Reinforcement Learning (RL) in Finance

Fulvio Raddi, Armin Sadighi, Jens Reil

June 11, 2025

1 Introduction

In financial markets, portfolio optimization related to the modern portfolio theory (MPT) aims to allocate assets efficiently to maximize returns while managing risk [1]. Traditional portfolio management techniques, such as mean-variance optimization [2] and dynamic programming approaches, rely on predefined assumptions about asset distributions and investor preferences. However, these methods struggle with the complexity and non-stationarity of real-world financial data.

Reinforcement learning (RL), a branch of machine learning focused on decision-making in dynamic environments, has emerged as a powerful tool for portfolio optimization. Unlike conventional models, RL can adaptively learn trading strategies by interacting with the market, updating its policies based on observed rewards, and optimizing actions in an uncertain and evolving financial landscape. By framing portfolio management as a sequential decision-making problem, RL enables the development of self-learning agents that adjust asset allocations in response to changing market conditions.

Despite the promise of RL in financial applications, several challenges hinder its adoption in portfolio optimization. Key issues include handling noisy and non-stationary financial data, defining effective reward functions that balance risk and return, ensuring interpretability of learned strategies, and addressing computational efficiency in real-world trading scenarios. The lack of standardized benchmarks and robust evaluation metrics further complicates the assessment of RL-based strategies. This research aims to develop a reinforcement learning framework for portfolio optimization that effectively learns adaptive investment strategies while addressing key financial constraints. By implementing and evaluating different RL algorithms, we seek to enhance the applicability of RL in real-world portfolio management. The main research question for this thesis is:

"How can reinforcement learning be effectively applied to portfolio optimization to achieve adaptive, risk-aware investment strategies?"

To comprehensively address the main research question, this thesis will explore several subquestions that highlight ..., which are mentioned below.

1. What are the most effective RL algorithms (e.g., **fill in our potential exploration areas for RL**) for portfolio optimization in financial markets?
2. How do **(mention here various constraints we are putting into place)** impact RL-based portfolio strategies, and how can they be integrated into the learning process?

By answering these questions, this research will contribute to developing a more robust and practical RL-based portfolio optimization framework, bridging the gap between theoretical advancements in machine learning and real-world financial applications.

2 Research Context

Reinforcement Learning (RL) has gained significant traction in quantitative finance as a means to automate trading strategies, optimize hedging, and enhance portfolio management. As financial markets exhibit non-stationary behavior, with price dynamics changing over time, traditional models struggle to adapt to these fluctuations, often leading to suboptimal investment decisions [3]. RL, however, provides the ability to continuously learn from interactions with the market, updating policies dynamically to optimize long-term returns [4]. A key advantage of RL over static optimization models is its capacity to balance exploration and exploitation, dynamically adjusting portfolio allocations based on real-time market conditions [5].

Policy gradient methods, as introduced by Sutton et al. [6], offer a powerful framework for continuous action spaces in financial decision-making. These methods optimize policies directly by estimating gradients of expected rewards, making them well-suited for portfolio allocation and risk management. Actor-critic methods such as Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A2C) have been applied in stock trading and high-frequency strategies, improving adaptability in dynamic environments. However, a key limitation remains the sample inefficiency of these methods, where high variance in gradient estimates can slow convergence.

Kakade & Langford published a NeurIPS paper on policy gradients [7] which introduces key concepts like Direct Policy Optimization. Unlike value-based RL algorithms, which estimate action-value functions and rely on discrete decision-making frameworks, policy gradient methods optimize a parametric policy directly. This direct policy optimization approach allows for smooth and adaptive decision-making, making it particularly useful for portfolio rebalancing, where traders must continuously adjust asset allocations rather than making rigid buy/sell decisions. In environments where financial instruments require precise position sizing and risk management, policy gradients offer a more flexible and dynamic alternative to discrete-action strategies.

One of the most compelling advantages of policy gradient methods in finance is their ability to handle complex trading environments where simple

buy/sell/hold signals are insufficient. Portfolio management requires continuous adjustments to asset weightings based on real-time market conditions, volatility, and risk factors. Traditional financial models often struggle with these dynamic adjustments, whereas policy gradient methods naturally accommodate fluid portfolio allocation strategies, allowing RL agents to refine their investment decisions incrementally rather than making abrupt, predefined shifts.

A significant limitation of policy gradient methods is high variance in gradient estimates, leading to slow learning and unstable convergence. This issue is especially problematic in financial applications, where data sparsity and market noise can disrupt training efficiency. These challenges align with findings from Szepesvári's work [8] on RL algorithms in Markov Decision Processes (MDPs) (Szepesvári, 2010), which highlights the importance of sample efficiency and variance reduction techniques in practical RL applications. To mitigate instability, actor-critic methods such as Proximal Policy Optimization (PPO) and Advantage Actor-Critic (A2C) have been applied to stock trading and high-frequency strategies, improving adaptability in dynamic markets.

One notable contribution in this field is FinRL, an open-source DRL framework designed for trading automation. It employs various market datasets, including historical stock prices from Yahoo Finance, cryptocurrency market data, and live trading APIs such as Alpaca and Interactive Brokers [9]. The framework consists of a structured three-layer architecture: an environment layer that integrates historical and live data with OpenAI Gym-style trading simulators, an agent layer that implements DRL algorithms such as DQN, PPO, A2C, TD3, and SAC, and an application layer for backtesting and evaluation [9]. FinRL significantly lowers the entry barrier for researchers and practitioners by providing a full-stack solution for DRL-based trading strategies. However, despite its modularity and accessibility, FinRL's reliance on historical data raises concerns about its ability to generalize to real-world trading conditions, where market dynamics and adversarial behaviors can drastically impact model performance [9].

Another influential study is the QLBS model, which integrates Q-learning with the Black-Scholes model to enhance option pricing. Unlike FinRL, which focuses on general trading automation, QLBS applies a more structured approach, utilizing Fitted Q Iteration to approximate the value function of an option portfolio [10]. The model incorporates inverse reinforcement learning (IRL) to estimate unknown reward functions, addressing market imperfections such as volatility smiles [10]. While the QLBS model effectively bridges traditional financial models with DRL, it operates under the assumption of a structured Markov Decision Process (MDP) for option pricing, which may fail to capture real-world complexities like liquidity constraints and transaction costs [10].

A broader study on DRL applications for stock trading and portfolio management explores how policy-based and actor-critic methods, such as PPO and A3C, can optimize asset allocation and high-frequency trading strategies [11]. This study highlights DRL's adaptability in dynamic market conditions, particularly through risk-adjusted reward functions that enhance stability in trading decisions [11]. However, the study also acknowledges the challenge of backtesting realism, as simulated environments often fail to account for execution

latencies, market slippage, and sudden regime shifts, limiting their real-world applicability [11].

A key challenge in RL-based portfolio optimization is the lack of domain-specific knowledge [4]. RL agents, when trained solely on historical data, struggle with market unpredictability, as structural shifts can render learned policies ineffective [4]. To address this, MetaTrader introduces a novel two-phase learning approach that integrates expert trading strategies into the RL framework [3]. In the first phase, the agent learns a diverse set of policies by incorporating imitation learning objectives alongside traditional RL rewards, allowing it to develop an understanding of different trading styles. In the second phase, a meta-policy is trained to select the most appropriate learned policy for a given market environment, improving interpretability and robustness [3]. Similarly, dynamic portfolio rebalancing leverages predictive modeling to guide RL agents toward more informed asset allocation decisions. By integrating Long Short-Term Memory (LSTM) models, RL agents gain a forward-looking perspective, enabling them to anticipate market shifts rather than merely reacting to price movements [5].

Risk management plays a crucial role in portfolio optimization, and RL presents new opportunities for dynamic rebalancing. Unlike traditional strategies that operate on fixed schedules, RL can dynamically adjust asset allocations in response to changing market conditions [5]. Research suggests that gradual rebalancing, where RL agents incrementally adjust allocations rather than making abrupt shifts, leads to reduced transaction costs and minimizes exposure to sudden market fluctuations [5]. This aligns with MetaTrader's approach, where agents transition between different policies rather than committing to a single trading strategy [3]. Moreover, RL-based methods incorporate risk-sensitive reward functions to balance returns and volatility, such as differential Sharpe ratio rewards, which optimize for risk-adjusted returns rather than absolute profits [4]. By penalizing excessive volatility, RL agents learn to make more stable investment decisions, reducing the likelihood of catastrophic losses in bear markets [4].

