Department of Informatics
King's College London
United Kingdom

7CCSMPRJ - Individual Project

Explainable Deep Reinforcement Learning for Portfolio Optimisation of Financial Assets

Name: **Ingrid Pérez Aguilera**

Student Number: K24087939

Course: Computational Finance M.Sc.

Supervisor: **Riaz Ahmad**

Word count: 15155

Abstract

The dynamic and stochastic nature of financial markets together with the highly non-stationary and noisy financial time series data make the task of portfolio optimisation particularly challenging. However, their nature makes them particularly well-suited for Deep Reinforcement Learning (DRL) algorithms, which can learn a trading strategy directly from the environment. Nonetheless, there are still significant challenges preventing its widespread adoption in the financial industry, such as difficulty in finding the appropriate algorithm with a suitable reward function and lack of transparency and interpretability. This thesis proposes an explainable DRL model for portfolio optimisation with the ability to leverage historical data to learn the optimal trading strategy that balances return maximisation and risk minimisation. Five prominent DRL algorithms are implemented and their performance is evaluated in different scenarios, including the impact of a larger financial environment representation, portfolio size and asset composition. Once the algorithms are trained, post-hoc explainability techniques are applied to understand which market conditions lead to a particular portfolio allocation. Although the performance of the algorithms does not generally outperform all of the benchmarks, the results show that the agents are a powerful tool in portfolio management capable of learning from high-dimensional data and adapting to changing market conditions. The crucial contribution of this research lies in the explainability framework, particularly the use of SHapley Additive exPlanations (SHAP) to provide insights into the decision-making process over the testing period in conjunction with interpretation for specific trading days.

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Riaz Ahmad, for his guidance and support throughout this project. I would also like to thank my Master's colleagues for being there during this journey and the valuable discussions we had.

I would like to convey my appreciation to my partner for his unwavering support, encouragement and all the help in reviewing the code and the report. I would also like to acknowledge my family's continuous support and encouragement throughout my studies.

Table of Contents

1 Introduction **1**

1.1 Objectives . . . 3

1.2 Report Structure . . . 4

2 Background **6**

2.1 Portfolio Optimisation . . . 6

2.1.1 Modern Portfolio Theory . . . 7

2.2 Deep Reinforcement Learning . . . 9

2.2.1 Reinforcement Learning . . . 10

2.2.2 Deep Reinforcement Learning Algorithms . . . 16

2.3 Explainable Artificial Intelligence . . . 23

2.3.1 Feature Importance . . . 25

2.3.2 Local Interpretable Model-agnostic Explanations . . . 25

2.3.3 Shapley Additive Explanations . . . 27

2.4 State of the Art of Deep Learning and Explainability for Portfolio Optimisation . . . 32

3 Methodology **36**

3.1 Problem Definition . . . 36

3.2 Markov Decision Process Model . . . 37

3.3 Deep Reinforcement Learning Algorithms 40
3.3.1 Hyper-parameter tuning 40
3.4 Post-hoc Explainability 41
3.4.1 Surrogate Model Explainability 42
3.4.2 Direct Model Explainability 43

4 Results **45**

4.1 Dataset 45
4.2 Experiment Design 47
4.3 Evaluation 51
4.3.1 Performance Metrics 51
4.3.2 Benchmark Strategies 52
4.4 Experiment: Algorithm Comparison 53
4.5 Experiment: Environment Representation 57
4.6 Experiment: Hyper-parameter Tuning 59
4.7 Explainability Results 61
4.7.1 Feature Importance Results 62
4.7.2 Local Interpretable Model-agnostic Explanations Results 63
4.7.3 Shapley Additive Explanations Results 64

5 Legal, Social, Ethical and Professional Issues **68**

5.1 Legal Issues 68
5.2 Social Issues 69
5.3 Ethical Issues 70
5.4 Professional Issues 71

6 Conclusion **72**

6.1 Future Work 73

References 75

A Algorithms 85

A.1 Deep Reinforcement Learning Algorithms 85

A.2 Explainability Algorithms 91

B State Representation 92

B.1 Technical Indicators 92

B.2 Macroeconomic Indicators 97

C Hyper-parameter tuning 99

D Datasets 101

D.1 Equities 101

D.1.1 Dow Jones 30 101

D.1.2 Euro Stoxx 50 103

D.1.3 FTSE 100 107

D.2 Commodities 112

D.3 Currencies 112

E Default Hyper-parameter Selection 114

F Evaluation Metrics 117

G Experiment Results 120

G.1 Experiment: Algorithm Comparison 120

G.2 Experiment: Environment Representation 122

H Explainability Results 124

H.1 Feature Importance 124

H.2 Local Importance Model-Agnostic Explanation Results 126

I Code 128

I.1 Configuration Module 131

I.2 Data Download and Pre-processing Module 142

I.3 Agents Module 152

I.4 Environments Module 156

I.5 Benchmarking Module 163

I.6 Explainability Module 168

I.7 Visualisation Module 180

I.8 Optimisation Module 196

List of Figures

2.1 Efficient Frontier in Risk-Return Space. [1] 9

2.2 Agent interaction with environment 11

2.3 Markov Decision Process with policy, transition, and reward functions . . 12

2.4 Taxonomy of Reinforcement Learning Algorithms [2] 17

2.5 Taxonomy of Explainable Artificial Intelligence Methods [3] 24

4.1 Evolution of the Cumulative Returns for the DowJones30 dataset with the OHLCV prices and indicators environment representation. 55

4.2 Top features from the surrogate model according to feature importance. . 62

4.3 Mean feature importance per asset from the surrogate model. 63

4.4 LIME explanations for the A2C algorithm at the first time step of the test dataset for the MSFT asset. The orange bars indicate features that contribute positively to the prediction, while the blue bars indicate features that contribute negatively. 64

4.5 Beeswarm SHAP explanations for the A2C algorithm for the AAPL asset. The x-axis represents the SHAP values, whilst the y-axis represents the top features. The colour indicates the feature value, with magenta being high and blue being low. 65

4.6 SHAP force plot for the weight allocation of the AAPL asset for the A2C algorithm. 66

4.7 SHAP force plot for the weight allocation of the AAPL asset for the A2C algorithm. 67

4.8 SHAP force plot for the weight allocation of the AAPL asset for the A2C algorithm at the first time step of the test dataset. 67

E.1 Train-Validation-Test Split for the Hyper-parameter tuning on a sample of five assets from the Dow Jones Industrial Average index. 115

E.2 Hyper-parameter tuning results for the A2C algorithm. 115

H.1 Top features grouped by asset from the surrogate model according to feature importance. 125

H.2 Mean feature importance per type of feature from the surrogate model. . 125

H.3 LIME explanations for the A2C algorithm at the first time step of the test dataset for the AAPL asset. 126

H.4 LIME explanations for the A2C algorithm at the first time step of the test dataset for the CSCO asset. 126

H.5 LIME explanations for the A2C algorithm at the first time step of the test dataset for the HON asset. 127

H.6 LIME explanations for the A2C algorithm at the first time step of the test dataset for the V asset. 127

List of Tables

2.1 Comparison of Deep Reinforcement Learning Algorithms 23

4.1 Train-Validation-Test Split for each dataset 47

4.2 Default hyper-parameter configurations. 48

4.2 Default hyper-parameter configurations. 49

4.3 Summary of experiments conducted. 50

4.4 Algorithm comparison results for the A2C implementation. 54

4.5 Algorithm comparison results for the DowJones30 dataset. The colour correspond to the best performing configurations, with blue for the best performing DRL algorithm and green for the best benchmark. 56

4.6 Environment representation experiment comparison according to the Sharpe ratio. The colour correspond to the best performing per row, with blue for the DowJones30 dataset and green for the Commodities dataset. . . . 57

4.6 Environment representation experiment comparison according to the Sharpe ratio. The colour correspond to the best performing per row, with blue for the DowJones30 dataset and green for the Commodities dataset. . . . 58

4.7 Hyper-parameter tuning experiment results for the DowJones30 dataset with simple environment representation. 59

4.8 Hyper-parameter tuning experiment results for the DowJones30 dataset with indicators environment representation. 60

C.1 Hyper-parameter tuning configurations for different RL algorithms. 100

D.1 Constituents of the Dow Jones Industrial Average index as of April 2025. [4] . . . 103

D.2 Constituents of the Euro Stoxx 50 index as of April 2025. [5] . . . 107

D.3 Constituents of the FTSE100 index as of April 2025. [6] . . . 112

D.4 6 Commodities Futures Contracts . . . 112

D.5 10 Currency Pairs . . . 113

E.1 Results of hyper-parameter tuning on the test dataset. . . . 116

G.1 Algorithm comparison results for the PPO implementation. . . . 121

G.2 Algorithm comparison results for the DDPG implementation. . . . 121

G.3 Algorithm comparison results for the TD3 implementation. . . . 121

G.4 Algorithm comparison results for the SAC implementation. . . . 122

G.5 Environment representation experiment comparison according to the cumulative return. The colour correspond to the best performing per row, with blue for the DowJones30 dataset and green for the Commodities dataset. . . . 122

G.5 Environment representation experiment comparison according to the cumulative return. The colour correspond to the best performing per row, with blue for the DowJones30 dataset and green for the Commodities dataset. . . . 123

List of Algorithms

1 Advantage Actor-Critic (A2C) Pseudo-code 86
2 Proximal Policy Optimisation (PPO) Pseudo-code 87
3 Deep Deterministic Policy Gradient (DDPG) Pseudo-code 88
4 Twin Delayed Deep Deterministic Policy Gradient (TD3) Pseudo-code . . 89
5 Soft Actor-Critic (SAC) Pseudo-code 90
6 Shapley Value Approximation 91

Nomenclature

Numbers and Arrays

a	Scalar
$\mathbf{a}$	Vector
$\mathbf{A}$	Matrix
$\mathbf{1}$	Matrix of 1's

Sets

$\mathbb{A}$	Set
$\mathbb{R}$	Set of real numbers
$[a, b]$	Real interval including a and b
(a, b)	Real interval excluding a and b
$[a, b)$	Real interval including a but excluding b
$(a, b]$	Real interval excluding a but including b
$\{0, 1\}$	Set containing 0 and 1
$\{0, 1, ..., n\}$	Set of all integers between 0 and n inclusive
$\emptyset$	Empty set
$\mathbb{A} \subset \mathbb{B}$	Set $\mathbb{A}$ is a subset of set $\mathbb{B}$
$\mathbb{A} \subseteq \mathbb{B}$	Set $\mathbb{A}$ is a subset or equal to set $\mathbb{B}$
$\mathbb{A} \cap \mathbb{B}$	Intersection of sets $\mathbb{A}$ and $\mathbb{B}$

