving the representation of market conditions.

2. Buy Knowledge Agent

Learns optimal entry points through reinforcement learning.
Following the parallel multi-processing strategy described in the literature review (3.6), several identical Buy agents are trained simultaneously on different market segments or time windows.
Each sub-agent explores different state–action trajectories, and their collective experience accelerates convergence and improves generalisation.

3. Sell Knowledge Agent

Learns exit strategies such as profit-taking, risk control, and closing positions. Like the Buy Agent, it operates in parallel environments and shares the same DDQN architecture, ensuring symmetry between entry and exit behaviour.

3. Sentiment Validation Agent

While the sentiment index offers a lightweight, quantitative mood feature during the data preprocessing activity, this **higher-level sentiment module** performs a deeper analysis of live news articles, earnings reports, and social-media data using NLP techniques.
Its purpose is to **validate (or even block)** trading actions suggested by the Buy and Sell agents, providing an additional reasoning layer similar to a human analyst double-checking a model's recommendation.

The 'trading' agents (buy and sell) are coordinated by an evolutionary GA trainer, which evolves each agent's hyper-parameters (learning rate, discount factor, exploration ratio, reward coefficients) according to a fitness function that measures risk-adjusted return, stability, and drawdown.
This outer optimisation loop enables self-adaptation across generations rather than relying on manual tuning.

Finally, an **NLP Interface Layer** translates quantitative outputs (e.g., "Buy AAPL @ 173 USD, expected Sharpe 1.4") into accessible natural-language recommendations such as

---

"*The system suggests opening a position in Apple Inc. this week, as technical momentum and sentiment remain positive with moderate risk exposure.*"

3.2 Domain and Users

The domain is algorithmic trading and financial decision support, which naturally demands adaptability and interpretability.
Financial markets are non-stationary: volatility, sentiment, and macroeconomic changes can render static models obsolete.
RL provides adaptability, while the NLP layer supports interpretability, two essential requirements in finance.

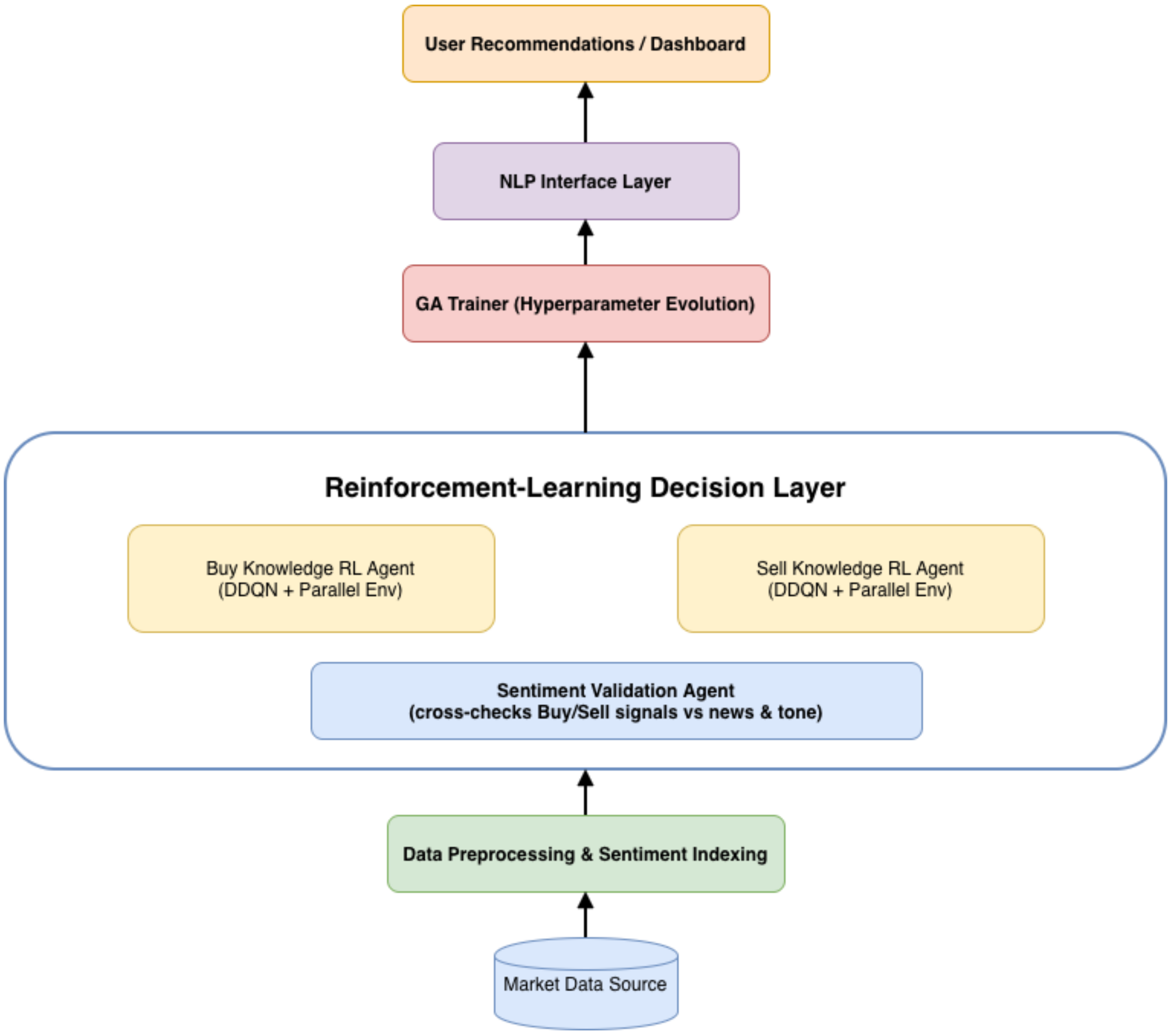
Although the system is built as a research prototype, potential users include:

- **Individual retail investors:** seeking understandable, data-driven recommendations.
- **Quantitative researchers:** testing RL and GA configurations, reward structures, and parallelised training.
- Fintech developers or institutions (future potential): exploring explainable, multi-agent advisory systems.

The design explicitly aims to balance three priorities: adaptive intelligence, modularity, and user transparency.