While model-based approaches attempt to forecast price movements using predictive models, recent research suggests that model-free RL methods offer superior generalization capabilities [4]. Unlike traditional models that rely on predefined assumptions, model-free RL agents dynamically adapt to unseen market conditions [4]. Studies on RL-based portfolio management demonstrate that Deep Q-Networks (DQN) and Policy Gradient methods outperform model-based approaches in asset allocation tasks, achieving higher Sharpe Ratios and cumulative returns [4]. Additionally, data augmentation techniques play a critical role in enhancing RL performance. Since financial data is often sparse and noisy, synthetic data generation stabilizes training, while generative modeling and pre-training on simulated environments help RL agents better navigate the complexities of real-world markets [4].

Overall, these studies illustrate the growing applicability of RL in finance, showcasing its potential in trading automation, option pricing, and portfolio optimization. However, several challenges remain, particularly in bridging the

gap between simulation and real-world execution, mitigating overfitting to historical data, and improving adaptation to market uncertainties. As research progresses, the integration of expert knowledge, imitation learning, predictive modeling, and risk-sensitive reward functions will be crucial in enhancing the robustness and reliability of RL-based financial strategies.

3 Methodology

3.1 CRISP-DM

This report handles a ML problem (RL problem). A standard approach is needed which translates business problems into ML tasks, suggest appropriate data transformations and data mining techniques, and provide means for evaluating the effectiveness of the results and documenting the experience. The CRISP-DM (*CR*oss *I*ndustry *S*tandard *P*rocess for *D*ata *M*ining) project defines a process model which provides a framework for carrying out data mining projects which is independent of both the industry sector and the technology used [12]. The CRISP-DM process model aims to make large data mining projects less costly, more reliable, repeatable, manageable, and faster. CRISP-DM is still a de-factor standard in data mining [13]. It can also be applied to ML problems.

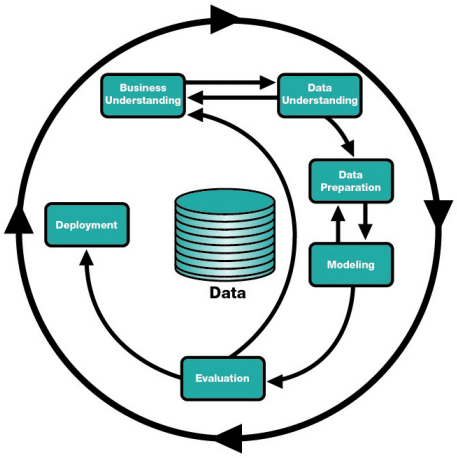


Figure 1: CRISP-DM, a framework for data mining projects [12].

The CRISP-DM framework is a widely recognized methodology that provides a structured approach to planning and executing data mining projects [13]. It comprises six major phases: *Business Understanding Data Understanding*, *Data Preparation*, *Modeling*, *Evaluation*, and *Deployment*. Each phase is crucial for the success of a data mining project, and they are typically executed in sequence, although iterations and loops between phases are common to refine processes

and improve outcomes [12]. The following is a general description of each phase, based on the CRIPS-DM guide by Schröer et al [13]:

1. *Business Understanding.* Identify the business problem, define data mining goals aligned with business objectives, and establish success criteria. Develop a detailed project plan.
2. *Data Understanding.* Collect data from various sources, perform exploratory analysis to understand its attributes and quality, and identify any initial insights or issues.
3. *Data Preparation.* Cleanse the data, select relevant subsets, derive new variables as needed, and transform the data into a final dataset ready for modeling.
4. *Modeling.* Choose appropriate modeling techniques based on the business problem and data type, configure model parameters, build and assess model performance.
5. *Evaluation.* Assess the model against business objectives, interpret results, determine business impact, and decide on the next steps. Review the process for improvements.
6. *Deployment.* Implement the data mining solution in a business setting, which may range from generating reports to operationalizing models. Plan for monitoring and maintenance to ensure ongoing effectiveness.

This report focuses on the CRISP-DM methodology, which outlines the most essential steps in the ML process. However, it is crucial to highlight the existing literature on CRISP-ML, which includes an extra monitoring step as monitoring will play a significant role in maintaining ML models after deployment. For this report focused on research, this was not necessary to include.

3.2 Reinforcement Learning Techniques

3.2.1 Markov Decision Process (MDP)

A Markov Decision Process (MDP) provides a mathematical framework for sequential decision-making under uncertainty. It is formally defined by a tuple (S, A, P, R, γ), where:

- S is the state space, representing all possible environments the agent can encounter.
- A is the action space, consisting of all possible actions an agent can take.
- $P(s'|s, a)$ is the transition probability function, defining the probability of transitioning to state s' given the current state s and action a.
- $R(s, a)$ is the reward function, providing a scalar feedback signal for taking action a in state s.
- $\gamma \in [0, 1]$ is the discount factor, controlling the importance of future rewards.

The goal in an MDP is to find an optimal policy π^* that maximizes the expected cumulative reward:

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t)\right]. \tag{1}$$

The value function $V^\pi(s)$ represents the expected return starting from state s while following policy π:

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \mid s_0 = s\right]. \tag{2}$$

Similarly, the action-value function $Q^\pi(s, a)$ defines the expected return when taking action a in state s and then following policy π:

$$Q^\pi(s, a) = \mathbb{E}\left[R(s, a) + \gamma V^\pi(s') \mid s, a\right]. \tag{3}$$

The Bellman optimality equation provides a recursive relationship for the optimal value function:

$$V^*(s) = \max_{a \in A}\left[R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^*(s')\right]. \tag{4}$$

Solving an MDP involves finding π^* using methods such as **Dynamic Programming (e.g., Value Iteration, Policy Iteration)**, **Monte Carlo Methods**, and **Temporal Difference Learning (e.g., Q-learning, SARSA)**. In financial applications, MDPs model portfolio optimization, trading strategies, and risk management under uncertainty. SARSA, due to its on-policy nature, is particularly useful in settings where risk-sensitive strategies are crucial.

3.2.2 SARSA: On-Policy Temporal Difference Learning

SARSA (State-Action-Reward-State-Action) is an on-policy reinforcement learning algorithm that estimates the action-value function $Q(s, a)$ while following the current policy. Unlike Q-learning, which is an off-policy method, SARSA updates its action-value estimates using the actions actually taken by the agent.

The SARSA update rule is given by:

$$Q(s, a) \leftarrow Q(s, a) + \alpha\left[R + \gamma Q(s', a') - Q(s, a)\right], \tag{5}$$

where:

- s, a are the current state and action,
- R is the received reward,
- s', a' are the next state and action,
- α is the learning rate, and

- γ is the discount factor.

Since SARSA is on-policy, it updates its Q-values based on the action taken according to the current policy rather than using the greedy action (as in Q-learning). This characteristic allows SARSA to incorporate risk-aware strategies, making it particularly useful in financial applications where conservative strategies are preferred to avoid excessive losses.

3.2.3 Deep Q-Learning

Q-learning is a type of reinforcement learning algorithm used to help an agent learn how to make decisions by interacting with an environment. The main goal of Q-learning is to learn the optimal action-selection policy that maximizes the cumulative reward over time. It does this by estimating a function called the Q-function, which represents the expected return (or future reward) of taking a certain action in a given state and following the optimal policy afterward.

Q-learning is model-free, meaning it does not require knowledge of the environment's dynamics. Instead, it updates its Q-values based on the feedback it receives from the environment after taking actions. Over time, and with enough exploration, Q-learning can converge to the optimal policy.

In the context of portfolio management, Q-learning can be used to learn optimal trading strategies. Here, the environment is the financial market, the state can include market indicators and portfolio holdings, and the actions are the possible ways of allocating funds across different assets. The reward is typically related to the change in portfolio value.

While traditional Q-learning works well in environments with small, discrete state and action spaces, it struggles when the state space is large or continuous, as is often the case in financial markets. To address this limitation, Deep Q-Learning (DQN) uses a deep neural network to approximate the Q-function instead of using a table.

DQN allows the agent to generalize across similar states and learn complex patterns from high-dimensional inputs, such as multiple financial indicators. In this approach, the neural network takes a feature vector representing the current state of the market and outputs Q-values for each possible action. The agent then selects actions based on these Q-values using an exploration strategy (e.g., ϵ-greedy).

In the context of portfolio management, Deep Q-Learning enables the creation of intelligent agents that can handle noisy, high-dimensional market data and learn dynamic strategies for allocating funds. These strategies can adapt over time and potentially outperform traditional fixed-rule or heuristic-based methods. Techniques like experience replay, target networks, and reward shaping are often used to stabilize training and improve performance.