$\mathbb{A} \cup \mathbb{B}$	Union of sets $\mathbb{A}$ and $\mathbb{B}$
$\mathbb{A} \setminus \mathbb{B}$	Set difference of sets $\mathbb{A}$ and $\mathbb{B}$

Indexing

a_i	Element i of vector $\mathbf{a}$, with indexing start at 1
$A_{i,j}$	Element i, j of matrix $\mathbf{A}$, with indexing start at 1

Algebra Operations

$\mathbf{A}^T$	Transpose of matrix $\mathbf{A}$
$\mathbf{A}^{-1}$	Inverse of matrix $\mathbf{A}$

Probability

$P(x)$	Probability distribution over a random variable x
$E[x]$	Expected value of random variable x
$Var[x]$	Variance of random variable x
$Cov[x, y]$	Covariance of random variables x and y
μ	Mean
σ	Standard deviation
σ^2	Variance
Σ	Covariance matrix
$\mathcal{N}(\mu, \sigma^2)$	Normal distribution with mean μ and variance σ^2
$\mathcal{N}(0, 1)$	Standard normal distribution

Functions

$f : \mathbb{A} \rightarrow \mathbb{B}$	Function f with domain $\mathbb{A}$ and range $\mathbb{B}$
$f(x)$	Function f evaluated at x
$f(x; \theta)$	Function f with parameters θ evaluated at x
$f'(x)$	Derivative of function f

$\int f(x)\,dx$	Integral of function f with respect to x
$\min f(x)$	Minimum of function f
$\min_x f(x)$	Minimum of function f with respect to x
$\max f(x)$	Maximum of function f
$\max_x f(x)$	Maximum of function f with respect to x
$\log x$	Natural logarithm of x
$x!$	Factorial of x

Markov Decision Process

t	Time step
$\mathcal{S}$	State space
s_t	Observation of the state at time step t
$\mathcal{A}$	Action space
a_t	Action taken at time step t
$R(s, a, s')$	Reward function
r_t	Reward received at time step t
$T(s, a, s')$	Transition function
γ	Discount factor
π	Policy
G	Long-term reward
V^π	State Value function for policy π
Q^π	State-Action Value function for policy π
A^π	Advantage function for policy π

Other Symbols

$\forall$	For all

$\in$	Element of
∇	Gradient
$\sum$	Summation symbol

Glossary

Advantage Actor-Critic Algorithm that uses both an actor (policy) and a critic (value function) to learn optimal policies by estimating the advantage of actions taken.

Algorithmic Trading Use of computer algorithms to automate trading decisions and execute trades in financial markets.

Artificial Intelligence Simulation of human intelligence processes by machines enabling them to perform tasks that typically require human intelligence, such as visual perception, speech recognition, decision-making, and language translation.

Bellman Equations Set of equations that describe the relationship between the value of a state or action and the values of subsequent states or actions in dynamic programming and reinforcement learning.

Black Box Term used to describe a system or model whose internal workings are not transparent or easily understood.

Deep Deterministic Policy Gradient Algorithm that uses deep neural networks to learn policies for continuous action spaces, combining the benefits of deep learning and policy gradient methods.

Deep Learning Subset of machine learning that uses neural networks with many layers to learn from large amounts of data, enabling the model to automatically learn complex patterns and representations.

Deep Neural Network Type of artificial neural network with a succession of layers of non-linear transformations capable of learning a representation of the data with various levels of abstraction.

Deep Reinforcement Learning Combination of deep learning and reinforcement learning, where deep neural networks are used to approximate the value function or policy in reinforcement learning tasks.

Efficient Frontier Set of optimal portfolios that offer the highest expected return for a given level of risk, or the lowest risk for a given level of expected return, in the context of portfolio optimisation.

Entropy Measure of uncertainty or randomness in a system, often used in the context of information theory and reinforcement learning to encourage exploration.

Explainable Artificial Intelligence Methods and techniques in the application of artificial intelligence that make the results of the models understandable by humans, providing insights into how decisions are made.

Exploitation Process of leveraging known information to make decisions or take actions that are expected to yield the highest reward.

Exploration Process of investigating and experimenting with different actions or strategies to discover their effects and improve decision-making.

Feature Importance Technique for determining the contribution of each feature in a machine learning model to its predictions, helping to identify which features are most influential.

Financial Markets Marketplaces where people trade financial securities, commodities, and other fungible items of value at low transaction costs and at prices that reflect supply and demand.

Hyper-parameter Parameter that is set before the learning process begins and control the learning process of a machine learning model, such as learning rate, batch size, and number of layers in a neural network.

Local Interpretable Model-agnostic Explanations Technique for explaining the predictions of any machine learning model by approximating it with a locally interpretable model, allowing users to understand the model's behaviour in a specific instance.

Machine Learning Subset of artificial intelligence that enables systems to learn from data and improve their performance over time without being explicitly programmed.

Markov Decision Process Mathematical model for sequential-decision making where the outcomes are uncertain, characterised by states, actions, rewards and a transition function.

Mean-Variance Optimisation Investment strategy that aims to construct a portfolio of assets that maximises expected return for a given level of risk, or minimises risk for a given level of expected return.

Modern Portfolio Theory Investment theory that aims to construct a portfolio of assets that maximises expected return for a given level of risk, or minimises risk for a given level of expected return, through diversification.

Partially Observable Markov Decision Process Extension of Markov Decision Process where the agent does not have access to the complete state information, requiring the use of belief states or observations to make decisions.

Portfolio Optimisation Process of selecting the best distribution of assets in a portfolio to achieve specific investment goals, such as maximising returns or minimising risk, while considering constraints and preferences.

Proximal Policy Optimisation Algorithm that optimises policies by ensuring that updates to the policy are not too large, maintaining a balance between exploration and exploitation.

Python Interpreted, object-oriented, high-level programming language.

Reinforcement Learning Subset of machine learning where an agent learns to make decisions by taking actions in an environment to maximise cumulative reward.

Reward Function Function that defines the feedback signal received by an agent in reinforcement learning, guiding the agent's learning process by providing rewards or penalties based on its actions.

SHapley Additive exPlanations Method for interpreting machine learning models by assigning each feature an importance value for a particular prediction, based on cooperative game theory.

Soft Actor-Critic Algorithm that combines the benefits of off-policy learning and entropy regularisation, allowing for more exploration and better stability in learning policies for continuous action spaces.

Supervised Learning Machine learning task where a model is trained on labelled data to learn a mapping from inputs to outputs.

Twin Delayed Deep Deterministic Policy Gradient Extension of Deep Deterministic Policy Gradient that addresses the overestimation bias in value function estimation by using two critic networks and delaying policy updates.

Unsupervised Learning Machine learning task where a model learns patterns or structures in unlabelled data without explicit supervision.

Acronyms

A2C Advantage Actor-Critic.

A3C Asynchronous Advantage Actor-Critic.

ADX Average Directional Index.

AI Artificial Intelligence.

AI Act Artificial Intelligence Act.

ATR Average True Range.

AUD Australian Dollar.

BCS British Computer Society.

BXY British Pound Currency Index.

CAD Canadian Dollar.

CCI Commodity Channel Index.

CHF Swiss Franc.

CNY Chinese Yuan.

DDPG Deep Deterministic Policy Gradient.

DJIA Dow Jones Industrial Average.

DL Deep Learning.

DNN Deep Neural Network.

DPG Deterministic Policy Gradient.

DQN Deep Q-Network.

DRL Deep Reinforcement Learning.

DW30 Dow Jones 30 Index.

DX Directional Movement Index.

DXY U.S. Dollar Index.

EMA Exponential Moving Average.

EU European Union.

EUR Euro.

Euro Stoxx 50 Euro Stoxx 50 Index.

EXY Euro Index.

FCA Financial Conduct Authority.

FTSE 100 Financial Times Stock Exchange 100 Index.

FVX 5-Year Treasury Yield.

GBP British Pound.

HKD Hong Kong Dollar.

HTML HyperText Markup Language.

IET The Institution of Engineering and Technology.

INR Indian Rupee.

IRX 3-Month Treasury Yield.

JPY Japanese Yen.

KRW South Korean Won.

LIME Local Interpretable Model-agnostic Explanations.

LLM Large Language Model.

LSTM Long Short-Term Memory.

MACD Moving Average Convergence Divergence.

MAPE Mean Absolute Percentage Error.

MDI Negative Directional Index.

MDP Markov Decision Process.

MiFID II Markets in Financial Instruments Directive II.

ML Machine Learning.

MPT Modern Portfolio Theory.

MSE Mean Squared Error.

MVO Mean-Variance Optimisation.

OHLCV Open, High, Low, Close, Volume.

PCA Principal Component Analysis.

PDI Positive Directional Index.

POMDP Partially Observable Markov Decision Process.

PPO Proximal Policy Optimisation.

RL Reinforcement Learning.

ROC Rate of Change.

RSI Relative Strength Index.

S&P 500 Standard and Poor's 500 Index.

SAAC Synchronous Advantage Actor Critic.

SAC Soft Actor-Critic.

SHAP SHapley Additive exPlanations.

SMA Simple Moving Average.

TD3 Twin Delayed Deep Deterministic Policy Gradient.

TNX 10-Year Treasury Yield.

TRPO Trust Region Policy Optimisation.

U.S. United States.

UK United Kingdom.

USD United States Dollar.

VaR Value at Risk.

VIX Volatility Index.

VXD Volatility DW30.

XAI Explainable Artificial Intelligence.

Chapter 1

Introduction

Financial markets are highly complex systems influenced by numerous factors, including financial and political events, social trends and technological advancements. Moreover, their evolving and stochastic nature requires using the most advance computational developments to model the financial environment. The tasks of financial time series prediction and portfolio optimisation are considerably intricate, due to the semi-strong form of market efficiency and the high level of noise. [7]

Algorithmic trading focuses on the application of analytical methods to automatically execute trading actions based on an algorithm without human intervention. Initially, the field mainly studied the usage of a computer program to follow a predefined strategy [8]. More recently, algorithmic trading involves environment perception by learning feature representation from highly non-stationary and noisy financial time series data, and decision-making by exploring the environment and simultaneously taking actions without supervision [9].

Machine Learning (ML) has an advantage for the task due to its ability to learn from historical data and make predictions about the future state of an environment. Research has explored the application of Deep Learning (DL) in future price prediction of financial assets [10, 11, 7, 12]. However, its main disadvantage is the inability to directly deal with trading, requiring an additional step to convert the predictions into actionable strategies. In contrast, Reinforcement Learning (RL) allows the algorithm to learn a trading strategy directly from the environment, without the need for a separate step [13, 14]. In this case, there are two main approaches: first, the algorithm can learn the amount of assets to buy, sell or hold at each time step [15], or second, the algorithm can learn the optimal portfolio allocation and automatically rebalance the portfolio weights at each time step [16].