3.3 System Architecture

The overall architecture of the *Financial Advisor Bot* follows a **multi-agent reinforcement-learning framework** inspired by *ProTrader-RL (2024)* and extended with **parallel multiprocessing**, **genetic hyperparameter evolution**, and **natural-language interpretability**.
The system emulates the workflow of professional traders, with distinct agents specialising in data preparation, decision-making, and validation.
Each agent operates within a unified environment coordinated by a Genetic Algorithm (GA) that evolves optimal learning parameters across generations.
At a high level, the system comprises **five functional layers**, as illustrated in the following diagram


User Recommendations / Dashboard
NLP Interface Layer
GA Trainer (Hyperparameter Evolution)
Reinforcement-Learning Decision Layer
Buy Knowledge RL Agent (DDQN + Parallel Env)
Sell Knowledge RL Agent (DDQN + Parallel Env)
Sentiment Validation Agent (cross-checks Buy/Sell signals vs news & tone)
Data Preprocessing & Sentiment Indexing
Market Data Source

Layer 1 - Data Preprocessing and Sentiment Indexing

Purpose:
Collect and normalise raw market and fundamental data (e.g., OHLC prices, volumes, volatility, moving averages, P/E, etc) together with a preliminary *sentiment index* derived from financial news, social media, or analyst commentary.

Functions:
- Cleans missing or noisy data and standardises numerical ranges.
- Aligns multi-source time series into consistent state representations for training.
- Computes a sentiment polarity score (e.g., via FinBERT or VADER) and appends it to the state vector, providing context for later trading decisions.

Output:

A structured, time-aligned tensor representing both quantitative and sentiment-driven market states.

Layer 2 - Reinforcement-Learning Decision Layer

This layer contains three cooperative agents: **Buy Knowledge Agent**, **Sell Knowledge Agent**, and the **Sentiment Validation Agent**.
All three interact in a shared environment following the Markov-Decision-Process (MDP) framework studied in the course.

Buy Knowledge Agent

- Learns *entry strategies* using a **Double Deep Q-Network (DDQN)** to mitigate Q-value overestimation.
- Each DDQN instance is trained in **multiple parallel environments** (using multiprocessing) to accelerate exploration and stabilise learning.
- Inputs: price trends, volatility indicators, and sentiment index.
- Outputs: buy probabilities and confidence levels for candidate assets.

Sell Knowledge Agent

- Learns exit or profit-taking strategies, also via **DDQN + parallel environments**.
- Aims to maximise cumulative reward while minimising drawdown and exposure time.
- Shares architecture with the Buy Agent for consistency and symmetry.

Sentiment Validation Agent

- Embedded *within* the RL layer, this agent validates the Buy/Sell signals against deeper market sentiment.
- Analyses news headlines, financial reports, and social-media tone to ensure alignment between model decisions and contextual sentiment.
- Produces a sentiment-consistency score that moderates or vetoes trading actions when negative tone conflicts with Buy recommendations.
- Enhances realism and robustness by reducing noise-driven trades.

Reward Design:
All RL agents optimise a *risk-adjusted reward* (Sharpe or Sortino ratio) instead of raw profit, balancing return and volatility.
Rewards are immediate (portfolio value delta) and cumulative over episodes.

Layer 3 - Genetic-Algorithm Trainer

The GA acts as an **outer-loop optimiser**, evolving the hyperparameters of the Buy and Sell RL agents rather than executing trades itself.

Process:

1. Initialise a population of RL agent configurations (learning rate, discount factor γ, exploration decay, reward coefficients).
2. Train each configuration for a limited number of episodes.
3. Evaluate *fitness* using a composite metric combining Sharpe ratio, stability (return variance), and maximum drawdown.
4. Apply *selection, crossover,* and *mutation* to evolve new configurations.
5. Iterate until convergence to stable, high-performing policies.

Outcome:
Self-adaptive RL agents that require minimal manual tuning and exhibit improved generalisation to new market conditions.

Layer 4 - Natural-Language Interface

Translate quantitative agent outputs into **human-readable financial advice** for non-technical users.

Functions:

- Converts structured recommendations into plain language, e.g.:
  "Based on current market trends and positive sentiment, the system suggests opening a short-term position in Apple (AAPL) with moderate risk."

- Supports simple query responses such as "Why sell Tesla now?" by referencing internal agent metrics and sentiment signals.
- Provides transparency, fostering user trust.

Layer 5 - User Interaction and Evaluation Dashboard

Provide a lightweight web-based dashboard that allows users to:

- View active recommendations and rationales.
- Monitor portfolio evolution and key performance indicators (Sharpe ratio, drawdown, cumulative return).
- Compare model vs. benchmark (buy-and-hold) performance.

3.4 Key Technologies and Methods

During the last year studying, I explored key techniques such as Reinforcement Learning (RL), Deep Q-Networks (DQN), and Genetic Algorithms (GA), which provided the conceptual basis for this work.
This project builds directly on those methods and techniques with key enhancements identified through my background and literature activities, such as:

- the use of **Double Deep Q-Networks (DDQN)** to improve learning stability

- the introduction of **parallel multiprocessing** to accelerate training and increase robustness

- the integration of a **Genetic Algorithm optimiser** to evolve hyperparameters automatically and reduce manual tuning bias.

3.4.1 Data Preprocessing & Sentiment Indexing

- **Data sources**: Yahoo Finance, Alpha Vantage, or Alpaca API
- **Libraries**: pandas, numpy, ta-lib, sklearn
- **Sentiment tools**: FinBERT or VADER for polarity scoring
- **Output**: State tensors combining OHLC, indicators, and sentiment index

3.4.2 Reinforcement-Learning Decision Layer

- **Framework**: OpenAI Gym, Stable-Baselines3 (PyTorch backend)
- **Algorithm**: Double DQN (DDQN) with parallel environments (multiprocessing)
- **Reward**: Risk-adjusted Sharpe ratio; actions {buy, hold, sell}
- **Agents**:
    - Buy Knowledge Agent: entry optimisation
    - Sell Knowledge Agent: exit optimisation
    - Sentiment Validation Agent: tone alignment (FinBERT)
- **Technologies:** PyTorch, OpenAI Gym, Ray RLlib (for parallelised environments).

3.4.3 Genetic Algorithm Optimiser

- **Genome encoding**: learning rate, γ, ε-decay, reward weighting
- **Fitness function**: $F = \alpha \cdot \text{Sharpe} - \beta \cdot \text{Drawdown} + \gamma \cdot \text{Stability}$
- **Evolution**: Selection, Crossover, Mutation

3.4.4 NLP Interface Layer

- **LLM**: Local model (e.g. LLaMA 3 via Ollama) or OpenAI GPT API
- **Goal**: Explain “why” behind buy/sell decisions in plain English
- **Output**: “The model detected strong momentum and positive sentiment in NVDA; it suggests a short-term buy position.”