Overall, Deep Q-Learning offers a promising and flexible approach to data-driven financial decision-making.

To formalize these methods mathematically, we can say that Q-learning aims to learn the optimal *action-value function* $Q^*(s, a)$, which represents the maxi-

mum expected return achievable from state s by taking action a and following the optimal policy thereafter:

$$Q^*(s_t, a_t) = \mathbb{E}\left[r_t + \gamma \max_{a'} Q^*(s_{t+1}, a')\right]$$

The goal of Deep Q-Learning is to approximate the optimal Q-function, which represents the expected cumulative reward for taking action a in state s and following the optimal policy thereafter. This is based on the Bellman optimality equation:

$$Q^*(s, a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*(s', a') \,\middle|\, s, a\right] \tag{6}$$

Here:

- s and s' are the current and next states
- a and a' are the current and next actions
- r is the reward received after taking action a in state s
- γ is the discount factor
- $\mathbb{E}_{s'}$ denotes the expectation over the next state

In Deep Q-Learning, a neural network with parameters θ is used to approximate the Q-function, i.e., $Q(s, a; \theta)$. The target Q-value is computed using a separate target network with parameters θ^-:

$$y = r + \gamma \max_{a'} Q(s', a'; \theta^-) \tag{7}$$

The loss function used to update the Q-network is usually the **Huber loss** (also known as Smooth L1 Loss), which is more stable than the mean squared error:

$$\mathcal{L}(\theta) = \mathbb{E}_{(s,a,r,s') \sim \mathcal{D}}\left[\text{Huber}\left(Q(s, a; \theta) - y\right)\right] \tag{8}$$

Where $\mathcal{D}$ is the experience replay buffer. The Huber loss is defined as:

$$\text{Huber}(x) = \begin{cases} \frac{1}{2}x^2 & \text{if } |x| < 1 \\ |x| - \frac{1}{2} & \text{otherwise} \end{cases} \tag{9}$$

The target network parameters θ^- are updated periodically with the parameters of the main Q-network to stabilize training:

$$\theta^- \leftarrow \theta \tag{10}$$

3.2.4 Advantage Actor-Critic (A2C)

Advantage Actor-Critic (A2C) is a RL that integrates both policy-based and value-based methods, improving training stability and efficiency. It is an extension of the standard actor-critic method, where a policy function (actor) and a value function (critic) work together to optimize decision-making in an environment [14].

In A2C, the policy is represented by an actor $\pi_\theta(a \mid s)$, which selects actions based on the current state s, while the critic estimates the state-value function $V^\pi(s)$, which represents the expected future rewards. Instead of using raw rewards, A2C employs an advantage function to reduce variance in gradient updates:

$$A(s,a) = Q(s,a) - V(s), \tag{11}$$

where $Q(s,a)$ is the action-value function, and $V(s)$ is the state-value function. Since $Q(s,a)$ is often unavailable, it is approximated using a bootstrapped estimate from temporal difference learning:

$$A(s,a) \approx r + \gamma V(s') - V(s), \tag{12}$$

where r is the immediate reward and γ is the discount factor. The policy is updated using the advantage-weighted gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\nabla_\theta \log \pi_\theta(a \mid s) A(s,a)\right], \tag{13}$$

which ensures that actions leading to higher-than-expected returns are reinforced, while suboptimal actions are discouraged. Simultaneously, the critic is trained to minimize the mean squared error between its value estimate and the target return:

$$L_V = \mathbb{E}\left[(r + \gamma V(s') - V(s))^2\right]. \tag{14}$$

Unlike Asynchronous Advantage Actor-Critic (A3C), which updates asynchronously **<empty citation>** A2C synchronizes updates across multiple environments in parallel. This improves training stability and efficiency by reducing variance in gradient estimates [15]. A2C is widely used for continuous control tasks and financial applications, such as portfolio optimization. Its combination of policy optimization and value estimation makes it a stable and efficient RL algorithm.

3.3 Validation Methods

3.3.1 Evaluation Metrics

Evaluating RL models in portfolio optimization requires metrics that assess profitability, risk management, and generalization. Unlike supervised learning, where accuracy and loss functions dominate, RL evaluation focuses on financial

performance indicators. The most relevant metrics include cumulative reward, Sharpe ratio, maximum drawdown, and alpha, among others [16] [17].

One of the fundamental measures is the *cumulative reward*, which represents the total return an agent accumulates over an episode:

$$G_t = \sum_{k=0}^{T} \gamma^k r_{t+k} \tag{15}$$

where r_t is the reward at time t, γ is the discount factor, and T is the episode length. Since raw cumulative rewards can be sensitive to scaling, risk-adjusted performance metrics such as the *Sharpe ratio* are more informative:

$$SR = \frac{\mathbb{E}[R_p - R_f]}{\sigma_p} \tag{16}$$

where R_p is the portfolio return, R_f is the risk-free rate, and σ_p is the standard deviation of returns. A higher Sharpe ratio indicates better risk-adjusted returns. For a more refined view of downside risk, the *Sortino ratio* replaces σ_p with the standard deviation of negative returns, making it particularly useful in financial applications.

Another critical risk metric is the *maximum drawdown (MDD)*, which measures the worst peak-to-trough decline in portfolio value:

$$MDD = \max_{t \in [0,T]} \left(\frac{V_{\max} - V_t}{V_{\max}} \right) \tag{17}$$

where $V_{\max}$ is the highest portfolio value observed up to time t, and V_t is the portfolio value at time t. A lower MDD indicates better capital preservation.

To compare performance against a benchmark, the *alpha* metric quantifies excess returns over the market:

$$\alpha = R_p - \beta R_m \tag{18}$$

where R_m is the market return and β measures sensitivity to market movements. Positive alpha suggests outperformance, while beta indicates risk exposure.

In addition to financial returns, an RL model's *profit-to-loss ratio (P/L Ratio)* is often used to assess trading efficiency:

$$PLR = \frac{\text{Total Profit Trades}}{\text{Total Loss Trades}} \tag{19}$$

A higher PLR signifies a strategy that generates more profitable trades than losing ones. Finally, the model's *time to convergence* is crucial in assessing training efficiency, ensuring that policy learning stabilizes within a reasonable number of episodes.

For reinforcement learning models in portfolio optimization, a combination of financial and RL-specific metrics is necessary to ensure robust evaluation. The Sharpe ratio and maximum drawdown capture risk-adjusted performance, while

alpha and the profit-to-loss ratio measure profitability relative to benchmarks. These metrics provide a comprehensive view of model performance beyond simple reward accumulation.

4 Experimental Setup

4.1 Approach

In this study, we focus on constructing and evaluating a reinforcement learning-based portfolio strategy using historical price data from three major companies of the S&P 500: Apple (AAPL), Microsoft (MSFT), and Alphabet (GOOGL). The dataset spans from January 1, 2020, to January 1, 2025, and is retrieved using the `yfinance` Python package.

To validate the stochastic behavior of the financial time series, we use the Ornstein–Uhlenbeck (OU) process as a baseline mean-reverting model. This is a well-known continuous-time stochastic process often used in financial mathematics to model interest rates, volatility, and other mean-reverting phenomena. We simulate the OU process in two configurations: one where the process starts at the long-term mean, and another where it starts away from the mean. This dual setup allows us to illustrate the impact of initial conditions on mean reversion behavior. The generated figures provide visual insights into how quickly the process returns to its mean under different initial conditions.

Our primary objective is to train a reinforcement learning agent to manage a portfolio of the three selected stocks. The agent's performance is evaluated by comparing the cumulative portfolio value over time against a benchmark equal-weight portfolio. This benchmark allocates equal proportions of capital to each asset and is rebalanced periodically. Such a comparison allows us to assess the added value of dynamic allocation strategies over naive diversification.

The coding environment used for this study is Jupyter Notebook[1]. This choice was made primarily due to the advantages Jupyter Notebook offers for Python programming, which is the preferred language for this study. Additionally, research conducted on Kaggle[2] for similar financial problems indicates that notebooks are widely used as a preferred tool for both coding and easy visualization. As mentioned, Jupyter Notebook allows for an easy and interactive way of coding, where code can be written and executed in small, manageable blocks. This is particularly beneficial for debugging, as it allows for real-time testing and iteration on individual pieces of code. Additionally, Jupyter Notebook allows for comprehensive documentation alongside the code, which enhances readability and understanding.

[1] The Jupyter Notebook is the original web application for creating and sharing computational documents. It offers a simple, streamlined, document-centric experience **Jupyter˙2024**.

[2] Kaggle allows users to find datasets, publish their own, collaborate with other data scientists and machine learning engineers, and participate in competitions to solve data science challenges **Kaggle˙2022**.