Despite the potential of RL in portfolio optimisation, its widespread adoption in the financial industry remains limited. This is primarily due to following challenges [17]:

1. difficulty in finding the appropriate algorithm with a suitable reward function and hyper-parameters to ensure efficiency and performance,
2. challenge of testing the algorithm in a real-world environment, and
3. lack of transparency of ML models, often referred to as black boxes, making it increasingly complex to interpret the algorithm's decisions.

In recent years, the rise in popularity of Artificial Intelligence (AI) and its widespread use have led to concerns regarding its decisions due to its black box nature. The concept of explainability in AI, known as Explainable Artificial Intelligence (XAI), refers to a model's ability to provide details and reasons to make itself understandable [18]. The term was first coined in 2016 to describe the need for users to effectively understand, trust and manage artificial intelligence applications [19]. The need for explainability becomes particularly relevant within the financial domain, where the regulatory framework requires transparency and accountability in automated decision-making. Various relevant applications, including volatility models [20], credit risk assessment [21] and portfolio construction [17], have explored the concept of explainability in financial applications.

Consequently, this thesis will focus on addressing the aforementioned challenges by exploring the application of Deep Reinforcement Learning (DRL) to portfolio optimisation and implementing explainability techniques.

1.1 Objectives

The objective of this thesis is to develop an explainable Deep Reinforcement Learning model for portfolio optimisation. A DRL model has the ability to leverage historical financial data to learn an investment strategy that efficiently allocates financial assets while maximising expected returns and minimising risk. Moreover, the incorporation of advanced explainability techniques enhances the interpretability and transparency of the model's decision-making. This project aims to bridge the gap between cutting-edge machine learning techniques and their practical application in finance by addressing the challenges of algorithm selection, simulation of real-world scenarios, and black box nature of ML models.

First, DRL models, such as Advantage Actor-Critic (A2C), Proximal Policy Optimisation (PPO), Deep Deterministic Policy Gradient (DDPG), Twin Delayed Deep Deterministic Policy Gradient (TD3) and Soft Actor-Critic (SAC), will be implemented to learn the optimal portfolio allocation from high-dimensional environment representations. The algorithms will be trained on historical financial data, including technical and macroeconomic indicators, with the goal of capturing the complex market dynamics. Each of the algorithms is better suited to a particular scenario, for instance, DDPG encourages maximum returns, while A2C reduces the variance and PPO is better at balancing exploration versus exploitation.

Second, post-hoc explainability techniques: Feature Importance, Local Interpretable

Model-agnostic Explanations (LIME) and SHapley Additive exPlanations (SHAP) will be applied to interpret the model's decisions. The goal of these is to understand which market conditions, represented by financial data, lead to the actions/decisions, encoded as portfolio weights.

Finally, the performance of the DRL models will be analysed in different scenarios, including the impact of a larger financial environment representation, portfolio size and asset composition. The results will be compared with traditional portfolio optimisation methods to evaluate the effectiveness of the proposed approach.

1.2 Report Structure

This report is organised into six chapters, each of which focuses on a concrete area related to the problem. Additional material, together with source code, is included in the appendices.

The current chapter, 1, presents the motivation and the objectives of this thesis. It gives an overview of the potential of DRL in portfolio optimisation and its main challenges, particularly the lack of transparency of ML models.

Chapter 2 provides an overview of the theoretical background of the project, including portfolio allocation and concepts. The problem of portfolio optimisation in the financial domain is outlined and the potential of DRL in this context is discussed. The chapter provides a comprehensive background explanation of the fundamentals of Deep Learning and Reinforcement Learning, including the main algorithms (A2C, PPO, DDPG, TD3, SAC). In addition, it gives an overview of the post-hoc explainability techniques: Feature Importance,Local Interpretable Model-agnostic Explanations (LIME) analysis and SHapley Additive exPlanations (SHAP) values, which will be used to interpret the model's decisions. Finally, it includes a comprehensive in-depth literature review on the topics of ML applied to portfolio optimisation and relevant applications of explainability techniques.

The methodology, chapter 3, describes the techniques and methods used to tackle the problem and outlines the implementation of the proposed solution. The chapter provides a detailed explanation of the architecture and components of the proposed DRL model, including the state representation and reward function. Furthermore, given the trained algorithms, it describes the implementation of the post-hoc explainability techniques.

The results of the experiments are presented in chapter 4, which analyses and evaluates the results obtained from the proposed implementation, while critically discussing the findings. It provides a detailed comparison of the proposed DRL strategies with traditional portfolio optimisation methods. Furthermore, it consists of an in-depth analysis of the model's decisions using post-hoc explainability techniques, in particular, feature importance, LIME and SHAP.

Chapter 5 discusses the legal, social, ethical and professional implications within the context of the project. By addressing these issues, the project aims to ensure that the proposed solution adheres to industry standards, while considering the implications of the technology.

Finally, the report concludes with a summary of the main points of the work, the contributions made, the results achieved as well as potential applications and future work in chapter 6.

Chapter 2

Background

This chapter provides an overview of the problem of portfolio optimisation in the financial domain, followed by a comprehensive explanation of the fundamentals of Deep Learning (DL) and Reinforcement Learning (RL), including the relevant algorithms in the field of Deep Reinforcement Learning (DRL). In addition, it discusses the need for explainability in Machine Learning (ML) and the relevant post-hoc explainability techniques to achieve interpretable and transparent models. Finally, it presents the state of the art in portfolio optimisation using DRL and the recent advancements in explainability techniques in the field.

2.1 Portfolio Optimisation

Portfolio optimisation is the process of selecting optimal weights for a portfolio of assets in order to maximise expected returns for a given level of risk, or conversely, to minimise risk for a given level of expected returns [22]. In mathematical terms, the problem requires finding a solution to the specified objective function, which is typically a function of the expected returns and the risk associated with the portfolio [23]. The task becomes further complicated if a time dimension is introduced, as the portfolio weights need to be adjusted over time to capture the changes in market conditions and asset prices [24].

2.1.1 Modern Portfolio Theory

There exist several traditional frameworks that formalise the problem of portfolio allocation. Markowitz's Modern Portfolio Theory (MPT) was proposed in 1952 [25] and it provides a mathematical framework where investors choose optimal portfolios based on risk and return, by either minimising the risk given a specified return or, maximising the return given a specified risk [26]. The theory extends the concept of diversification by suggesting that owning financial assets of different kinds is less risky than owning assets of the same kind, due to the correlations between assets.

The main assumptions in MPT are:

- investors are risk-averse, rational, and seek to maximise return for a given risk;
- returns are normally distributed;
- markets are frictionless, meaning there are no transaction costs; and
- assets are infinitely divisible.

Under these assumptions, portfolio risk and return can be modelled as an optimisation problem. Let $\mathbf{w} = (w_1, w_2, \ldots, w_N)^T$ denote the portfolio weight vector, where each w_i indicates the proportion of capital allocated to asset i, subject to the budget constraint:

$$\sum_{i=1}^{N} w_i = 1 \quad \Leftrightarrow \quad \mathbf{w}^T\mathbf{1} = 1 \tag{2.1}$$

with $\mathbf{1} \in \mathbb{R}^N$ being a vector of ones, and subject to the non-negativity constraint, meaning that short-selling is not allowed:

$$w_i \geq 0 \quad \forall i = 1, 2, \ldots, N. \tag{2.2}$$

Let $\boldsymbol{\mu} = (R_1, R_2, \ldots, R_N)^T$ represent the vector of expected returns, and $\Sigma \in \mathbb{R}^{N \times N}$ the covariance matrix of asset returns. The expected return of the portfolio is then given by:

$$R_p = \mathbf{w}^T \boldsymbol{\mu}, \tag{2.3}$$

and the portfolio risk is quantified by the variance of returns:

$$\sigma_p^2 = \mathbf{w}^T \Sigma \mathbf{w}. \tag{2.4}$$

This formulation provides the foundation for solving the mean-variance optimisation problem, by either:

- minimising portfolio variance σ_p^2 subject to a target expected return R_p, or
- maximising expected return R_p subject to a risk constraint σ_p.

The Markowitz mean-variance optimisation problem can be expressed as:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \mathbf{w}^T \Sigma \mathbf{w} \\ \text{subject to} \quad & \begin{cases} \mathbf{w}^T \boldsymbol{\mu} = R_p \\ \mathbf{w}^T \mathbf{1} = 1 \\ \mathbf{w} \geq 0 \end{cases} \end{aligned} \tag{2.5}$$

Solving for varying levels of target return leads to a set of optimal portfolios that form the efficient frontier. It is typically visualised in a risk-return space, where the x-axis represents the risk (standard deviation) and the y-axis represents the expected return, as shown in Figure 2.1. Portfolios below the curve are sub-optimal, while those on the frontier represent the best achievable combinations of risk and return.

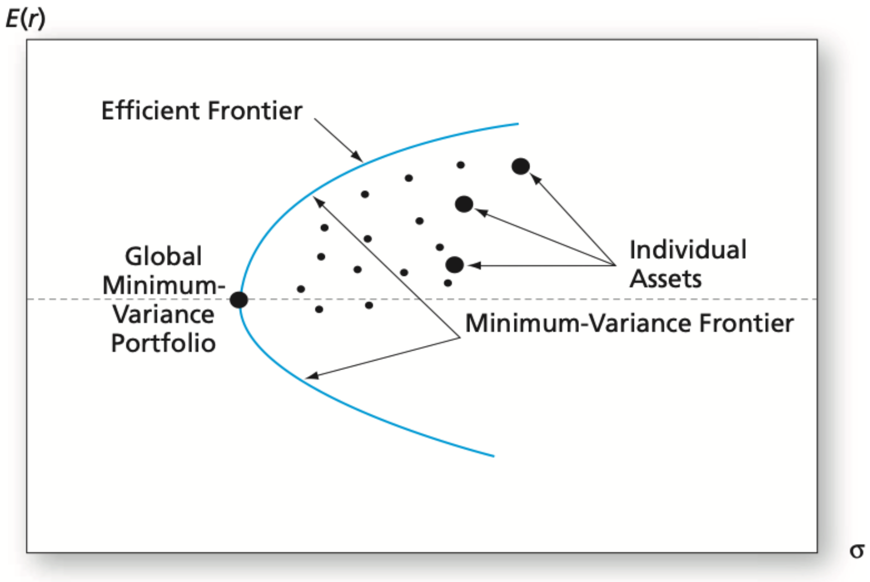

Figure 2.1: Efficient Frontier in Risk-Return Space. [1]

Despite the simplicity in the formulation of MPT, its assumptions do not reflect the behaviour of real markets. Modern financial markets are dynamic, non-stationary, and feature non-linear relationships, which have driven research into other approaches better suited to capture their complexities.