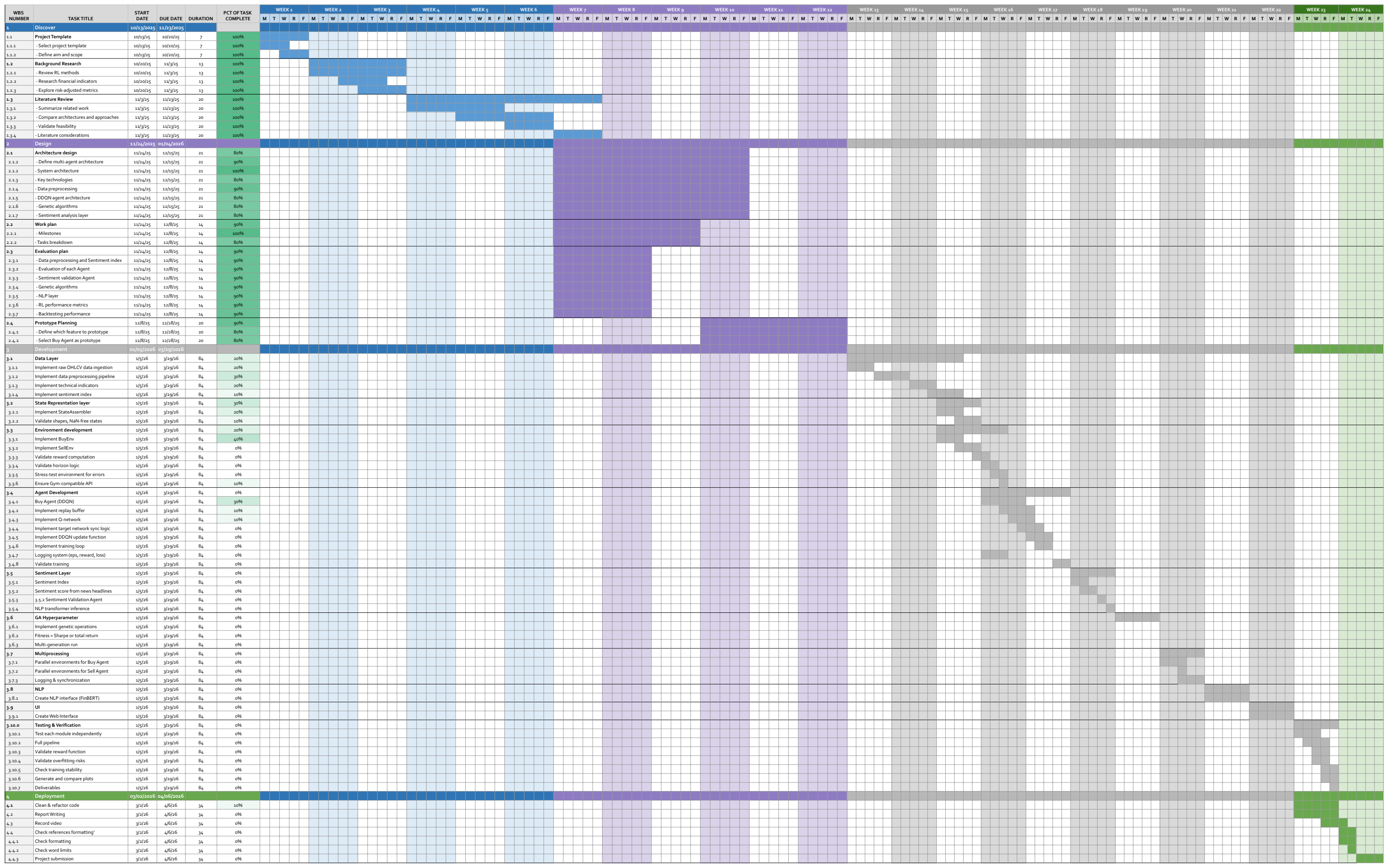
3.4.5 User Dashboard / Evaluation

- **Framework**: Streamlit or Flask

- **KPIs**: Sharpe, Sortino, Max Drawdown, Cumulative Return
- **Visualization**: Plotly or Matplotlib
- **Backtesting**: backtesting.py or custom simulation engine

3.5 Work plan

All the dates are visible close to the category (Discovery, Design, Development and Delivery)