4.2 Data Understanding

In this project, we work with historical financial data from the S&P 500 index. The data spans from January 1, 2020, to January 1, 2025, and is retrieved from Yahoo Finance. The S&P 500 is a widely followed stock market index that includes 500 of the largest publicly traded companies in the United States, making it a representative benchmark for the overall performance of the U.S. equity market.

The dataset contains daily price information, including open, high, low, close, adjusted close, and trading volume. This time series data serves as the foundation for our analysis and modeling tasks, enabling us to explore market dynamics, test trading strategies, and evaluate the performance of reinforcement learning algorithms in a realistic financial environment.

4.3 Data Preparation

For the data preparation stage, we focused on retrieving and transforming historical stock price data for a subset of S&P 500 companies specifically[3].

We first extracted the daily closing prices for the selected tickers and removed any rows containing missing values to ensure data consistency. The closing prices were then transformed into cumulative returns. This was achieved by computing the daily percentage change, adding one to each value, and taking the cumulative product over time. This transformation reflects the compounded performance of each asset over the time horizon.

Additionally, we computed daily log returns from the cumulative returns to use as inputs for our modeling tasks. These return series are commonly used in financial analysis due to their statistical properties, such as approximate normality and time-additivity.

This preprocessing ensures that the input data is clean, well-structured, and aligned with standard practices in quantitative finance, enabling more robust modeling and evaluation.

4.4 Modeling

In this study, we apply RL techniques to construct adaptive portfolio strategies. Our focus lies on three key RL algorithms: Deep Q-Learning (DQN), Advantage Actor-Critic (A2C), and State-Action-Reward-State-Action (SARSA). Each method offers a different perspective on how to learn optimal asset allocation policies based on historical financial data.

4.5 Model Evaluation

The evaluation of reinforcement learning (RL) models in the context of portfolio optimization requires a multifaceted approach that accounts for both profitability and risk. To this end, we employ a suite of financial performance metrics that

[3]We used Apple (AAPL), Microsoft (MSFT), and Alphabet (GOOGL).

provide a comprehensive view of how well each model performs under realistic trading conditions.

The primary measure of success is the *cumulative reward*, which reflects the total return an agent accumulates over the duration of a trading episode. While cumulative rewards offer a raw indicator of performance, they do not consider the variability or risk associated with the returns. To address this limitation, we incorporate the *Sharpe ratio*, a widely used metric for risk-adjusted returns. A higher Sharpe ratio implies that the model is able to generate higher returns per unit of risk. We also consider the *Sortino ratio*, which focuses specifically on downside risk by only accounting for negative return volatility. This makes it particularly useful in evaluating models where capital preservation is critical.

By using this diverse set of evaluation criteria—spanning raw performance, risk-adjusted returns, downside risk, market-relative performance, trade quality, and training dynamics—we are able to conduct a rigorous and balanced assessment of our RL models. This ensures that model selection is guided not only by profitability, but also by robustness and real-world applicability.

5 Results

5.1 SARSA: On-Policy Temporal Difference Learning

We investigate whether a SARSA-based reinforcement learning agent can learn to approximate the long-term mean of an Ornstein–Uhlenbeck (OU) process. The OU process is defined by the following stochastic differential equation:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t,$$

where $\theta > 0$ is the rate of mean reversion, μ is the long-term mean (set to 0), σ is the volatility, and dW_t is a Wiener process increment. The process exhibits mean-reverting behaviour and is commonly used in financial modelling and physical systems.

To simulate this process in a reinforcement learning framework, we discretize time with $\Delta t = 1$, and approximate the continuous dynamics using the Euler–Maruyama method:

$$x_{t+1} = x_t + \theta(\mu - x_t)\Delta t + \sigma\sqrt{\Delta t} \cdot \varepsilon_t, \tag{20}$$

where $\varepsilon_t \sim \mathcal{N}(0, 1)$. The state space is defined by discretizing the real-valued variable x_t into N bins over a bounded interval, e.g., $[-10, 10]$. The index of the bin in which x_t falls defines the discrete state s_t used by the agent.

We implement a tabular SARSA agent with the following specifications:

- **States (S):** Discretized positions of x_t over $N = 100$ bins.
- **Actions (A):** $\{-1, 0, +1\}$, representing small corrective impulses to the left, none, or to the right, respectively.
- **Policy:** ϵ-greedy with $\epsilon = 0.1$.

- **Reward:** At each time step, the agent receives $r_t = -|x_t|$, penalizing distance from the mean.
- **Update rule (SARSA):**

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right], \tag{21}$$

  where $\alpha = 0.1$ is the learning rate and $\gamma = 0.99$ is the discount factor.

We train the agent for 500 episodes, each consisting of 100 steps. At the beginning of each episode, the process is initialized at $x_0 = 5.0$, far from the mean $\mu = 0$, to test the agent's ability to learn mean-reverting behaviour from a non-equilibrium state. At each time step, the agent selects an action, applies the corresponding perturbation to x_t, and then observes the new state following the OU dynamics. The reward signal encourages behaviour that minimizes deviation from the mean.

Figure 2 illustrates the trajectory x_t of the Ornstein–Uhlenbeck process over 100 discrete time steps during the final episode of SARSA training. The process is initialized at $x_0 = 5.0$, a value significantly distant from the long-term mean $\mu = 0$, in order to assess the agent's ability to learn a mean-reverting policy.

In the initial phase (approximately $t = 0$ to $t = 15$), the trajectory exhibits a rapid decline toward the mean. This indicates that the agent has successfully learned to apply control actions that efficiently reduce the deviation from equilibrium. The steep descent demonstrates the effectiveness of the learned policy in exploiting the mean-reverting nature of the process.

For $t > 15$, the trajectory stabilizes and oscillates with small amplitude around the mean. These bounded fluctuations reflect the intrinsic stochasticity of the Ornstein–Uhlenbeck dynamics, modulated by the ϵ-greedy exploration strategy of the SARSA agent. Importantly, the system remains consistently close to zero, suggesting that the agent's policy is both stable and robust under noise.

These results confirm that the SARSA algorithm is capable of learning an effective control policy for a continuous-time, mean-reverting stochastic process. The agent not only steers the system toward the equilibrium point but also sustains proximity to the mean in the presence of randomness.

Now we investigate the application of the SARSA reinforcement learning algorithm for dynamic portfolio optimization. The objective is to evaluate whether a SARSA agent can learn to allocate capital among a set of assets in order to maximize the risk-adjusted return, as measured by the Sharpe ratio. We use adjusted closing prices of three prominent stocks: Apple Inc. (AAPL), Microsoft Corp. (MSFT), and Alphabet Inc. (GOOGL), retrieved via the `yfinance` API. The data spans the period from January 1, 2020, to January 1, 2025. The daily log returns are computed as:

$$r_t^{(i)} = \log\left(\frac{P_t^{(i)}}{P_{t-1}^{(i)}}\right),$$

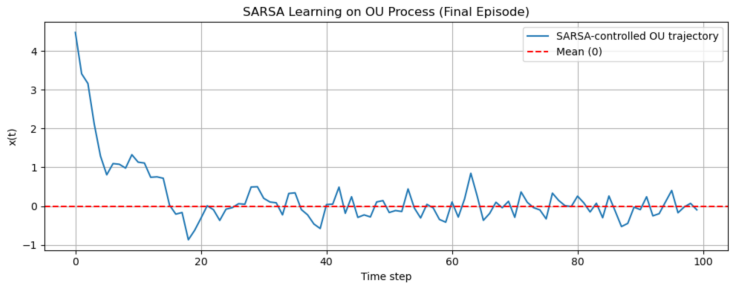


Figure 2: Trajectory of x_t in the final episode of SARSA training on an Ornstein–Uhlenbeck process. The agent starts from $x_0 = 5.0$ and learns to remain close to the mean $\mu = 0$.

where $P_t^{(i)}$ denotes the price of asset i at time t.

We formulate the portfolio optimization problem as a finite Markov Decision Process (MDP) with the following components:

- **State (s_t):** The state at time t consists of the current allocation weights and recent returns for all assets. The state space is discretized to enable tabular SARSA learning.
- **Actions (a_t):** The agent can adjust the portfolio weights by selecting from a discrete set of allocation strategies. Each action corresponds to a reallocation vector across the assets, constrained such that weights sum to 1.
- **Reward (r_t):** At each time step, the reward is the portfolio return over the next period, defined as:

$$r_t = \sum_{i=1}^{N} w_t^{(i)} r_{t+1}^{(i)}, \tag{22}$$

  where $w_t^{(i)}$ is the weight of asset i in the portfolio at time t.
- **Policy:** An ϵ-greedy strategy is employed, balancing exploration and exploitation. The exploration rate ϵ decays over time.
- **Update Rule (SARSA):** The Q-values are updated using:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right], \tag{23}$$

  where α is the learning rate and γ is the discount factor.