2.2 Deep Reinforcement Learning

Machine Learning is a branch of Artificial Intelligence (AI) that focuses on the use of data and algorithms to imitate the way humans learn by gradually improving their accuracy over time [27]. There are three main tasks in ML [28]:

- **Supervised Learning**: Task of training a classification or regression model from labelled data, where the model learns to map inputs to outputs based on examples.

- **Unsupervised Learning**: Task of drawing inferences from datasets consisting of unlabelled data, where the model learns to identify patterns or structures in the data.

- **Reinforcement Learning**: Task of training an agent to sequentially make decisions by taking actions in an environment with the goal of maximising cumulative reward, using feedback from the environment to learn an optimal strategy.

Deep Learning is a set of methods and techniques to solve such ML tasks, specially in supervised and unsupervised learning tasks. DL focuses on the use of Deep Neural Networks (DNNs) [29], which are characterised by a succession of layers of non-linear transformations that allow the model to learn a representation of the data with various levels of abstraction.

Therefore, Deep Reinforcement Learning (DRL) combines Deep Learning (DL) and Reinforcement Learning (RL) to solve sequential decision-making problems with high-dimensionality in the environment representation. This approach has gained significant attention in recent years due to its success in various applications, including robotics [30] and game playing [31, 32].

2.2.1 Reinforcement Learning

As mentioned, RL is a type of ML that solves the problem of sequential decision-making through continuous interaction with an environment. The agent learns to take actions given a representation of the environment's state with the goal of optimising a pre-defined notion of reward. The agent learns by successively adjusting its policy based on its observations and interactions with the environment.

The RL problem can be formalised as a discrete-time stochastic control process where an agent interacts with the environment. At each time step t, the agent observes the state of the environment $s_t \in \mathcal{S}$, takes an action $a_t \in \mathcal{A}$ to obtain a reward $r_t \in \mathbb{R}$ and transition to a new state $s_{t+1} \in \mathcal{S}$, where $\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space [28]. The agent's interaction with the environment is visually represented in Figure 2.2.

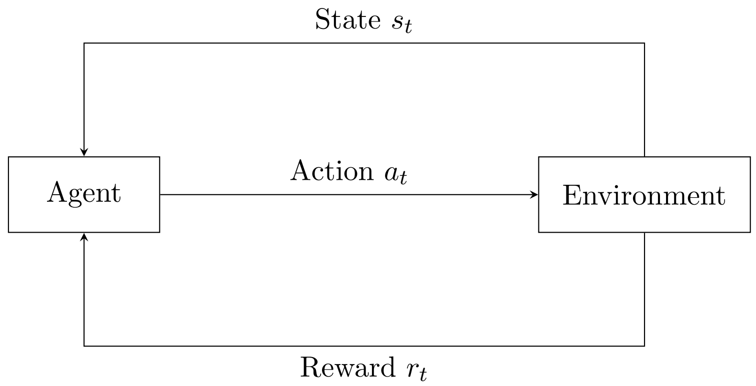

Figure 2.2: Agent interaction with environment

A discrete time stochastic control process can be formalised as a Markov Decision Process (MDP), if it fulfils the Markov Property.

Definition 2.2.1 (Markov Property)**.** A discrete time stochastic control process satisfies the Markov Property if:

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) \tag{2.6}$$

$$P(r_t|s_t, a_t) = P(r_t|s_t, a_t, s_{t-1}, a_{t-1}, \dots, s_0, a_0) \tag{2.7}$$

where $P(s_{t+1}|s_t, a_t)$ is the transition probability of moving to state s_{t+1} given the current state s_t and action a_t, and $P(r_t|s_t, a_t)$ is the reward function that defines the expected reward received at time t given the current state and action.

This implies that the state s_{t+1} at a future time step $t + 1$ only depends on the current state s_t and action a_t. Similarly, the reward r_t only depends on the current state and action and not on the history of previous states and actions. Consequently, a Markov Decision Process [33] is a discrete time stochastic control process defined as:

Definition 2.2.2 (Markov Decision Process)**.** An MDP is a tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, T, R, \gamma)$, where:

- $\mathcal{S}$ is the state space: $s_t \in \mathcal{S}$,
- $\mathcal{A}$ is the action space: $a_t \in \mathcal{A}$,
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ is the transition function,
- $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathcal{R}$ is the reward function, where $\mathcal{R} \in [0, R_{max}]$ is the set of all possible rewards bounded by $R_{max} \in \mathbb{R}^+$, and
- $\gamma \in [0, 1)$ is the discount factor.

At each time step, the probability of advancing to the next state s_{t+1} is given by the transition function $T(s_t, a_t, s_{t+1})$ and the reward r_t is given by the reward function $R(s_t, a_t, s_{t+1})$. This can be visualised in Figure 2.3.

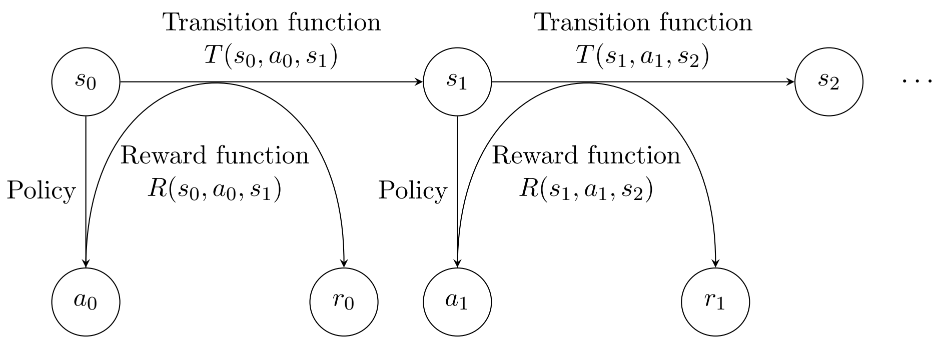

Figure 2.3: Markov Decision Process with policy, transition, and reward functions

The agent's objective is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maps states to actions, in order to maximise the expected cumulative reward over time. Policies can be categorised as:

- deterministic: $\pi(s) : \mathcal{S} \rightarrow \mathcal{A}$, at a given state s, the policy specifies the only available action to take, or
- stochastic: $\pi(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$, at a given state s, the policy specifies the probability of taking action a.

Markov Decision Process are based on the idea that the current state is fully representative of the environment. However, in most real world scenarios, the agent does not have access to the complete state. In such cases, a Partially Observable Markov Decision Process (POMDP) can be used to model the uncertainty in the agent's observations and actions.

The goal of the agent is to maximise the cumulative long-term reward G_t, which is defined as the sum of discounted rewards over time:

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = r_{t+1} + \gamma G_{t+1} \tag{2.8}$$

where $\gamma \in [0, 1)$ is the discount factor and is used to balance the importance between immediate and future rewards. If the discount factor is set to 0, the agent is myopic and only maximises the immediate reward; whereas, as γ approaches 1, the agent becomes more far-sighted and places greater importance on future rewards.

The expected cumulative reward is defined as the state value function $V^\pi(s) : \mathcal{S} \rightarrow \mathbb{R}$, which is the expected return when starting from state s and following policy π:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t | s_t = s \right] \tag{2.9}$$

where $\mathbb{E}_\pi$ denotes the expectation over the policy π.

Similarly, the state-action value function $Q^\pi(s, a) : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ is defined as the expected return when starting from state s, taking action a, and then following policy π:

$$Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[G_t | s_t = s, a_t = a\right] \tag{2.10}$$

The state value function $V^{\pi}(s)$ and the state-action value function $Q^{\pi}(s,a)$ are related as follows:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) Q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[Q^{\pi}(s,a) \mid s_t = s\right] \tag{2.11}$$

Moreover, the advantage function $A^{\pi}(s,a)$ combines both the state value function $V^{\pi}(s)$ and the state-action value function $Q^{\pi}(s,a)$, and is defined as:

$$A^{\pi}(s,a) = Q^{\pi}(s,a) - V^{\pi}(s) \tag{2.12}$$

A policy π is said to be optimal if the policy's value function is the optimal value function of the MDP, defined as:

$$V^{*}(s) = \max_{\pi'} V^{\pi'}(s), \forall s \in \mathcal{S} \tag{2.13}$$

$$Q^{*}(s,a) = \max_{\pi'} Q^{\pi'}(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A} \tag{2.14}$$

The optimal policy π^{*} is:

$$\pi^{*}(s) = \arg\max_{a \in \mathcal{A}} Q^{*}(s,a) \tag{2.15}$$

As a result, the optimal policy is the greedy policy that performs the optimal actions at each time step as determined by the optimal value functions. This framework enables the agent to determine optimal actions that maximise long-term returns by evaluating immediate information, without requiring knowledge of the values of future states and actions.

The Bellman equations [34] provide a recursive relation between the value functions in terms of the future state/action values. There are four main Bellman equations, classified in two groups: the Bellman expectation equations and the Bellman optimality equations. The Bellman expectation equations are defined as follows:

$$V^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi \left(a \mid s\right) \sum_{s' \in \mathcal{S}} T\left(s, a, s'\right) \left[R\left(s, a, s'\right) + \gamma V^{\pi}(s')\right] \tag{2.16}$$

$$Q^{\pi}(s, a) = \sum_{s' \in \mathcal{S}} T\left(s, a, s'\right) \left[R\left(s, a, s'\right) + \gamma \sum_{a' \in \mathcal{A}} \pi\left(a' \mid s'\right) Q^{\pi}(s', a')\right] \tag{2.17}$$

and the Bellman optimality equations are defined as:

$$V^{*}(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} T\left(s, a, s'\right) \left[R\left(s, a, s'\right) + \gamma V^{*}(s')\right] \tag{2.18}$$

$$Q^{*}(s, a) = \sum_{s' \in \mathcal{S}} T\left(s, a, s'\right) \left[R\left(s, a, s'\right) + \gamma \max_{a' \in \mathcal{A}} Q^{*}(s', a')\right] \tag{2.19}$$

Although explicitly solving the Bellman equations would lead to the optimal policy, it is often intractable due to the size of the state and action spaces. Therefore, in RL algorithms, the goal is to learn an approximation of the optimal value functions, which can be used to derive the optimal policy. Another problem that arises is that of balancing exploration and exploitation [35]. Theoretically, following the greedy action yields the optimal policy, but this is only true if all the action values are known. In practice, the agent at each time step and given state chooses either an action whose value is higher, thus exploiting its current knowledge, or picks an action at random, thus exploring the environment and gaining more information about the state-action space, leading to potentially discovering a better action than the greedy one.