WBS NUMBER	TASK TITLE	START DATE	DUE DATE	DURATION	PCT OF TASK COMPLETE
1	Discover	10/13/2025	11/23/2025		
1.1	**Project Template**	10/13/25	10/20/25	7	100%
1.1.1	- Select project template	10/13/25	10/20/25	7	100%
1.1.2	- Define aim and scope	10/13/25	10/20/25	7	100%
1.2	**Background Research**	10/20/25	11/3/25	13	100%
1.2.1	- Review RL methods	10/20/25	11/3/25	13	100%
1.2.2	- Research financial indicators	10/20/25	11/3/25	13	100%
1.2.3	- Explore risk-adjusted metrics	10/20/25	11/3/25	13	100%
1.3	**Literature Review**	11/3/25	11/23/25	20	100%
1.3.1	- Summarize related work	11/3/25	11/23/25	20	100%
1.3.2	- Compare architectures and approaches	11/3/25	11/23/25	20	100%
1.3.3	- Validate feasibility	11/3/25	11/23/25	20	100%
1.3.4	- Literature considerations	11/3/25	11/23/25	20	100%
2	Design	11/24/2025	01/04/2026		
2.1	**Architecture design**	11/24/25	12/15/25	21	80%
2.1.2	- Define multi-agent architecture	11/24/25	12/15/25	21	90%
2.1.2	- System architecture	11/24/25	12/15/25	21	100%
2.1.3	- Key technologies	11/24/25	12/15/25	21	80%
2.1.4	- Data preprocessing	11/24/25	12/15/25	21	90%
2.1.5	- DDQN agent architecture	11/24/25	12/15/25	21	80%
2.1.6	- Genetic algorithms	11/24/25	12/15/25	21	80%
2.1.7	- Sentiment analysis layer	11/24/25	12/15/25	21	80%
2.2	**Work plan**	11/24/25	12/8/25	14	90%
2.2.1	- Milestones	11/24/25	12/8/25	14	100%
2.2.2	- Tasks breakdown	11/24/25	12/8/25	14	80%
2.3	**Evaluation plan**	11/24/25	12/8/25	14	90%
2.3.1	- Data preprocessing and Sentiment index	11/24/25	12/8/25	14	90%
2.3.2	- Evaluation of each Agent	11/24/25	12/8/25	14	90%
2.3.3	- Sentiment validation Agent	11/24/25	12/8/25	14	90%
2.3.4	- Genetic algorithms	11/24/25	12/8/25	14	90%
2.3.5	- NLP layer	11/24/25	12/8/25	14	90%
2.3.6	- RL performance metrics	11/24/25	12/8/25	14	90%
2.3.7	- Backtesting performance	11/24/25	12/8/25	14	90%
2.4	**Prototype Planning**	12/8/25	12/28/25	20	90%
2.4.1	- Define which feature to prototype	12/8/25	12/28/25	20	80%
2.4.2	- Select Buy Agent as prototype	12/8/25	12/28/25	20	80%
3	Development	01/05/2026	03/29/2026		
3.1	**Data Layer**	1/5/26	3/29/26	84	20%
3.1.1	Implement raw OHLCV data ingestion	1/5/26	3/29/26	84	20%
3.1.2	Implement data preprocessing pipeline	1/5/26	3/29/26	84	30%
3.1.3	Implement technical indicators	1/5/26	3/29/26	84	20%
3.1.4	Implement sentiment index	1/5/26	3/29/26	84	10%
3.2	**State Represntation layer**	1/5/26	3/29/26	84	30%
3.2.1	Implement StateAssembler	1/5/26	3/29/26	84	20%
3.2.2	Validate shapes, NaN-free states	1/5/26	3/29/26	84	10%
3.3	**Environment development**	1/5/26	3/29/26	84	20%
3.3.1	Implement BuyEnv	1/5/26	3/29/26	84	40%
3.3.2	Implement SellEnv	1/5/26	3/29/26	84	0%
3.3.3	Validate reward computation	1/5/26	3/29/26	84	0%
3.3.4	Validate horizon logic	1/5/26	3/29/26	84	0%
3.3.5	Stress-test environment for errors	1/5/26	3/29/26	84	0%
3.3.6	Ensure Gym-compatible API	1/5/26	3/29/26	84	10%
3.4	**Agent Development**	1/5/26	3/29/26	84	0%
3.4.1	Buy Agent (DDQN)	1/5/26	3/29/26	84	30%
3.4.2	Implement replay buffer	1/5/26	3/29/26	84	10%
3.4.3	Implement Q-network	1/5/26	3/29/26	84	10%
3.4.4	Implement target network sync logic	1/5/26	3/29/26	84	0%
3.4.5	Implement DDQN update function	1/5/26	3/29/26	84	0%
3.4.6	Implement training loop	1/5/26	3/29/26	84	0%
3.4.7	Logging system (eps, reward, loss)	1/5/26	3/29/26	84	0%
3.4.8	Validate training	1/5/26	3/29/26	84	0%
3.5	**Sentiment Layer**	1/5/26	3/29/26	84	0%
3.5.1	Sentiment Index	1/5/26	3/29/26	84	0%
3.5.2	Sentiment score from news headlines	1/5/26	3/29/26	84	0%
3.5.3	3.5.2 Sentiment Validation Agent	1/5/26	3/29/26	84	0%
3.5.4	NLP transformer inference	1/5/26	3/29/26	84	0%
3.6	**GA Hyperparameter**	1/5/26	3/29/26	84	0%
3.6.1	Implement genetic operations	1/5/26	3/29/26	84	0%
3.6.2	Fitness = Sharpe or total return	1/5/26	3/29/26	84	0%
3.6.3	Multi-generation run	1/5/26	3/29/26	84	0%
3.7	**Multiprocessing**	1/5/26	3/29/26	84	0%
3.7.1	Parallel environments for Buy Agent	1/5/26	3/29/26	84	0%
3.7.2	Parallel environments for Sell Agent	1/5/26	3/29/26	84	0%
3.7.3	Logging & synchronization	1/5/26	3/29/26	84	0%
3.8	**NLP**	1/5/26	3/29/26	84	0%
3.8.1	Create NLP interface (FinBERT)	1/5/26	3/29/26	84	0%
3.9	**UI**	1/5/26	3/29/26	84	0%
3.9.1	Create Web Interface	1/5/26	3/29/26	84	0%
3.10.0	**Testing & Verification**	1/5/26	3/29/26	84	0%
3.10.1	Test each module independently	1/5/26	3/29/26	84	0%
3.10.2	Full pipeline	1/5/26	3/29/26	84	0%
3.10.3	Validate reward function	1/5/26	3/29/26	84	0%
3.10.4	Validate overfitting risks	1/5/26	3/29/26	84	0%
3.10.5	Check training stability	1/5/26	3/29/26	84	0%
3.10.6	Generate and compare plots	1/5/26	3/29/26	84	0%
3.10.7	Deliverables	1/5/26	3/29/26	84	0%
4	Deployment	03/02/2026	04/06/2026		
4.1	Clean & refactor code	3/2/26	4/6/26	34	10%
4.2	Report Writing	3/2/26	4/6/26	34	0%
4.3	Record video	3/2/26	4/6/26	34	0%
4.4	Check references formatting'	3/2/26	4/6/26	34	0%
4.4.1	Check formatting	3/2/26	4/6/26	34	0%
4.4.2	Check word limits	3/2/26	4/6/26	34	0%
4.4.3	Project submission	3/2/26	4/6/26	34	0%

3.6 Evaluation Plan

Evaluation proceeds in modular stages to ensure each component works correctly before full integration. All tests use the same dataset, identical transaction costs, and no look-ahead bias. The benchmark is **S&P500** buy-and-hold.

3.6.1 Data Preprocessing & Sentiment Index

Objective: verify correct data handling and reliable sentiment scoring.

Procedure:

- Run two matched backtests:
  - A: indicators only
  - B: indicators + sentiment
- Validate:
  - missing-data handling
  - time alignment
  - sentiment reaction to real news events

Measures:
missing-data ratio, sentiment alignment, cumulative return vs. drawdown.