The SARSA agent is trained on daily returns with a rolling window and backtested across the entire time horizon. After learning, the portfolio value is computed using cumulative returns:

$$V_t = V_{t-1}(1 + r_t),$$

starting from an initial value $V_0 = 1$.

To evaluate performance, we compute the Sharpe ratio:

$$\text{Sharpe Ratio} = \frac{\mathbb{E}[r_t]}{\sigma(r_t)},$$

where $\mathbb{E}[r_t]$ and $\sigma(r_t)$ denote the mean and standard deviation of the portfolio's daily returns, respectively.

An equal-weight benchmark portfolio is constructed by assigning fixed weights of $1/3$ to each of the three assets throughout the period. The Sharpe ratio for both strategies is reported below.

Table 1: Sharpe Ratios of Portfolio Strategies (2020–2025).

Strategy	**Sharpe Ratio**
SARSA-Optimized Portfolio	0.9368
Equal-Weight Portfolio	0.9162

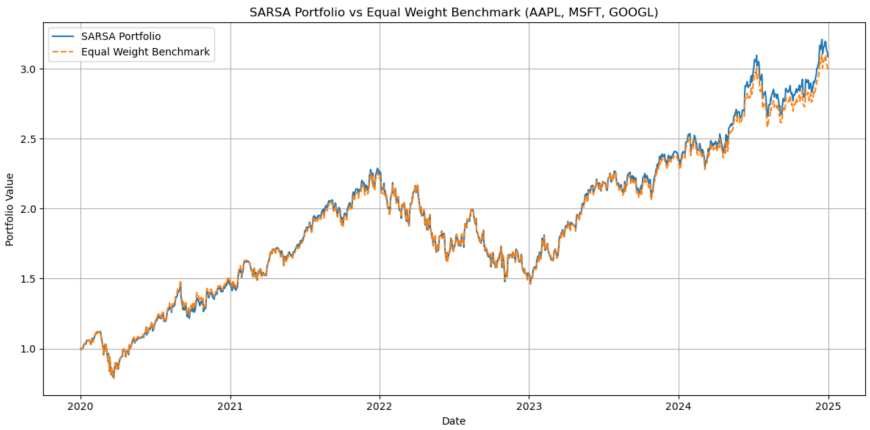


Figure 3: Comparison of the cumulative portfolio values for a SARSA-based dynamic allocation strategy and an equal-weight benchmark portfolio over the period from January 1, 2020, to January 1, 2025. Both portfolios consist of the same three assets: AAPL, MSFT, and GOOGL.

Figure 3 illustrates the evolution of portfolio value over a five-year investment horizon using two different strategies. The first strategy employs the SARSA reinforcement learning algorithm to dynamically adjust the portfolio weights of three stocks: AAPL, MSFT, and GOOGL. The second, used as a benchmark, maintains a static allocation with equal weights of $1/3$ per asset.

The SARSA agent was trained to maximize a reward function that incorporates both returns and volatility penalization, with the goal of improving the risk-adjusted performance of the portfolio. States were defined based on a discretized representation of recent return trends, while actions corresponded to discrete portfolio allocations. At each time step, the SARSA agent updated its Q-values using observed returns and selected the next action via an ϵ-greedy policy, balancing exploration and exploitation.

As shown in the plot, both portfolios track closely in value over time, indicating that the SARSA agent does not deviate significantly from the benchmark under normal market conditions. However, during specific market phases—particularly around bullish momentum or recovery from drawdowns—the SARSA-controlled portfolio exhibits slightly higher peaks. This reflects the adaptive nature of the learned policy, which reallocates exposure to more favorable assets as estimated by the agent's value function.

Quantitatively, the final Sharpe ratio of the SARSA portfolio was computed as 0.9368, slightly outperforming the equal-weight benchmark's Sharpe ratio of 0.9162. Although the improvement is modest, it highlights the capability of the reinforcement learning agent to adjust allocation in a way that enhances reward relative to risk.

These results suggest that reinforcement learning methods such as SARSA, even in their tabular and discretized form, can be used to derive competitive portfolio strategies that slightly outperform traditional heuristic benchmarks over extended periods in terms of risk-adjusted returns.

5.2 Deep Q-Learning

The dataset is constructed using historical price data for a set of financial assets namely Apple, Microsoft, and Google retrieved via the `yfinance` API. Daily data spanning several years is collected to ensure sufficient variability for training and evaluation.

To represent the state of the market at each timestep, the code computes a range of financial indicators:

- The 20-day moving average helps capture short-term trends.
- The Relative Strength Index (RSI) indicates potential overbought or oversold conditions.
- The Moving Average Convergence Divergence (MACD) provides momentum signals.
- Short-term (1-day) and medium-term (5-day) returns quantify recent performance.
- A 21-day rolling standard deviation is used to estimate volatility.

These features are normalized and concatenated into a single feature vector, which serves as the input to the Deep Q-Network. This representation allows

the agent to learn from both price dynamics and technical signals. The size of the features is 21, so for each asset we have 7 additional computed market indicators.

A custom environment simulates the portfolio management process. At each timestep, the environment receives an action from the agent in the form of a portfolio allocation across the available assets. The environment then computes the resulting change in portfolio value based on real asset returns and updates the agent's state accordingly.

To guide learning, three distinct reward functions are implemented:

1. **Raw Return Reward:** Directly reflects the portfolio's day-to-day return.

2. **Sharpe-like Reward:** A risk-adjusted reward calculated by subtracting a volatility penalty from the return, encouraging higher risk-adjusted performance. This acts as a proxy for the sharpe ratio but is simpler to calculate, but is not as accurate as the real function.

3. **Drawdown-based Reward:** Penalizes large drops from peak portfolio value, promoting more stable growth.

Each reward function serves to emphasize different objectives, allowing experimentation with agent behavior under varying definitions of success.

The Deep Q-Network is implemented using PyTorch. It maps state vectors to Q-values for a discrete set of portfolio actions. For the action space, all the possible combination of the three assets, that sum to 1 are calculated with a 0.05 step size. For example the combination 0.9 Apple, 0.05 Google and 0.05 Microsoft is a possible combination, as well as 0.4 Apple, 0.4 Google and 0.2 Microsoft. All these possible vectors that some to one are 231 cases. This means that our neural network that learns the Q-values, has an input size of 231. The architecture consists of a feed-forward neural network with multiple hidden layers and ReLU activations.

Key training techniques include:

- **Experience Replay:** Past transitions are stored and sampled randomly to break correlation between updates.

- **Target Network:** A separate, periodically updated copy of the main network is used to compute stable target Q-values.

- **Double DQN Strategy:** Used to reduce overestimation of action values.

- **Exploration-Exploitation Trade-off:** ϵ-greedy policy with decay ensures balanced exploration during training.

To train the model for the following steps are followed in the code:

1. Initialize replay buffer $\mathcal{D}$, Q-network $Q(s, a; \theta)$, and target network $Q(s, a; \theta^-)$.

2. For each episode:

(a) Observe initial state s_0.

(b) For each time step t:

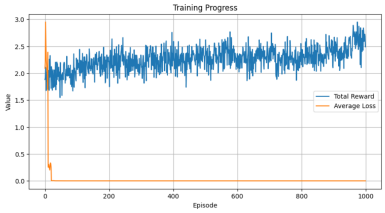
- Select action a_t using an ϵ-greedy policy.
- Execute a_t, observe reward r_t and next state s_{t+1}.
- Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$.
- Sample a mini-batch from $\mathcal{D}$ and compute target y.
- Minimize the loss $\mathcal{L}(\theta)$.
- Periodically update $\theta^- \leftarrow \theta$.

The agent is trained over 1,000 episodes using the training portion of the data (80%). Loss is computed using Smooth L1 Loss (Huber loss), which combines the stability of mean squared error with the robustness of mean absolute error. Gradient clipping is applied to prevent instability during updates.

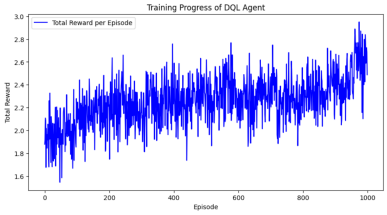
After training, the agent is evaluated on the remaining 20% of the data. Its performance is benchmarked against two alternative strategies:

- **Equal-weight allocation:** The portfolio is rebalanced daily with equal weights across all assets.
- **Random allocation:** Asset weights are randomly generated at each step.