2.2.2 Deep Reinforcement Learning Algorithms

A Reinforcement Learning (RL) agent includes one or more of the following components [28]:

- a representation of the value function that provides a prediction of the value of each state or state-action pair,
- a direct representation of the policy, and
- a model of the environment, consisting of estimates of transition and reward functions.

Depending on the components, the main RL paradigms are:

- **Model-free** algorithms do not learn a representation of the environment, but focus on the value function, the policy or both. These algorithms can be further divided into:
  - **Value-based** algorithms learn an approximation of the value function, which is used to compute the state or state-action values. The policy is not learnt explicitly but can be derived from the value function. Examples include Deep Q-Network (DQN) [36] and C51 [37].
  - **Policy-based** algorithms learn a direct representation of the policy, which is used to select actions. Advantage Actor-Critic (A2C), Asynchronous Advantage Actor-Critic (A3C) [38] and Proximal Policy Optimisation (PPO) [39] are examples of policy-based algorithms.
  - **Actor-Critic** algorithms combine both value-based and policy-based approaches, where the actor learns the policy and the critic learns the value function. The critic provides feedback to the actor to improve the policy.

Some examples are Deep Deterministic Policy Gradient (DDPG) [40], Twin Delayed Deep Deterministic Policy Gradient (TD3) [41], and Soft Actor-Critic (SAC) [42].

- **Model-based** algorithms include a model of the environment, which can be used to simulate future states and rewards. For example, World Models [43] learns a model of the environment and AlphaZero [44] has a representation of the model.

The taxonomy of the RL algorithms is shown in Figure 2.4.

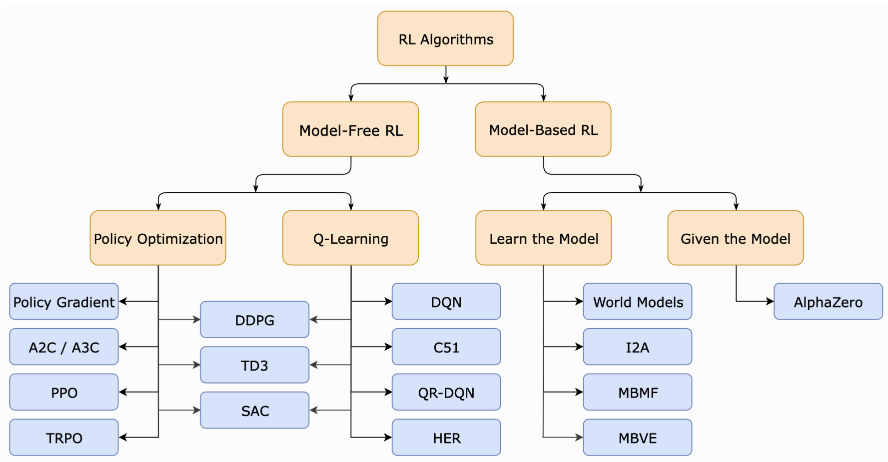

Figure 2.4: Taxonomy of Reinforcement Learning Algorithms [2]

The full potential of RL algorithms is achieved when leveraging the power of Deep Learning to solve dynamic stochastic control problems that have high-dimensionality in their representation of the state and action spaces. DRL algorithms use Deep Neural Networks to approximate the value functions, or use gradient ascent to find the optimal policy parameters. This thesis will focus on policy-based and actor-critic algorithms.

2.2.2.1 Advantage Actor-Critic (A2C)

The Advantage Actor-Critic (A2C) algorithm was developed by Mnih et al. (2016) in their paper on *Asynchronous methods for deep reinforcement learning* [38]. The main contribution is the usage of the advantage function to address the variance issues present in policy gradient methods. The A2C is the synchronous version of A3C, and is preferred due to its better performance in terms of training time and cost-effectiveness [45].

The algorithm consists of a dual-network architecture, where the actor network learns a stochastic policy $\pi\left(a_t \mid s_t; \theta\right)$ and the critic network learns the value function $V\left(s_t; \theta_v\right)$. The policy and the value function are updated after every t_{max} actions or when a terminal state is reached. The advantage function is estimated as follows:

$$A\left(s_t, a_t; \theta, \theta_v\right) = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V\left(s_{t+k}; \theta_v\right) - V\left(s_t; \theta_v\right) \tag{2.20}$$

where k represents the n-step return and is upper-bounded by the maximum number of steps t_{max}, and γ is the discount factor. The algorithm's objective function is defined as:

$$J\left(\theta, \theta_v\right) = \mathbb{E}_\pi\left[\log \pi\left(a_t \mid s_t; \theta\right) A\left(s_t, a_t; \theta, \theta_v\right)\right] \tag{2.21}$$

The pseudo-code for the algorithm for A2C is outlined in Appendix **??**.

2.2.2.2 Proximal Policy Optimisation (PPO)

Proximal Policy Optimisation (PPO) is a policy gradient algorithm that was introduced by Schulman et al. (2017) in their paper on *Proximal Policy Optimization Algorithms* [39] with the objective of constraining policy updates. The algorithm balances sufficiently large updates in order to improve the policy, while avoiding excessively large changes that could hinder performance.

PPO improves the performance of Trust Region Policy Optimisation (TRPO) [46] by using a clipped surrogate objective function that ensures that the probability ratio r_t is bounded within a range of $[1-\epsilon, 1+\epsilon]$, where ϵ is a hyperparameter that controls the clipping range. The objective function is defined as:

$$L_t^{CLIP}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t \right) \right], \tag{2.22}$$

where

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \tag{2.23}$$

is the probability ratio between the current policy π_θ and the old policy $\pi_{\theta_{old}}$

Another improvement with respect to TRPO is the use of simple first-order optimisation methods as opposed to the second-order methods used in TRPO. Consequently, PPO maintains the stability and reliability of other trust-region methods, while being easier to implement and more computationally efficient. The pseudo-code for the algorithm for PPO is outlined in Appendix **??**.

2.2.2.3 Deep Deterministic Policy Gradient (DDPG)

Deep Deterministic Policy Gradient (DDPG) is an off-policy actor-critic algorithm that was introduced by Lillicrap et al. (2015) in their paper on *Continuous control with deep reinforcement learning* [40]. The algorithm arises to solve the challenges of applying Deep Q-Network [36] to continuous action spaces by applying Deterministic Policy Gradient (DPG), which enables the efficient computation of the policy gradient without the need to integrate over the action space.

The algorithm uses the following networks: the actor network $\mu(s_t; \theta_\mu)$, which learns a deterministic policy, the critic network $Q(s_t, a_t; \theta_Q)$, which learns the state-action value function, and the actor's and critic's target networks. The actor network is updated using DPG:

$$\nabla_{\theta_\mu} J \approx \mathbb{E}_{s_t \sim \mathcal{D}} \left[\nabla_a Q(s_t, a_t; \theta_Q) \nabla_{\theta_\mu} \mu(s_t; \theta_\mu) \right], \tag{2.24}$$

where $\mathcal{D}$ is the replay buffer that stores the agent's experiences, θ_μ are the parameters of the actor network, and θ_Q are the parameters of the critic network, and the critic network is updated using the Bellman equations:

$$\nabla_{\theta_Q} J \approx \mathbb{E}_{s_t \sim \mathcal{D}} \left[\left(r_t + \gamma Q(s_{t+1}, \mu(s_{t+1}; \theta_\mu); \theta'_Q) - Q(s_t, a_t; \theta_Q) \right) \nabla_{\theta_Q} Q(s_t, a_t; \theta_Q) \right], \tag{2.25}$$

where θ'_Q are the parameters of the target critic network.

From Deep Q-Network, the algorithm incorporates a replay buffer to store the agent's experiences, which allows the agent to learn from past experiences and improve its performance over time. Moreover, DDPG incorporates noise, typically Ornstein-Uhlenbeck noise [47], to the actions taken by the actor network to encourage exploration of the action space. The pseudo-code for the algorithm for DDPG is outlined in Appendix **??**.

2.2.2.4 Twin Delayed Deep Deterministic Policy Gradient (TD3)

Twin Delayed Deep Deterministic Policy Gradient (TD3) is an extension of DDPG introduced by Fujimoto et al. (2018) in their paper on *Addressing Function Approximation Error in Actor-Critic Methods* [41]. The proposal addresses the hyper-parameter sensitivity and overestimation bias present in the critic network of DDPG. The main improvements are:

- **Clipped Double Q-Learning**: The algorithm employs two critic networks and uses their minimum value to compute the target value for the actor network, which reduces the overestimation bias.
- **Delayed Policy Updates**: The actor and critic networks updates are performed

at different frequencies. The critic networks are updated every time step, whereas the actor and target networks are updated less frequently. The main benefit is that it allows the critic to improve the accuracy of the value estimates before the actor updates its policy.

- **Target Policy Smoothing**: The algorithm adds noise to the target action during the critic updates to smooth the value function over similar actions, resulting in a lower impact of approximation errors and more robust policies.

The pseudo-code for the algorithm for TD3 is outlined in Appendix **??**.

2.2.2.5 Soft Actor-Critic (SAC)

Soft Actor-Critic (SAC), despite being an off-policy actor-critic algorithm, represents a paradigm shift in RL. The algorithm was presented by Haarnoja et al. (2018) in their paper on *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor* [42] and is based on the maximum entropy framework, which aims to maximise both the expected return and the entropy of the policy, encouraging the agent to succeed at the task while acting as randomly as possible.

The algorithm uses two critic networks to estimate the state-action value function, and a stochastic actor network that learns a policy that maximises the expected return while also maximising the entropy of the policy. The critic networks are updated using the Bellman equations, and the actor network is updated using the policy gradient method. The objective function for the actor network is defined as:

$$J(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\mathbb{E}_{a_t \sim \pi_\theta} \left[\alpha \log \pi_\theta(a_t|s_t) + Q(s_t, a_t; \theta_Q) \right] \right], \tag{2.26}$$

where α is a temperature parameter that controls the trade-off between exploration and exploitation, and $\mathcal{D}$ is the replay buffer that stores the agent's experiences. The critic networks are updated using the Bellman equations:

$$J(\theta_Q) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\left(r_t + \gamma \min_{i=1,2} Q(s_{t+1}, \mu_\theta(s_{t+1}) + \mathcal{N}(0,\sigma); \theta_{Q_i}) - Q(s_t, a_t; \theta_{Q_i}) \right)^2 \right], \tag{2.27}$$

where $\mathcal{N}(0,\sigma)$ is the noise added to the action to encourage exploration, and θ_{Q_i} are the parameters of the two critic networks. The temperature parameter α is learned automatically by maximising the entropy of the policy, which is defined as:

$$J(\alpha) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\alpha \log \pi_\theta(a_t|s_t) \right]. \tag{2.28}$$

This parameter is used to control the trade-off between the entropy and the reward terms, effectively controlling the stochasticity of the policy.