3.6.2 Buy Agent

Objective: confirm the DDQN Buy Agent outperforms a simple baseline.

Procedure:

- A: buy-and-hold baseline
- B: DDQN Buy Agent

Measures:
cumulative return, drawdown, proportion of profitable vs. unprofitable entries.

3.6.3 Sell Agent

Objective: confirm learned exit strategies reduce losses.

Procedure:

- A: Buy Agent + simple exit rule
- B: Buy Agent + learned Sell Agent

Measures:
drawdown reduction, loss-trade reduction, impact on overall return.

3.6.4 Sentiment Validation Agent

Objective: test whether sentiment veto reduces poor trades.

Procedure:
- A: Buy + Sell only
- B: Buy + Sell + sentiment validation

Measures:
drawdown stability, reduction in large negative trades, change in turnover.

3.6.5 Genetic Algorithm

Objective: test whether GA-optimised hyperparameters improve stability.

Procedure:
- A: manually chosen hyperparameters
- B: GA-optimised hyperparameters

Measures:
return, drawdown, consistency across random seeds.

3.6.6 NLP Layer

Objective: test clarity and trust in explanations.

Procedure:
5–10 non-experts evaluate:
- A: recommendation only
- B: recommendation + explanation

Measures:
clarity, trust, usefulness (1–5 scale).

3.6.7 Full System Evaluation (PoC)

Objective: evaluate the complete pipeline.

Procedure:
Compare:
- full system
- S&P500 baseline
- technical-rule baseline

- ablated variants (no GA, no sentiment)

Measures:
Sharpe, drawdown, cumulative return, consistency.

Success criteria:
- comparable or better than S&P500
- significantly reduced drawdown
- explanations understandable to users

4 Prototype: Buy Agent with DDQN and Market Data Pipeline

The purpose of this prototype is to demonstrate the feasibility of one of the most central technical components of the overall system: the **Buy Knowledge Agent**, implemented using a **Double Deep Q-Network (DDQN)** and trained inside a custom **OpenAI Gym** environment that encodes both market conditions and technical indicators. The prototype focuses on the "entry decision" problem, for example deciding whether it is the right moment to open a long position). This is the first step since Buy and Sell agents together are the core decision-making layer of the architecture designed in the previous chapter.
Rather than attempting to build the full multi-agent system at this early stage, I focused on:

- building and validating the **data preprocessing + rolling-state pipeline**,
- designing and testing a **custom environment** that follows the Markov Decision Process (MDP) structure discussed in the literature review,
- implementing a **DDQN agent** with replay buffer, target network and ε-greedy exploration,
- training the Buy agent for a small number of episodes
- evaluating its behaviour through a greedy (no-exploration) run and through a basic visualisation of buy signals over the price series.

The prototype therefore provides a minimal but complete version of the reinforcement-learning pipeline for financial decision-making. It also matches the recommendations made in the background research: the literature review showed that single-pass or naïve DQN agents tend to overestimate Q-values and become unstable in financial environments. For this specific reason, the prototype uses **Double DQN**, which separates the action-selection and value-evaluation networks, as well as a replay buffer to decorrelate updates.

In later iterations I will extend it with the **Sell agent**, the **sentiment validation agent**, and the **GA hyperparameter optimiser**.

4.1 Dataset and State Construction

To ensure realism and reproducibility, the prototype uses real historical market data. The chosen dataset for this first experiment is **Apple (AAPL)**, fetched from Yahoo Finance via the *yfinance* library. The initial dataset contains daily OHLCV prices for the interval defined in the project configuration file (*config/data_config.yaml*).

The feature approach follows the design described in Section 4.1 of the report: we compute a set of core technical indicators (RSI, MACD, Bollinger Bands p-band, ATR, ROC, OBV, MFI, Williams %R) and append the adjusted close price at each date.

These features are computed using the *ta* library and arranged in a Pandas DataFrame. In line with the evaluation plan, I focused on having clean and possibly leak-free features:

- all indicators are lagged by one timestep,
- any rows containing NaNs are dropped,
- all features are normalised implicitly through their original scale, without future information leaking forward.

Once features are validated, I constructed the RL state. Since the agent must learn from local market patterns, each state is formed by a rolling window of 30 timesteps, flattened into a single feature vector. This creates a final state dimensionality of 270 features per state (30 days × 9 indicators).

The final dataset sizes are:

- Raw dataset shape: **(1224, 10)**
- Rolling-state shape: **(1194, 270)**
- No missing values in either the state tensor or the price series
-

The rolling state vector and aligned price series are then passed into the custom *BuyEnv*.

4.2 Custom Buy Environment

The custom environment formalises the buy-decision problem as a short-horizon evaluation task. The agent sees a state vector and may choose between two actions:

- **0 = HOLD**
- **1 = BUY**

When the agent **buy**s, the episode ends immediately. The reward is computed as the **K-day forward return** (with K=5 in this prototype), minus a small transaction cost (0.1%). This formulation aligns with findings from multiple papers reviewed: buy decisions are often better modelled as discrete entry events with short-term horizon labels, rather than continuous portfolio trading.

In practice:
```
reward = (exit_price - entry) / entry - self.cost
```

The episode ends either when the agent **buys** or when the end of the time series is reached (no further forward return is possible). This environment is extremely lightweight, making it ideal for early-stage testing of DDQN behaviour.

A simple reset and step loop was tested manually before training, and the environment correctly returned:

Initial state shape: (270,)
step=0, action=1, reward=0.0157, done=True, info={'t': 1, 'price': 47.02927780151367}

This validated that the reward logic, indexing, and state assembly were correct.