After running the experimentation, the best result is obtained from the portfolio return reward function and the other two objective functions are not doing so good. Also, the proxy for sharpe ratio is doing marginally better than the equal-weight portfolio the other parameters are not so optimal.



(a) Reward and Huber Loss during training



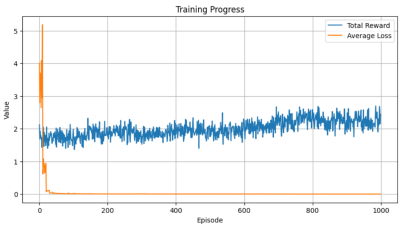
(b) Reward during training

Figure 4: Portfolio return as objective function.

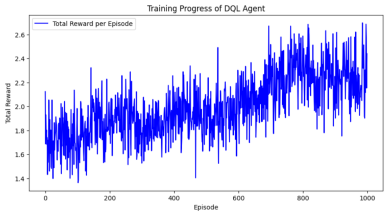
It can be observed in Figures 4 5 6 that the training minimizes the loss and the rewards have more or less a positive trend showing that the model is getting better over the training periods.

The model is then tested on the out-of-sample model and the the result can be observed in Figures 7 8 9. The portfolio with return as the objective function is outperforming the equal-weight portfolio by almost 29%.

This implementation successfully demonstrates that a Deep Q-Learning agent can be trained to manage a multi-asset portfolio and outperform simple heuristic-based strategies on historical data. Further extensions could incorporate transaction costs, more frequent trading intervals, more enhanced objective functions,

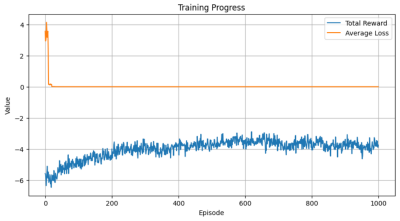


(a) Reward and Huber Loss during training

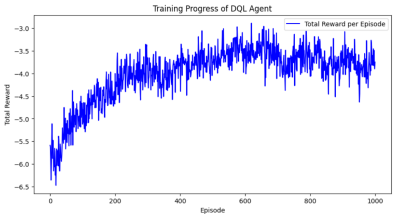


(b) Reward during training

Figure 5: Sharpe ratio proxy return as objective function.



(a) Reward and Huber Loss during training



(b) Reward during training

Figure 6: Drawdown proxy return as objective function.

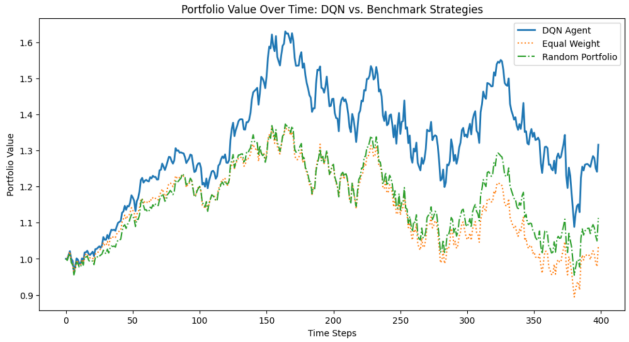


Figure 7: Portfolio value over time with portfolio return as reward.

as well as different loss functions and training hyperparameters. Although, it has to be stated that for portfolios with more assets the complexity of the neural network will increase exponentially and this method would not be that suitable. Also, because the nature of this method is discrete actions, a lot of actions can get lost, for example here we only consider trades with steps of 0.05 where as selling and buying 0.01 of each asset is also a possibility. So, it would be better to use methods that have continuous action space. In summary the performance

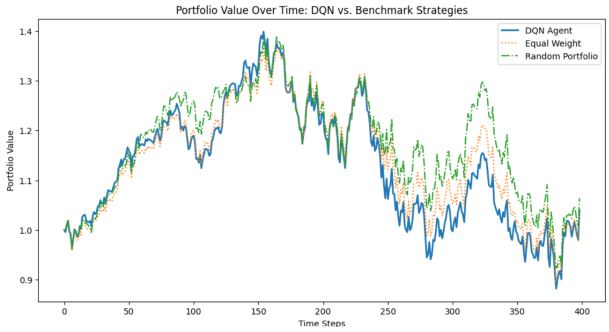


Figure 8: Portfolio value over time with sharpe ratio proxy as reward

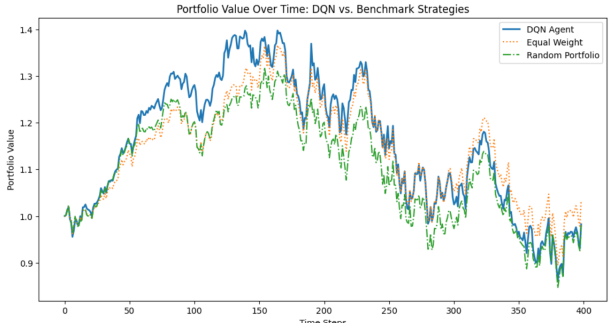


Figure 9: Portfolio value over time with drawdown proxy as reward

of the agents can be oberved in the bellow table.

Strategy	**Final Value**
DQN (Return Reward)	1.32
DQN (Sharpe Reward)	1.04
DQN (Drawdown Reward)	0.98
Equal-Weight Baseline	1.03

Table 2: Final portfolio values for different strategies after evaluation.

5.3 Advantage Actor-Critic (A2C)

We first define a market environment based on real historical data. Using the yfinance library, daily price data for three major technology stocks Apple (AAPL), Microsoft (MSFT), and Google (GOOGL) were collected over the period from January 1, 2020, to January 1, 2025. After filtering for missing values, we calculate cumulative returns to simulate price trajectories and compute daily returns for the reinforcement learning environment. We employed the A2C algorithm from the `stable-baselines3` library, using an MLP policy and a very

low learning rate (like 1e-6) to ensure stable convergence. The environment is vectorized using `DummyVecEnv` to interface with the RL framework. Training was conducted over 1,000,000 timesteps for the algorithm.

To isolate the learning dynamics from real-world noise and structural changes, we construct a synthetic environment based on the Ornstein-Uhlenbeck (OU) process. This mean-reverting stochastic process is widely used in quantitative finance to model interest rates and commodity prices. The OU process is defined by the following stochastic differential equation:

$$dx_t = \theta(\mu - x_t)dt + \sigma dW_t,$$

where $\theta > 0$ is the rate of mean reversion, μ is the long-term mean (set to 0), σ is the volatility, and dW_t is a Wiener process increment. The process exhibits mean-reverting behaviour and is commonly used in financial modelling and physical systems.

We simulate multivariate OU processes with the 3 assets mentioned (Apple (AAPL), Microsoft (MSFT), and Google (GOOGL)), each initialized either at the long-run mean (μ) or at a value far from the mean (e.g., $x_0 = 5.0$) to study mean reversion dynamics. Parameters such as the mean reversion rate ($\theta = 0.15$), volatility ($\sigma = 0.5$), and time increment ($\Delta_t = 1/252$) were chosen to reflect daily returns. The final process is shown in Figure 10.

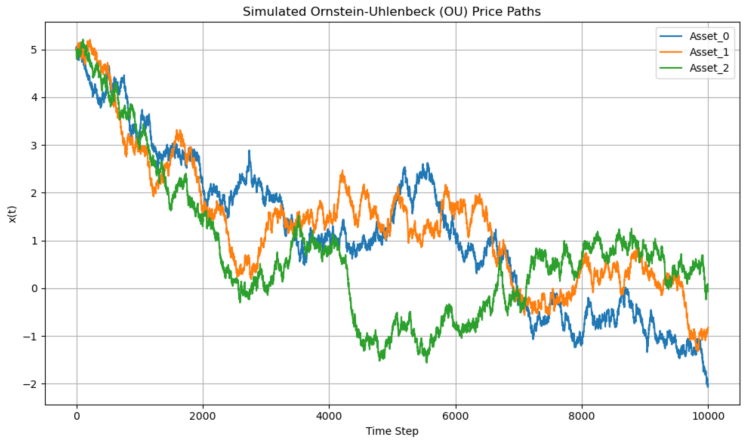


Figure 10: Trajectory the created Ornstein–Uhlenbeck process. The price paths starts from $x_0 = 5.0$ and learns to remain revert to the mean $\mu = 0$.