The pseudo-code for the algorithm for SAC is outlined in Appendix **??**.

2.2.2.6 Algorithm comparison

The five algorithms presented above are among the most widely used in the field of DRL, each addressing specific challenges in policy estimation and value function approximation. A2C is an on-policy actor-critic method that reduces variance through advantage estimation, while PPO changed on-policy learning by introducing a clipped surrogate objective that ensures stable policy updates. Although DDPG pioneered continuous control through deterministic policies, TD3 addressed the overestimation bias of its predecessor via twin critics and delayed updates. SAC revolutionised the field of DRL by incorporating maximum entropy principles.

An overview of the algorithms and their key characteristics is summarised in Table 2.1.

Algorithm	**Policy Type**	**On/Off Policy**	**Key Idea**
A2C	Stochastic	On-Policy	Advantage Function Estimation
PPO	Stochastic	On-Policy	Clipped Policy Updates
DDPG	Deterministic	Off-Policy	Actor-Critic + Replay buffer
TD3	Deterministic	Off-Policy	Double critics + Delayed updates
SAC	Stochastic	Off-Policy	Max-entropy RL

Table 2.1: Comparison of Deep Reinforcement Learning Algorithms

2.3 Explainable Artificial Intelligence

Explainable Artificial Intelligence (XAI) refers to a set of processes, methods and techniques that enable human users to comprehend and trust the outcomes and decisions made by Artificial Intelligence (AI) systems. The goal is to make AI algorithms more transparent, interpretable, and accountable, allowing end-users to understand the outputs of the models. This is particularly important in Deep Learning (DL) where the models are often considered black boxes due to their complexity, non-linearity and high-dimensionality, making it difficult to understand how they arrive at their decisions. With explainable systems, the benefits are numerous ranging from informed decision making to increased user adoption and better governance [48].

Although there are many possible classifications of XAI methods, they can be broadly categorised into two main categories. First, transparent algorithms are inherently understandable and interpretable, such as linear regression, decision trees and rule-based systems. Second, post-hoc explanations are methods that require the usage of an additional algorithm to clarify the decisions made by a model. Examples include saliency maps [49] and interaction data [50].

Another classification relates to whether the explanation method depends on the type of model it is being applied to. On the one hand, model-agnostic methods can be applied to any model regardless of its internal architecture, effectively treating all types of models as black boxes. The analysis is conducted by understanding the input-output behaviour and how small perturbations to the input impact the output. Examples include Local Interpretable Model-agnostic Explanations (LIME) [51] and SHapley Additive exPlanations (SHAP) [52]. On the other hand, model-specific methods are tailored to a particular architecture and leverage its components as part of the explanation process. In the case of neural networks, this can be achieved by analysing the learned features or by incorporating attention mechanisms [53]. The model-agnostic methods can be further sub-divided between global and local explanations. The goal of global explainability is to provide an understanding of the model's overall behaviour across an entire dataset, while local explainability focuses on explaining the individual predictions.

The taxonomy of the XAI methods is shown in Figure 2.5.

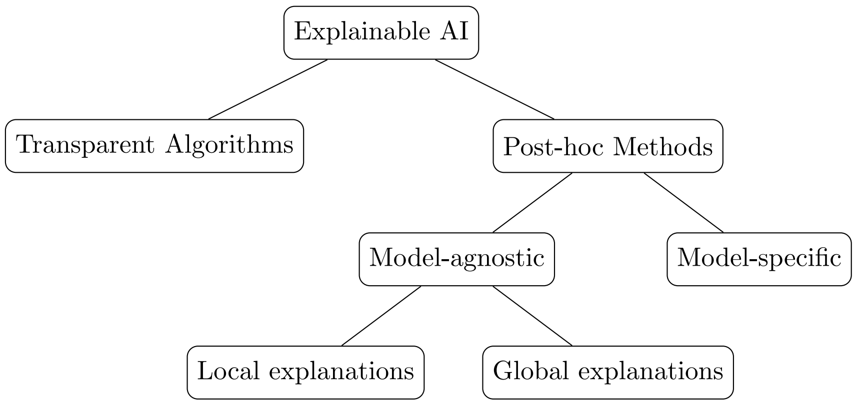

Figure 2.5: Taxonomy of Explainable Artificial Intelligence Methods [3]

2.3.1 Feature Importance

Feature Importance is a global model-specific method that quantifies the contribution of each feature to the model's predictions. In particular, permutation feature importance [54] measures the contribution of each feature by evaluating the model's performance on the original dataset and comparing it to the performance on a permuted version of the dataset, where a specific feature is randomly shuffled. The process allows to understand how much the model's relies on a particular feature for its prediction, by measuring the decrease in predictive power if the input feature's values vary.

An alternative method, well-suited for tree-based models, is impurity-based feature importance. Random forests split their features with the goal of reducing an impurity measure at each node, normally Gini impurity for classification tasks or Mean Squared Error (MSE) for regression. As such, a variable responsible for a split with a large decrease in impurity is considered important [55]. As a result, the impurity importance of a feature is the sum of all impurity reductions across all nodes and trees in the forest where the particular feature was used to split the data. While powerful, this method suffers from bias towards features with high cardinality and they do not necessarily reflect the ability of a feature to make useful predictions on the test set, given that importance is computed on the training set.

2.3.2 Local Interpretable Model-agnostic Explanations

Local Interpretable Model-agnostic Explanations (LIME) is a model-agnostic explanation method that interprets individual predictions of a Machine Learning model by analysing the model locally around the prediction of interest. The method, introduced by Ribeiro et al. (2016) in their paper on *Why should I trust you? Explaining the predictions of any classifier* [51], utilises the black box model to understand what happens to the outputs when the input data is slightly modified, then fits a simpler, interpretable model to the perturbed data to approximate the black box model's behaviour in that local region.

For a more rigorous definition, let $\mathcal{L}(f, g, \pi_x)$ be a measure of how unfaithful an explanation model g is in approximating the model $f : \mathbb{R}^d \to \mathbb{R}$ in the neighbourhood of the instance x defined as π_x. The goal is to find an interpretable model $g \in G$ that minimises the following objective function:

$$\xi(x) = \mathcal{L}(f, g, \pi_x) + \Omega(g) \tag{2.29}$$

where $\Omega(g)$ is a regularisation term that penalises the complexity of the explanation model g. This encourages interpretability, meaning a qualitative understanding of the relationship between inputs and outputs, and promotes local fidelity, ensuring the explanation accurately approximates the model's behaviour in that region.

The method works as follows:

- **Instance Selection**: Start with a specific prediction instance that requires explanation.
- **Perturbation**: Generate synthetic data points by perturbing the original instance.
- **Model Querying**: Obtain predictions from the black box model for these perturbed instances.
- **Weighting**: Assign weights to the synthetic data points based on their proximity to the original instance.
- **Surrogate Training**: Train a simple, interpretable model (typically linear regression) on the weighted synthetic dataset.

- **Explanation**: Use the surrogate model to explain the original prediction by interpreting its coefficients or structure.

Its main advantages are its ability to work across different types of data and models, due to its model-agnostic nature, and the fact that its explanations are human-friendly and include a fidelity measure that indicates how well the explanation approximates the black box model's local behaviour. Nonetheless, the method offers only local explanations, which may not generalise well to the entire dataset; it relies on the choice of neighbourhood; and the complexity of the surrogate model needs to be defined in advance and can compromise the interpretability of the explanation [3].

2.3.3 Shapley Additive Explanations

Another model-agnostic technique is SHapley Additive exPlanations (SHAP), which was introduced by Lundberg and Lee (2017) in their paper on *A unified approach to interpreting model predictions* [52]. The method is based on cooperative game theory and the concept of Shapley values [56].

The Shapley value arises in cooperative game theory to answer the question of how to fairly distribute the contribution of each player in a coalition. In this framework, a coalitional game is represented as a tuple (N, v), where $N = \{1, 2, \ldots, n\}$ is the set of players and $v : 2^N \rightarrow \mathbb{R}$ is the characteristic function that assigns a value $v(S)$ to each possible coalition $S \subseteq N$, with the convention that $v(\emptyset) = 0$. The characteristic function $v(S)$ represents the total value that coalition S can achieve through cooperation.

Mathematically, the Shapley value for a feature i is defined as:

$$\phi_i = \sum_{S \subseteq N \setminus \{i\}} \frac{|S|!(|N| - |S| - 1)!}{|N|!} \left[v_{S \cup \{i\}}(x_{S \cup \{i\}}) - v_S(x_S) \right] \tag{2.30}$$

where $S \subseteq N$ is a subset of all features N, $v_{S \cup \{i\}}$ is marginal contribution of player i

to coalition S, and v_S is the value of coalition S without player i. This formula can be understood as computing the weighted average of marginal contributions of feature i across all possible coalitions.

The advantage of Shapley values is that it is the only attribution method that results in a fair payout. In particular, it satisfies the four fundamental properties that define a fair allocation:

- **Efficiency:** The sum of all Shapley values equals the value of the grand coalition:

$$\sum_{i \in N} \phi_i = v(N)$$

- **Symmetry:** The contributions of two players i and j should be equal if they make identical marginal contributions to all possible coalitions:

$$v(S \cup \{i\}) = v(S \cup \{j\}), \forall S \subseteq N \setminus \{i, j\} \implies \phi_i = \phi_j$$

- **Dummy:** If a player contributes nothing to any coalition they join, their Shapley value should be zero:

$$v(S \cup \{i\}) = v(S), \forall S \subseteq N \setminus \{i\} \implies \phi_i = 0$$

- **Additivity:** For two games (N, v_1) and (N, v_2), the Shapley value of the combined game $(N, v_1 + v_2)$ equals the sum of individual Shapley values:

$$\phi_i(v_1 + v_2) = \phi_i(v_1) + \phi_i(v_2)$$

The algorithm for approximating Shapley values is outlined in Appendix 6. However, the method is computationally expensive, as it requires evaluating the model for all possible coalitions of features.