4.3 Double Deep Q-Network Implementation

The DDQN agent follows the 'usual' structure:

- **Online Q-network** (*q_net*)
- **Target Q-network** (*target_net*)
- **Replay buffer**
- **Mini-batch updates** (batch size 64)
- **Huber/MSE loss and Adam optimiser**
- **Target network update every 500 learning steps**
- **Epsilon-greedy schedule for exploration -> exploitation:**

```
epsilon_start=1.0,
epsilon_end=0.05,
epsilon_decay_steps=5_000,
```

The core DDQN update rule is implemented in the agent:

1. Choose next action from online network
2. Evaluate that action via the target network.
3. Compute Bellman target:

```
     with torch.no_grad():
            next_q_online = self.q_net(next_states)
            next_actions = torch.argmax(next_q_online, dim=1)
            next_q_target = self.target_net(next_states)
            next_q_sa = next_q_target.gather(
                1, next_actions.unsqueeze(1)
            ).squeeze(1)

            target = rewards + self.gamma * (1.0 - dones) *
next_q_sa
```

4.4 Training the Prototype

The prototype was trained on:

- 50 episodes
- Warmup: 500 steps to fill replay buffer
- Window=30, horizon=5
- Transaction cost=0.001

During training, the console logs showed stable decreasing ε and varied rewards:

```
[Episode 1/50] Reward=0.0404, Epsilon=0.998
[Episode 10/50] Reward=0.0115, Epsilon=0.995
[Episode 20/50] Reward=0.0157, Epsilon=0.992
[Episode 30/50] Reward=-0.0023, Epsilon=0.987
[Episode 40/50] Reward=0.0085, Epsilon=0.983
[Episode 50/50] Reward=0.0157, Epsilon=0.979
```

The variation between episodes is expected since each episode corresponds to a single pass through the time series, and the agent may choose different buy timings based on ε-greedy exploration.

The final 10 rewards were:

```
Final rewards: [0.015658972948789596, 0.008492360986769198,
0.015658972948789596, -0.005347877576947212,
-0.0022531328247860074, 0.011511264532804488,
0.015658972948789596, 0.008492360986769198,
0.015658972948789596, 0.015658972948789596]
```

This confirms that the reward distribution is realistic (sometimes positive, sometimes negative) and the agent is correctly learning that some entries are better than others.

4.5 Greedy Policy Evaluation

After training, a greedy (ε=0) policy was executed:

```
Greedy run: total_reward=-0.0053, steps=3
```

This means:

- the agent decided to **buy** after 3 timesteps,
- the resulting 5-day forward return was slightly negative (-0.5%)

I would consider this as a realistic and acceptable for a prototype: markets are noisy, and the prototype is trained for only ~100 steps. What I think is important is:

- the agent is functional
- decisions do not crash
- the RL training loop is stable
- rewards propagate correctly
- environment terminates properly
- DDQN updates are happening

This validates feasibility, which is the goal for the prototype.

4.6 Prototype Notebook

To better understand the behaviour, I created a python notebook including a plot of:

- the price series
- the buy points suggested by the greedy agent

The expected behaviour is:

- only a single BUY point per episode
- BUY tends to occur in local upward-trend regions
- earlier buys indicate over-optimism
- later buys indicate more conservative behaviour

The final notebook (included below) generates:

- Training reward curve
- Price vs BUY marker plot
- DDQN loss curve

4.7 Evaluation of the Prototype

One of the targets for the prototype was of course to align with the literature, which was based again on what we studied during the course, The prototype design did follow the following best practices:

- DDQN instead of DQN (to reduce Q-overestimation)
- replay buffer for decorrelated updates
- target network for stability
- short-horizon reward structure

The prototype successfully demonstrates:

- the full data-to-state pipeline

- the correct construction of rolling window states
- a functional custom Gym environment
- a complete DDQN agent with replay buffer
- stable training behaviour and convergence of epsilon
- greedy evaluation without errors

For the prototype, the warmup period for the replay buffer is intentionally set to 10 steps because our BuyEnv produces very short episodes (due to the one-shot BUY structure). In the production version, warmup will be dynamically scaled to the dataset length (e.g., 200–1000 steps) and BuyEnv will be extended to support multi-step episodes instead of terminating after a BUY. This enables the replay buffer to contain a more diverse set of transitions before training begins, improving stability and reducing overfitting.

4.8 Conclusion

The prototype demonstrates that the reinforcement-learning core of the project (the Buy Knowledge Agent) is technically feasible, correctly implemented, and behaves in a stable and interpretable manner.
The production version will adopt multi-process training, larger replay buffers, risk-adjusted rewards, and GA-driven hyperparameter tuning. These changes are intentionally excluded from the prototype to keep the PoC lean, transparent, and easy to evaluate.
Here below the screenshots of the ipynb I created to test and evaluate the prototype.

prototype

November 23, 2025

1 Import and config

Using this cell to import all the necessary libraries

[1]:
```
# %pip install matplotlib
```

[2]:
```
import yaml
import pandas as pd
import matplotlib.pyplot as plt

from pipeline.build_dataset import make_state_frame
from features.state_assembler import StateAssembler
from envs.buy_env import BuyEnv
from agents.buy_agent_ddqn import BuyAgentTrainer
```

```
Gym has been unmaintained since 2022 and does not support NumPy 2.0 amongst
other critical functionality.
Please upgrade to Gymnasium, the maintained drop-in replacement of Gym, or
contact the authors of your software and request that they upgrade.
Users of this version of Gym should be able to simply replace 'import gym' with
'import gymnasium as gym' in the vast majority of cases.
See the migration guide at
https://gymnasium.farama.org/introduction/migration_guide/ for additional
information.
```

2 Load config and data

[3]:
```
cfg = yaml.safe_load(open("config/data_config.yaml"))

ticker = "AAPL"
dataset = make_state_frame(ticker, cfg)
print("Raw dataset shape:", dataset.shape)
print(dataset.tail())
```

```
[*********************100%***********************]  1 of 1 completed

Raw dataset shape: (1224, 10)
            return_1d     rsi14  macd_diff      bb_b     atr14     roc10  \
Date
```

```
2023-12-22  -0.182773  0.089437  -0.627491 -0.178014 -1.820035 -0.282916
2023-12-26  -0.538524 -0.175613  -0.849465 -0.533543 -1.984075 -0.573011
2023-12-27  -0.341273 -0.314185  -1.030830 -0.759738 -2.209548 -0.357041
2023-12-28  -0.084725 -0.299801  -1.123309 -0.790062 -2.209548 -0.531208
2023-12-29   0.041502 -0.208808  -1.127636 -0.746857 -2.209548 -0.867669

                 obv     mfi14   willr14       price
Date
2023-12-22  1.349133 -0.061185 -0.080590  191.788773
2023-12-26  1.259801 -0.044919 -0.988173  191.243912
2023-12-27  1.190502 -0.392735 -1.621949  191.342987
2023-12-28  1.300912 -0.723739 -1.453421  191.768951
2023-12-29  1.382170 -0.785871 -1.292257  190.728745
```