To understand how RL learns mean reversion, a minimalist control environment was created (we called it `OUControlEnv`) where the agent observes the current state $x(t)$ and can apply a direct control signal. The reward is set to the negative squared distance from the mean, encouraging the agent to push the system back toward equilibrium. Figure 11 illustrates the trajectory x_t of the Ornstein–Uhlenbeck process over 10.000 discrete time steps during the final

episode of A2C training. The process is initialized at $x_0 = 5.0$, a value significantly distant from the long-term mean $\mu = 0$, in order to assess the agent's ability to learn a mean-reverting policy.

Training results show that A2C successfully learns a stabilizing strategy, which means the A2C agent learns a continuous-valued control policy capable of finely stabilizing the system around the mean (See Figure 11).

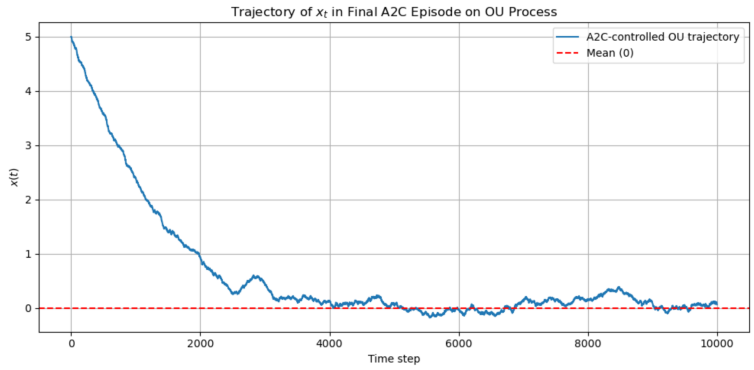


Figure 11: Trajectory of x_t in the final episode of A2C training on an Ornstein–Uhlenbeck process. The agent starts from $x_0 = 5.0$ and learns to remain close to the mean $\mu = 0$.

After training the A2C agent on both environments, we compare its performance to a naïve equal-weight portfolio strategy on the S&P500. We rerun the `PortfolioEnv` using the trained A2C model and OU-generated prices. The agent's performance is benchmarked against a naïve equal-weight portfolio strategy, where capital is distributed equally among all assets at each time step.

Two key performance metrics are analyzed:

- **Portfolio Value Over Time:** The evolution of total wealth is tracked across the investment horizon.
- **Sharpe Ratio:** A standard measure of risk-adjusted return, defined as:

$$\text{Sharpe Ratio} = \frac{\mathbb{E}[r]}{\sigma_r}\sqrt{252}, \tag{24}$$

  where r is the series of log returns, $\mathbb{E}[r]$ is the mean return, σ_r is the standard deviation of returns, and the factor $\sqrt{252}$ annualizes the Sharpe ratio based on daily returns.

Figure 12 shows that the A2C agent grows its portfolio value and outperforms the equal-weight baseline in the OU process environment. The final portfolio value and Sharpe ratio are both higher for the A2C strategy, indicating more efficient risk-adjusted performance.

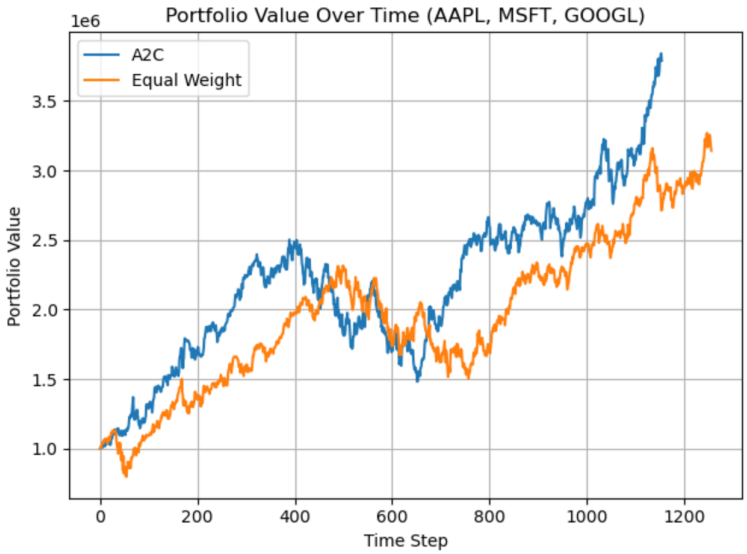


Figure 12: Portfolio value over time for A2C model.

Strategy	**Final Portfolio Value ($)**	**Sharpe Ratio**
A2C (RL Agent)	3,789,708.93	1.0622
Equal Weight Benchmark	3,142,314.32	0.8081

Table 3: Comparison of final portfolio values and Sharpe ratios between A2C and an equal-weight benchmark strategy.

The results in Table 3 demonstrate that RL, and in particular the A2C algorithm, can learn profitable portfolio allocation strategies in a mean-reverting synthetic environment. This synthetic setup, based on the OU process, offers a controlled testbed for evaluating learning behavior without the structural nonstationarity and noise found in real-world data.

Moreover, the use of log-return rewards and multiplicative wealth updates provides a financially grounded objective that aligns with growth-optimal strategies. Future enhancements could include the *incorporation of transaction costs*, state discretization for interpretability, and training on a broader range of market regimes to assess generalization. The incorporation of transaction costs is actually something we tried, which is described below. Because, in real-world financial markets, portfolio rebalancing is not free — trades incur transaction costs that can significantly reduce net returns, especially for high-turnover strategies. Reinforcement learning (RL) agents, if left unconstrained, may adopt overly reactive policies that result in frequent portfolio reallocations. To encourage more realistic behavior, we extend our portfolio environment to include transaction cost penalties and examine how different levels of cost affect the

agent's behavior and performance.

We modify the existing `PortfolioEnv` environment by introducing a transaction cost parameter c, which models proportional slippage or trading friction. At each time step, the agent incurs a cost proportional to the portfolio *turnover*, defined as the ℓ_1 norm of the difference between the new and previous portfolio weights:

$$\text{Turnover}_t = \sum_{i=1}^{N} |w_{t,i} - w_{t-1,i}|$$

The net portfolio return is then adjusted to reflect this cost:

$$r_t^{\text{net}} = r_t^{\text{gross}} \cdot (1 - c \cdot \text{Turnover}_t)$$

The reward function includes both the log of net return and a penalty for turnover:

$$\text{Reward}_t = \log\left(r_t^{\text{net}}\right) - \lambda \cdot c \cdot \text{Turnover}_t$$

This formulation encourages the agent to rebalance only when the expected benefit outweighs the cost of adjusting positions.

An A2C agent is trained for 100.000 timesteps on the modified environment using historical price and return data for three assets (AAPL, MSFT, and GOOGL), with a baseline transaction cost of $c = 0.001$. The resulting policy balances performance and trading costs, and is benchmarked against a uniform buy-and-hold portfolio.

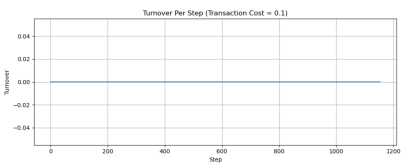
To evaluate how transaction costs influence trading behavior, we perform a sensitivity analysis by testing the trained A2C policy in environments with varying transaction costs:

$$c \in \{0.1,\ 0.01,\ 0.001,\ 0.0001\}$$

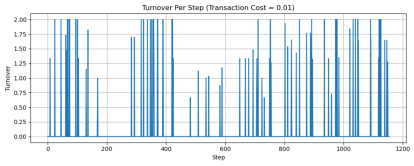
The policy is held fixed (trained at $c = 0.001$), and we record the turnover at each time step for each cost setting. The figures below display the resulting turnover time series.

As expected, when transaction costs are high ($c = 0.1$), the agent avoids trading altogether. Conversely, under low transaction costs ($c = 0.0001$), the agent engages in more frequent rebalancing. The results demonstrate cost-aware behavior: the agent maintains its portfolio allocation unless the expected gain clearly outweighs the frictional cost.

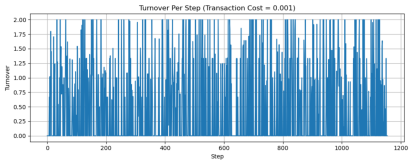
This experiment demonstrates that modeling transaction costs is essential for deploying RL-based portfolio strategies in realistic settings. Even without retraining, the agent exhibits adaptive behavior in response to different market frictions. The findings highlight the value of reward shaping, cost-aware decision-making, and robustness testing under varied transaction cost regimes.