The use of Shapley values in ML was not introduced until 2011, when Štrumbelj and Kononenko [57] proposed it as a method for explaining black-box regression models. However, it was popularised in 2017 by Lundberg and Lee [52], who introduced the SHAP framework. Although their proposal relies on the principles of Shapley values, their main contribution is to represent the Shapley value as an additive feature attribution method. The explanation is given as:

$$g(\mathbf{z}') = \phi_0 + \sum_{j=1}^{M} \phi_j z'_j \tag{2.31}$$

where g is the explanation model, $\mathbf{z}' \in \{0,1\}^M$ is the coalition vector, M is the maximum coalition size, ϕ_0 is the bias term, and ϕ_j is the feature attribution for feature j.

2.3.3.1 KernelSHAP

The KernelSHAP algorithm builds on the SHAP framework by using a kernel-based approach to estimate Shapley values. It approximates the Shapley value by sampling different coalitions and using a weighted linear regression model to fit the contributions of each feature. The method relies on not all coalitions contributing equally to the final Shapley value and as such uses kernel weights to identify the most important coalitions. The main steps of the KernelSHAP algorithm are:

- Sample a set of coalitions from the feature space: $\mathbf{z}'_k \in \{0,1\}^M, k \in \{1,\ldots,K\}$, where K is the total number of samples.
- For each coalition $\mathbf{z}'_k$, compute the model's prediction by first converting $\mathbf{z}'_k$ to the original feature space and then applying the model $\hat{f} : \hat{f}\left(h_{\mathbf{x}}\left(\mathbf{z}'_k\right)\right)$.
- Compute the weight for each coalition with the SHAP kernel: $\pi_{\mathbf{x}}(\mathbf{z}')$
- Fit a weighted linear regression model to the sampled coalitions and their corre-

sponding predictions.

- Return Shapley values ϕ_k as the coefficients of the fitted model.

The sample coalitions are generated by randomly selecting subsets of the features, or equivalently, a vector of 0s and 1s indicating whether a feature is included in the coalition. The sampled coalitions represent the dataset for the regression model whose target is the prediction for a coalition. However, the sampled coalitions are not on the target feature space and, as such, it is necessary to implement another function $h_{\mathbf{x}}(\mathbf{z}') = \mathbf{z}$ that maps the sampled coalitions to the original feature space. In the case of tabular data, this is done by replacing the features that are not included in the coalition with their mean value or a random sample from its possible values. As a result, sampling from the marginal distribution ignores the dependence structure between features.

Although there are similarities between SHAP and LIME, their main difference lies in feature weighting in the regression model. The SHAP weighting assumes that small and large coalitions provide the most information about its isolated effects or its total effects, respectively. Whereas coalitions with half the features add little information about a specific feature's contributions. The proposed weighting function for KernelSHAP is:

$$\pi_{\mathbf{x}}(\mathbf{z}') = \frac{(M-1)}{\binom{M}{|\mathbf{z}'|}|\mathbf{z}'|\,(M-|\mathbf{z}'|)}, \tag{2.32}$$

where M is the maximum coalition size and $|\mathbf{z}'|$ is the number of features in the coalition $\mathbf{z}'$.

In addition, since the coalitions with the smallest and the largest number of features are the most informative, the sampling process is biased towards coalitions of size 1 and $M-1$, resulting in $2M$ possible coalitions. Then, the remaining budget $K-2M$ is used to sample coalitions of size 2 to $M-2$. This process continues until the sampled coalitions reach the desired number of samples.

Consequently, given the samples, the KernelSHAP algorithm fits a weighted linear regression model to estimate the Shapley values. The model is defined as:

$$g(\mathbf{z}') = \phi_0 + \sum_{j=1}^{M} \phi_j z'_j. \tag{2.33}$$

The goal is to minimise the following loss function:

$$\mathcal{L}(\hat{f}, g, \pi_{\mathbf{x}}) = \sum_{\mathbf{z}' \in \mathcal{Z}} \left[\hat{f}\left(h_{\mathbf{x}}\left(\mathbf{z}'\right)\right) - g(\mathbf{z}')\right]^2 \pi_{\mathbf{x}}(\mathbf{z}'), \tag{2.34}$$

where $\mathbf{Z}$ is the dataset of sampled coalitions.

2.3.3.2 TreeSHAP

A variant of their original proposal is TreeSHAP [58], designed specifically for tree-based models, such as decision trees, random forests or gradient-boosted trees. In this case, the method is model-specific and by leveraging the structure of the tree, it improves the computational efficiency by reducing computation time from exponential for KernelSHAP to polynomial.

The method works by exploiting the tree structure and the main steps of the algorithm are as follows:

- For each feature, traverse the tree and compute the contribution of the feature to the prediction at each node.
- For each node, compute the contribution of the feature to the prediction by considering the difference between the prediction at the node and the prediction at the parent node.
- For each feature, compute the Shapley value by averaging the contributions across

all nodes in the tree.

2.4 State of the Art of Deep Learning and Explainability for Portfolio Optimisation

The emergence of Machine Learning (ML) and its rise in popularity have led to a new research direction in various financial applications, due to its ability to learn complex patterns from high-dimensional data and adapt to changing market conditions. Particularly, the main approaches in the financial field include the use of Deep Learning and Reinforcement Learning (RL) for price trend prediction [59] and portfolio optimisation [60].

The topic of portfolio optimisation with multiple risky financial assets has been extensively studied and it still poses a challenging task due to the complex and stochastic nature of financial markets. Traditional approaches to portfolio optimisation, such as the Mean-Variance Optimisation (MVO) [25], have been widely used, but they often rely on assumptions that do not hold in practice, such as the normality of returns and the stability of the covariance matrix. As a result, these methods can lead to suboptimal portfolios and poor performance in real-world scenarios.

The use of Deep Learning architectures is particularly prominent in the context of price prediction, where deep neural networks are employed to learn complex patterns in historical price data. For instance, Long Short-Term Memory (LSTM) architectures are particularly well-suited for financial time series forecasting due to their ability to capture long-term dependencies and temporal patterns in the data. The performance of these architectures can lead to a Mean Absolute Percentage Error (MAPE) as low as 2.72%, as demonstrated by Chaudhary (2025) [61]. Shen et al. (2020) [7] also propose a LSTM architecture for short-term trend prediction. Despite its simple architecture, its main contributions rely on the feature engineering process. The proposal incorporates feature extension, followed by recursive feature elimination and finally, performs Principal Component Analysis (PCA) to reduce the dimensionality of the input data for the LSTM model. The use of PCA improves the training efficiency of the architecture by 36.8% [7].

The use of ensemble methods is also prominent for price prediction, where techniques such as stacking, blending, boosting and bagging are employed to combine the predictions of multiple models. For example, Nti et al. (2020) [11] explore twenty-five different ensemble classifiers and regressors based on Decision Trees, Support Vector Machines and Neural Networks. Their research shows that although stacking and boosting ensemble algorithms provide better results in terms of accuracy, they are the most computationally expensive.

When it comes to portfolio optimisation, DRL algorithms are particularly better suited as they address the sequential decision making nature of portfolio management. Unlike traditional Supervised Learning approaches that require labelled data, DRL enables agents to learn optimal investment strategies through direct interaction with market environments. This paradigm allows for the dynamic adjustment of portfolio weights without the need for predefined rules, making it suitable to capture market non-linearities and adapt to changing conditions. There are numerous DRL algorithms, as outlined in Section 2.2.2, with each being particularly befitting to different aspects of the portfolio management process, namely, DDPG encourages maximum returns, while A2C reduces the variance.

Liu et al. (2018) [15] propose the use of DDPG for profitable stock trading, where the agent learns to buy, hold and sell stocks based on the historical price data with the goal of maximising the investment return. Their proposal outperforms the Dow Jones Industrial Average (DJIA) and the min-variance portfolio allocation strategies, achieving an annualised return of 25.86% and a Sharpe ratio of 1.79. Their work is further extended by Liu (2020) [62] by proposing a Python library, FinRL, which provides a comprehensive framework for developing and evaluating DRL algorithms in the context of financial trading. Although the library provides implementation for numerous DRL algorithms including DQN, A2C and SAC, in this paper, they only evaluate the performance of TD3 and DDPG on the Dow Jones Industrial Average (DJIA) constituents for the tasks of multiple stock trading and portfolio allocation. Notwithstanding, an interesting aspect of their work is the incorporation of the turbulence index, that measures extreme asset price fluctuations, to control portfolio risk. Beyond traditional assets, optimal portfolio allocation has also been explored in the context of cryptocurrencies [63] and future contracts [64].

Despite the significant applications of DRL in portfolio optimisation, the field still faces significant challenges that prevent its widespread adoption. When it comes to market conditions, identifying the correct objective function that guarantees both efficiency and accuracy remains an open research question. Finally, one of the most significant concerns is the lack of transparency of ML models. If there is to be widespread use of these models in the financial field, great strides must be made in the area of explainability.

Barriedo et al. (2020) [18] provide a comprehensive overview of Explainable Artificial Intelligence (XAI) methods and their need in different fields. In 2024, the European Union has passed the Regulation 2024/1689, commonly known as the AI Act [65]. The goal is to ensure that the AI systems used in the European Union are safe, transparent and traceable. Given the regulatory requirements, it is crucial for financial institutions to adopt techniques that enhance the interpretability and traceability of ML models.

The application of XAI methods has gained traction in the financial domain, with notable use cases including stock market trend prediction [66] and auditing [67]. However, despite examples of XAI methods in finance, there is still a lack of research in DRL. An interesting study was conducted by González Cortés et al. (2024) [17], whose authors address the black box behaviour of DRL models by proposing an intrinsically transparent algorithm. They use Advantage Actor-Critic (A2C) [1] enhanced with an attention-layered LSTM to determine the importance of the input features. Although their results show an improvement in performance compared against two variants of the Markowitz model, their implementation does not scale well with the number of assets in the portfolio, as the attention mechanism is independent for each stock.

Another recent study by de-la-Rica-Escudero et al. (2025) [68] proposes post-hoc explainability methods to interpret the decisions of PPO agent optimised for portfolio management. They implement three model-agnostic explainability techniques: SHAP, LIME and feature importance analysis. Despite their proposal's ability to provide insights into the model's decision-making process, they do not use the model prediction function directly but instead implement a surrogate model to approximate the decision boundary of the PPO agent, effectively adding an additional layer to the model.