3 Building rolling states

[4]:
```
feature_cols = [c for c in dataset.columns if c != "price"]
assembler = StateAssembler(feature_cols=feature_cols, window_size=30)

state_df = assembler.assemble(dataset)
prices = dataset["price"].iloc[30:]  # align with state_df

print("state_df shape:", state_df.shape)
print("prices shape:", prices.shape)
print("NaNs in state_df:", state_df.isna().sum().sum())
print("NaNs in prices:", prices.isna().sum())
```

```
state_df shape: (1194, 270)
prices shape: (1194,)
NaNs in state_df: 0
NaNs in prices: 0
```

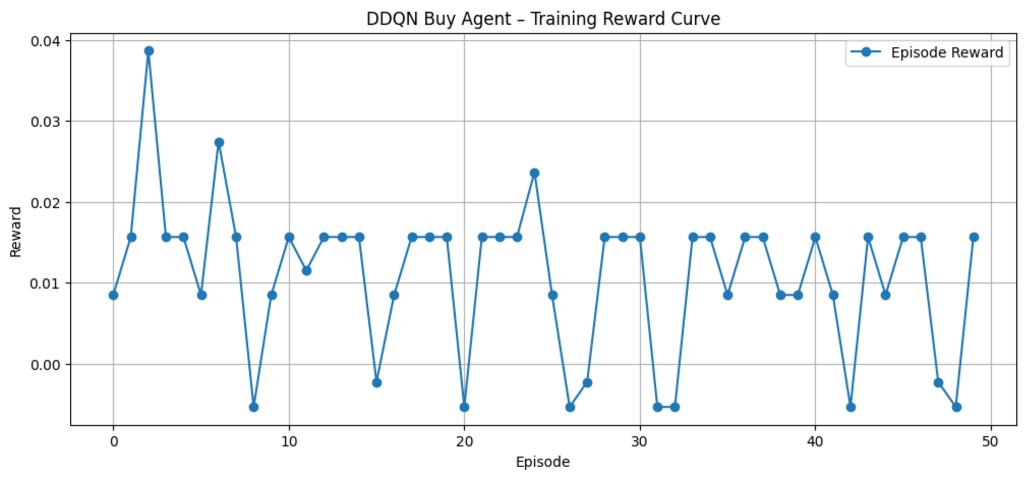
4 Train DDQN Buy Agent

[5]:
```
trainer = BuyAgentTrainer(
        cfg_path="config/data_config.yaml",
        ticker="AAPL",
        window_size=30,
        horizon=5,
        transaction_cost=0.001,
    )

rewards = trainer.train(
    n_episodes=50,
    warmup_steps=500,
    max_steps_per_episode=None,
)

print("Last 10 episode rewards:", rewards[-10:])

greedy_policy = trainer.make_greedy_policy()
total_reward, steps = greedy_policy()
print(f"Greedy run: total_reward={total_reward:.4f}, steps={steps}")
```

```
[*********************100%***********************]  1 of 1 completed

[BuyTrainer] Raw dataset shape for AAPL: (1224, 10)
[BuyTrainer] After dropna: (1224, 10)
[BuyTrainer] Rolling state_df shape: (1194, 270)
[BuyTrainer] state_dim=270, n_actions=2
[Episode 1/50] Reward=0.0085, Epsilon=1.000
[Episode 10/50] Reward=0.0085, Epsilon=0.994
[Episode 20/50] Reward=0.0157, Epsilon=0.991
[Episode 30/50] Reward=0.0157, Epsilon=0.987
[Episode 40/50] Reward=0.0085, Epsilon=0.983
[Episode 50/50] Reward=0.0157, Epsilon=0.980
Training complete.
Final rewards: [0.015659015625715256, 0.008492113091051579,
-0.0053479536436498165, 0.015659015625715256, 0.008492113091051579,
0.015659015625715256, 0.015659015625715256, -0.0022531943395733833,
-0.0053479536436498165, 0.015659015625715256]
Last 10 episode rewards: [0.015659015625715256, 0.008492113091051579,
-0.0053479536436498165, 0.015659015625715256, 0.008492113091051579,
0.015659015625715256, 0.015659015625715256, -0.0022531943395733833,
-0.0053479536436498165, 0.015659015625715256]
Greedy run: total_reward=0.0157, steps=1
```

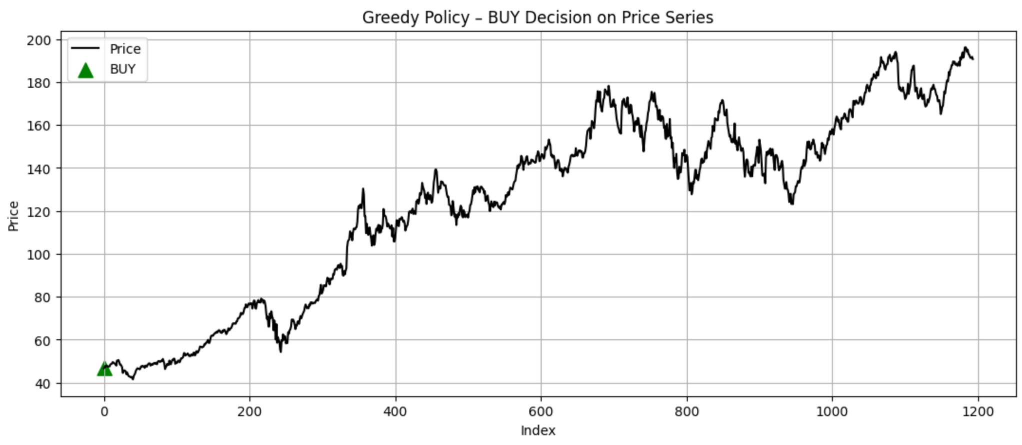
5 Plot training reward curve

[6]:
```
plt.figure(figsize=(12,5))
plt.plot(rewards, label="Episode Reward", marker="o")
plt.title("DDQN Buy Agent - Training Reward Curve")
plt.xlabel("Episode")
plt.ylabel("Reward")
plt.grid(True)
plt.legend()
plt.show()
```