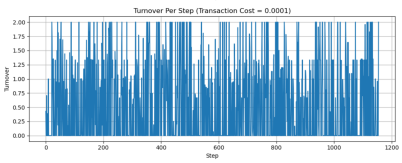
(a) Transaction cost $c = 0.1$



(b) Transaction cost $c = 0.01$



(c) Transaction cost $c = 0.001$



(d) Transaction cost $c = 0.0001$

Figure 13: Turnover per step under different transaction cost levels using a fixed A2C policy. Higher transaction costs suppress rebalancing activity, while lower costs allow more active trading.

5.4 Overview

Place outcomes, explaina bit what you see

6 Conclusion & Discussion

In this project, we investigated the performance of three reinforcement learning algorithms—SARSA, Deep Q-Learning, and Advantage Actor-Critic (A2C)—within the context of financial trading environments. Each method was assessed based on its learning stability, profitability, and adaptability to market dynamics.

SARSA, as an on-policy method, offered stable learning behavior and conservative strategies. While it achieved moderate returns, it was more robust in avoiding large drawdowns, suggesting its suitability in risk-averse trading frameworks. However, its simplicity and limited capacity to generalize from complex state spaces constrained its overall performance in more volatile conditions.

Deep Q-Learning demonstrated a stronger ability to approximate optimal policies through neural networks, leading to improved returns over SARSA in most scenarios. Its off-policy nature allowed it to explore more aggressive strategies, though this came at the cost of occasional instability and sensitivity to hyperparameters. The results indicated that while Deep Q-Learning can outperform traditional tabular methods, it requires careful tuning and regularization to be effective in financial domains.

A2C emerged as the most promising approach among the three. By combining the strengths of both policy-based and value-based methods, it achieved a balance between performance and learning stability. A2C consistently produced higher cumulative rewards and was better equipped to adapt to changing market

conditions, thanks to its actor-critic architecture. Its ability to optimize both the policy and the value function in parallel contributed to faster convergence and more nuanced trading behavior.

Overall, while each algorithm has its merits, A2C proved to be the most capable in our financial RL environment, suggesting that actor-critic methods hold strong potential for real-world trading systems. Future work may focus on extending this approach with more advanced variants such as PPO or SAC, and incorporating richer market features and transaction cost models to enhance realism and applicability.

Limitations

While the results of this study provide valuable insights into the application of reinforcement learning algorithms in finance, several limitations must be acknowledged.

1. *Simplified Environment:* The trading environment used in our experiments was a simulation with constrained parameters and assumptions. Market frictions such as slippage, liquidity constraints, and transaction costs were either omitted or overly simplified. These factors play a significant role in real-world trading and could significantly affect algorithm performance.

2. *Limited Asset Scope:* Our experiments focused on a narrow set of financial instruments or synthetic data, which may not generalize well to broader markets. Real financial environments are multidimensional, involving diverse asset classes, correlations, and macroeconomic factors that were not incorporated in our models.

3. *Short Training Horizons and Sparse Data:* Due to computational constraints, the training horizons were relatively short. Reinforcement learning algorithms, particularly A2C and Deep Q-Learning, benefit from long-term training with vast and diverse datasets. The limited data scope may have hindered the agents from learning more complex, generalizable policies.

4. *Model and Hyperparameter Sensitivity:* Deep learning-based methods like Deep Q-Learning and A2C were sensitive to hyperparameter choices and initialization. Performance variability across runs was non-negligible, highlighting the need for more systematic hyperparameter tuning and evaluation across multiple seeds to ensure robustness.

5. *Evaluation Metrics and Risk Considerations:* Performance evaluation focused primarily on cumulative rewards or returns, without incorporating comprehensive risk-adjusted metrics such as Sharpe ratio, maximum drawdown, or downside risk. This may have skewed the perceived effectiveness of certain strategies that were risk-seeking rather than risk-aware.

6. *Stationarity Assumption:* All algorithms implicitly assumed stationarity in market dynamics, which rarely holds true in real financial markets. Regime shifts, sudden volatility spikes, and changing correlations are common and would challenge the adaptability of RL agents trained in more stable, simulated settings.

In sum, while the study provides a valuable comparative framework, it should be seen as a foundation for further work rather than a ready-to-deploy solution. Addressing these limitations through more realistic simulations, richer datasets, and improved evaluation methodologies will be critical for advancing reinforcement learning in practical financial applications.

Future Work

Looking ahead, there are several promising directions to further advance the application of reinforcement learning in finance based on the findings of this study. One important step is the integration of transaction costs, bid-ask spreads, and other market frictions into the simulation environment to more accurately reflect the realities of financial trading. This would allow agents to develop strategies that are not only profitable but also cost-efficient and realistic. Expanding the scope of the environment to include multiple assets could enable portfolio-level decision-making and allow for the study of diversification and hedging strategies, which are essential for managing risk in practice. Additionally, adopting more advanced reinforcement learning algorithms such as Proximal Policy Optimization (PPO), Soft Actor-Critic (SAC), or Deep Deterministic Policy Gradient (DDPG) could improve both the stability and performance of the learning process, especially in more complex and continuous action spaces. Enriching the state space with technical indicators, macroeconomic data, or alternative sources such as news sentiment could enhance the agent's ability to recognize market regimes and adapt accordingly. Another valuable direction is the refinement of reward functions to include risk-sensitive metrics, such as Sharpe ratio or maximum drawdown, which would encourage the development of more robust and stable strategies. Given the inherently non-stationary nature of financial markets, future work could also explore meta-learning or transfer learning techniques to help agents retain useful knowledge and adapt more quickly to shifting market conditions. Ultimately, deploying these models in live or paper-trading environments would be essential to validate their real-world applicability and performance, offering insights into how reinforcement learning can be responsibly and effectively applied to financial decision-making.

References

[1] T. I. Team, *Modern portfolio theory: What mpt is and how investors use it*, Aug. 2023. [Online]. Available: `https://www.investopedia.com/terms/m/modernportfoliotheory.asp`.

[2] L. F. Torres, *Portfolio optimization: The markowitz mean-variance model*, Dec. 2023. [Online]. Available: https://medium.com/latinxinai/portfolio-optimization-the-markowitz-mean-variance-model-c07a80056b8a.

[3] H. Niu, S. Li, and J. Li, "Metatrader: An reinforcement learning approach integrating diverse policies for portfolio optimization," *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*, pp. 1573–1583, Oct. 2022. DOI: 10.1145/3511808.3557363.

[4] A. Filos, "Reinforcement learning for portfolio management," *Imperial College London*, 2018.

[5] Q. Y. Lim, Q. Cao, and C. Quek, "Dynamic portfolio rebalancing through reinforcement learning," *Neural Computing and Applications*, vol. 34, no. 9, pp. 7125–7139, Dec. 2021. DOI: 10.1007/s00521-021-06853-3.

[6] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," *Advances in Neural Information Processing Systems*, 1999.

[7] Kakade and Langford, "Approximately optimal approximate reinforcement learning," *Proceedings of the nineteenth international conference on machine learning (pp. 267-274)*, 2002.

[8] Szepesvari, "Algorithms for reinforcement learning," *Springer nature*, 2009.

[9] X.-Y. Liu, H. Yang, J. Gao, and C. D. Wang, "Finrl: Deep reinforcement learning framework to automate trading in quantitative finance," *SSRN*, 2021. SSRN: 3955949.

[10] I. Halperin, "The qlbs q-learner goes nuqlear: Fitted q iteration, inverse rl, and option portfolios," *SSRN*, 2018. SSRN: 3102707.

[11] T.-V. Pricope, "Deep reinforcement learning in quantitative algorithmic trading: A review," *arXiv preprint arXiv:2106.00123*, 2021.

[12] R. Wirth and J. Hipp, "Crisp-dm: Towards a standard process model for data mining," *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, vol. 1, pp. 29–39, Apr. 2000. DOI: 10.1.1.198.5133.

[13] C. Schröer, F. Kruse, and J. M. Gómez, "A systematic literature review on applying crisp-dm process model," *Procedia Computer Science*, vol. 181, pp. 526–534, 2021. DOI: 10.1016/j.procs.2021.01.199.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 2018.

[15] V. M. et al., "Asynchronous methods for deep reinforcement learning," *International Conference on Machine Learning (ICML)*, 2016.

[16] J. Moody and M. Saffell, "Performance functions and reinforcement learning for trading systems and portfolios," *Journal of Computational Intelligence in Finance*, vol. 6, pp. 3–13, 1998.

[17] Z. Jiang, D. Xu, and J. Liang, "A deep reinforcement learning framework for the financial portfolio management problem," *arXiv preprint arXiv:1706.10059*, 2017.