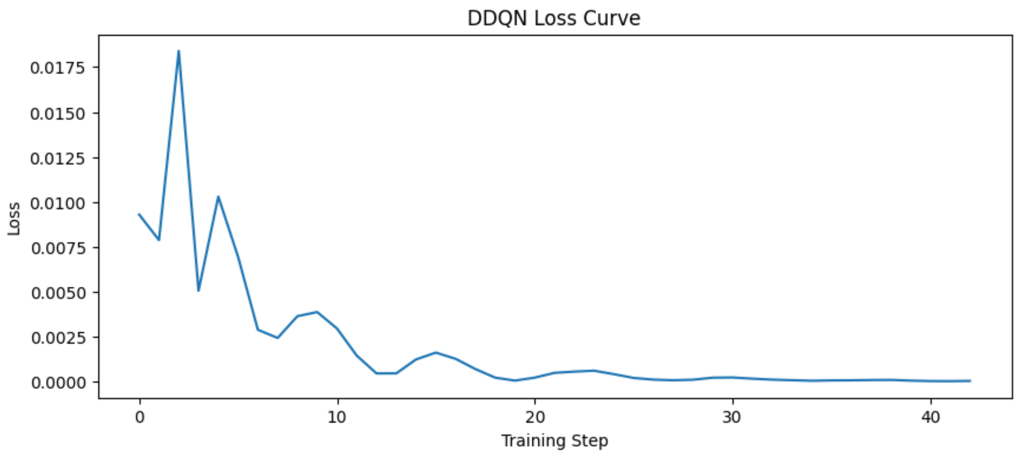
6 Buy signal

[7]:
```
# Get environment price series
prices = trainer.prices.copy()
index = prices.index

# Re-run greedy to get signal points
env = trainer.env
state = env.reset()
done = False
buy_index = None

while not done:
    a = trainer.agent.select_action(state, greedy=True)
    if a == 1:    # BUY
        buy_index = env.idx
    state, r, done, info = env.step(a)

plt.figure(figsize=(13,5))
plt.plot(prices.values, label="Price", color="black")
if buy_index is not None:
    plt.scatter([buy_index], [prices.iloc[buy_index]], s=120, color="green",␣
↪label="BUY", marker="^")
plt.title("Greedy Policy - BUY Decision on Price Series")
plt.xlabel("Index")
plt.ylabel("Price")
plt.legend()
plt.grid(True)
plt.show()
```



7 Loss history

[8]:
```
plt.figure(figsize=(10,4))
plt.plot(trainer.agent.loss_history)
plt.title("DDQN Loss Curve")
plt.xlabel("Training Step")
plt.ylabel("Loss")
plt.show()
```



References

[1] Salman Bahoo, Marco Cucculelli, Xhoana Goga, and Jasmine Mondolo. 2024. *Artificial intelligence in finance: A comprehensive review through bibliometric and content analysis.* SN Business & Economics 4. https://doi.org/10.1007/s43546-023-00618-x

[2] Francesco Bertoluzzo and Marco Corazza. 2012. *Testing different reinforcement learning configurations for financial trading: Introduction and applications.* Procedia Economics and Finance 3, 68–77. https://doi.org/10.1016/S2212-5671(12)00122-0

[3] Andrew G. Barto and Richard S. Sutton. 1996. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. MIT Press.

[4] Dimitri P. Bertsekas and John N. Tsitsiklis. 1996. *Neuro-Dynamic Programming*. Athena Scientific.

[5] Charles Gold. 2003. *FX trading via recurrent reinforcement learning*. Proceedings of the IEEE International Conference on Computational Intelligence for Financial Engineering, 363–370.

[6] John Moody, Lizhong Wu, Yuansong Liao, and Matthew Saffell. 1998. *Performance functions and reinforcement learning for trading systems and portfolios.* Journal of Forecasting 17, 441–470.

[7] William D. Smart and Leslie Pack Kaelbling. 2000. *Practical reinforcement learning in continuous spaces*. Proceedings of the 17th International Conference on Machine Learning, 3–10.

[8] Gregory Zuckerman. 2019. *The Man Who Solved the Market: How Jim Simons Launched the Quant Revolution.* Penguin Press.

[9] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction* (2nd ed.). MIT Press.

[10] John H. Holland. 1975. Adaptation in Natural and Artificial Systems. University of Michigan Press.
[11] John Moody and Matthew Saffell. 2001. *Learning to trade via direct reinforcement*. IEEE Transactions on Neural Networks 12, 4, 875–889. https://doi.org/10.1109/72.935097

[12] PwC. 2017. *Sizing the prize: What's the real value of AI for your business and how can you capitalise?* PwC Report. https://www.pwc.com/gx/en/issues/analytics/assets/pwc-ai-analysis-sizing-the-prize-report.pdf

[13] Bin Li and Steven C.H. Hoi. 2012. *Online portfolio selection: A survey*. ACM Computing Surveys 46. https://doi.org/10.1145/2512962

[14] Xinbo Ding, Shanwei Wu, and Zhiquan Luo. 2019. *Hybrid deep reinforcement learning with genetic algorithm for stock trading.* IEEE Access 7. https://doi.org/10.1109/ACCESS.2019.2957005

[15] Dong Zeng, Zhiqiang Wu, and Ming Li. 2023. *A hyperparameter adaptive genetic algorithm based on deep Q-learning network*. Journal of Circuits, Systems, and Computers. https://doi.org/10.1142/S0218126623502258

[16] Cong Ma, Jiangshe Zhang, Junmin Liu, Lizhen Ji, and Fei Gao. 2021. *A parallel multi-module deep reinforcement learning algorithm for stock trading*. Neurocomputing. https://www.sciencedirect.com/science/article/pii/S0925231221005233

[17] Zhengyao Jiang, Dixing Xu, and Jinjun Liang. 2017. *A deep reinforcement learning framework for the financial portfolio management problem.* arXiv:1706.10059. https://arxiv.org/abs/1706.10059

[18] Da Woon Jeong and Yeong Hyeon Gu. 2024. *Pro Trader RL: Reinforcement learning framework for generating trading knowledge by mimicking the decision-making patterns of professional traders.* Expert Systems with Applications 124465. https://doi.org/10.1016/j.eswa.2024.124465

[19] Xiao-Yang Liu, Hongyang Yang, Qian Chen, Runjia Zhang, Liuqing Yang, Bowen Xiao, and Christina Dan Wang. 2020. *FinRL: A deep reinforcement learning library for automated stock trading in quantitative finance*. arXiv:2011.09607. https://arxiv.org/abs/2011.09607

[20] AI4Finance-Foundation. *FinRL GitHub repository.* https://github.com/AI4Finance-Foundation/FinRL

Development

yFinance: https://github.com/ranaroussi/yfinance