Given the lack of research in the area of explainability for DRL models that perform automated portfolio allocation and the current limitations of existing methods, this thesis proposes to provide a comprehensive study of DRL techniques tailored for portfolio optimisation in financial markets, by exploring the potential of five prominent algorithms (A2C, PPO, DDPG, TD3 and SAC) across various market conditions and asset classes. Moreover, due to the complex and hidden nature of those algorithms, their decision-making process will be explained using XAI methods. The objective is to bridge the gap between the performance of DRL and algorithmic transparency.

[1] In the paper [17], they refer to A2C as Synchronous Advantage Actor Critic (SAAC)

Chapter 3

Methodology

This chapter covers the methodology and framework established to provide an explainable Deep Reinforcement Learning (DRL) model capable of optimising a portfolio of financial assets. The chapter is structured as follows: first, it describes the architecture and components of the proposed DRL model, including the state representation, reward function and training process. Second, it discusses the evaluation metrics and experimental setup used to assess the performance of the proposed solution. Finally, it outlines the implementation of the explainability techniques used to interpret the model's decisions.

3.1 Problem Definition

The problem of portfolio optimisation is the task of finding an optimal allocation of financial assets in a portfolio to maximise expected returns while minimising risk. Thus, it is necessary to decide how to rebalance the portfolio at each time step in a highly stochastic and complex financial market. This can be formulated using a Markov Decision Process (MDP) framework, where the agent interacts with the environment by deciding the optimal allocation based on the state of the environment at each time step to maximise the expected cumulative reward over time. Deep Reinforcement Learning (DRL) gives the agent the ability to learn the optimal policy directly from the environment by taking actions and receiving rewards.

3.2 Markov Decision Process Model

Due to the dynamic, stochastic and interactive nature of financial markets, a Markov Decision Process is a suitable framework to model the problem. The main elements of the MDP model are defined as follows:

- State space $\mathcal{S}$
- Action space $\mathcal{A}$
- Reward function $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$
- Transition function $T : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$
- Discount factor $\gamma \in [0, 1)$

The state space $\mathcal{S}$ is a vector representation of the financial environment. For a portfolio of D assets, the features that describe the state include asset prices, technical indicators and macroeconomic indicators:

- Close price $\mathbf{p}_t \in \mathbb{R}^D$: Adjusted close prices of the assets at time t.
- Open price $\mathbf{o}_t \in \mathbb{R}^D$: Opening prices of the assets at time t.
- High price $\mathbf{h}_t \in \mathbb{R}^D$: Highest prices of the assets at time t.
- Low price $\mathbf{l}_t \in \mathbb{R}^D$: Lowest prices of the assets at time t.
- Volume $\mathbf{v}_t \in \mathbb{R}^D$: Trading volume of the assets at time t.

- Technical indicators $\mathbf{I}_t \in \mathbb{R}^{D \times I}$: For each of the D assets, a vector $\mathbf{i}_t$ of I technical indicators, such as moving averages, relative strength index (RSI), and Bollinger Bands, calculated from the asset prices.

- Macroeconomic indicators $\mathbf{M}_t \in \mathbb{R}^{D \times M}$: For each of the D assets, a vector $\mathbf{m}_t$ of M macroeconomic indicators, such as volatility index and interest rates, which provide additional context about the financial environment.

- Covariance matrix $\mathbf{C}_t \in \mathbb{R}^{D \times D}$: For each of the D assets, a vector of D values representing the covariance between the assets in the portfolio at time t.

The description of technical and macroeconomic indicators is provided in more detail in Appendix B.

The action space $\mathcal{A}$ is the set of possible actions that the agent can take at each time step. For the portfolio optimisation problem, the actions correspond to portfolio weights and are defined as follows:

$$\mathbf{a}_t = \mathbf{w}_t : w_{t,d} \in [0, 1] \quad \forall d \in \{1, \ldots, D\}, \tag{3.1}$$

where $\mathbf{w}_t$ is a vector of portfolio weights at time t, representing the allocation of the portfolio to each asset. The weights are constrained to be non-negative and sum to one:

$$\sum_{d=1}^{D} w_{t,d} = 1, \quad w_{t,d} \geq 0 \quad \forall d \in \{1, \ldots, D\}. \tag{3.2}$$

Moreover, they are initialised to be equal for all assets, meaning that the agent starts with an equal allocation to each asset in the portfolio. The reason behind this is to avoid an initial bias and allow the agent to learn an allocation from the environment rather than favour any particular asset.

The transition function T describes how the state of the environment changes in response to the action taken. It is defined as:

$$s_{t+1} = T(s_t, a_t), \tag{3.3}$$

where s_{t+1} is the new state of the environment after taking action a_t in state s_t. The transition function is determined by the dynamics of the financial market, which are influenced by the asset prices, trading volume and other factors.

The reward function R models the direct reward of taking an action a_t in state s_t and transitioning to a new state s_{t+1}. It is defined as the change in the portfolio value from time t to time $t+1$:

$$R_{t+1} = R(s_t, a_t, s_{t+1}) = V_{t+1} - V_t, \tag{3.4}$$

where the value of the portfolio at time t is given by the dot product of the portfolio weights and the asset close prices:

$$V_t = \mathbf{w}_t \cdot \mathbf{p}_t. \tag{3.5}$$

Regardless of the choice of reward function, the goal of the agent is to learn a policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximises the expected cumulative reward over time, which can be expressed as:

$$J(\pi) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R(s_t, a_t, s_{t+1})\right], \tag{3.6}$$

where T is the time horizon and γ is the discount factor that determines the importance of future rewards.

The environment is implemented in Python [1] using the `Gym` library [69], which provides a standard interface for reinforcement learning environments. The environment is defined as a class that inherits from the `gym.Env` class and implements the required methods:

[1] https://www.python.org/

`reset`, `step` and `render`.

3.3 Deep Reinforcement Learning Algorithms

The proposed solution is based on the DRL framework, which allows the agent to learn the optimal policy directly from the environment by taking actions and receiving rewards. The algorithms used are:

- Advantage Actor-Critic (A2C)
- Proximal Policy Optimisation (PPO)
- Deep Deterministic Policy Gradient (DDPG)
- Twin Delayed Deep Deterministic Policy Gradient (TD3)
- Soft Actor-Critic (SAC)

The implementation is done using the `Stable Baselines3` library [70], which provides a set of state-of-the-art DRL algorithms with a consistent interface and easy-to-use API. The pseudo-code for each algorithm is provided in Appendix A.1.

3.3.1 Hyper-parameter tuning

Hyper-parameter tuning is a crucial step in the training process of DRL models, as the right choice of hyper-parameters can significantly impact the model's performance. In this thesis, hyper-parameter tuning refers to the process of optimising the training parameters of the DRL algorithms to maximise their performance in the task of portfolio optimisation. The hyper-parameters tuned include the learning rate, batch size, number of training steps, and other algorithm-specific parameters, which are summarised in Appendix C.

To implement hyper-parameter tuning in Python, the `wandb` [71] library is used, which provides a simple and efficient way to track experiments, visualise results, and manage hyper-parameter sweeps. A sweep is defined as a search for hyper-parameters that optimises a cost function, in our case, the Sharpe ratio [72]. Given that the models were implemented using the `Stable Baselines3` library, the integration with `wandb` allows for seamless tracking of hyper-parameter configurations and their corresponding performance metrics [73].

As mentioned above, sweeps can optimise a cost function to avoid naively testing every possible combination. Using `wandb`, a Bayesian optimisation approach is taken [74], which uses a probabilistic model to estimate the performance of different hyper-parameter configurations and selects the next configuration to test based on the expected improvement over the current best configuration. This allows for a more efficient search of the hyper-parameter space and reduces the number of configurations that need to be tested. Another option to reduce the time taken to find the optimal hyper-parameters is to use early termination. This method will stop a poorly performing run before it has fully completed, saving computational resources.

3.4 Post-hoc Explainability

Given the goal of improving the explainability of the DRL models, this thesis adopts explainability techniques to interpret the model's decision-making process in a transparent manner. By using post-hoc methods, rather than modifying each model's architecture to enhance their transparency, the proposal is model-agnostic and can be applied to any DRL model. Consequently, it combines the ability to find the most suitable architecture while maintaining the interpretability.

The explainability techniques implemented are:

- Feature Importance
- Local Interpretable Model-agnostic Explanations (LIME)
- SHapley Additive exPlanations (SHAP)

Following the work from de-la-Rica-Escudero et al. (2025) [68], the implementation of these techniques follows two directions. First, as in their paper, a surrogate model maps the state space to the action space as a proxy for the model's decisions. The second direction is to use the LIME and SHAP techniques directly on the DRL model to interpret its decisions.

3.4.1 Surrogate Model Explainability

The surrogate model is trained to approximate the behaviour of the DRL model by learning the mapping from its inputs, the environment representation, to its outputs, the portfolio weights. In the paper [68], the authors do not explicitly acknowledge the use of a surrogate model, even though their code implementation does so. A potential reason behind not explaining the model's actions directly could be the code complexity in using SHAP with a DL model.

Given that the action space is continuous, the surrogate model is implemented using a `RandomForestRegressor` [75], which is a non-parametric model that can capture complex relationships between the inputs and outputs. The model is trained on the state-actions pairs of the test data, which is the object of the explanations. However, since the model has a number of hyper-parameters, it requires careful tuning to achieve optimal performance. Consequently, hyperparameter tuning was used find the optimal architecture using `HalvingGridSearchCV` [76], which is a method that iteratively narrows down the search space by evaluating a subset of hyper-parameters and discarding the less promising ones. The use of grid search rather than Bayesian Optimisation, as was done for DRL hyper-parameter tuning 3.3.1, is to replicate the approach taken in [68]. Once the optimal hyper-parameters have been found and the surrogate model has been trained, its prediction function is used as a proxy to interpret the original model's decisions.

Firstly, feature importance is built-in for Random Forest Regressors, and can be easily accessed with the built-in property `feature_importances_` [77]. This method provides the importance of each feature in the state representation by using a combination of the fraction of the samples a feature contributes to and the mean decrease in impurity.

Secondly, LIME and SHAP are applied to the surrogate model via the predict function to provide local explanations for individual predictions. Both of these techniques provide insights into the model's decisions by perturbing the input data and observing the changes in the output.

The LIME implementation is done using the `LimeTabularExplainer` [78], which is designed to work with tabular data and provides a way to explain individual predictions by approximating the model's behaviour locally with a linear model. Similarly, for SHAP, the framework provides a particular implementation for tree-based models, which is used to compute the SHAP values efficiently by exploiting the structure of the trees. Consequently, the `TreeExplainer` [79] is used to compute the SHAP values for the surrogate model.

3.4.2 Direct Model Explainability

Undoubtedly, a surrogate model adds an additional layer of complexity and may obscure the understanding of the original model's decisions. Therefore, the LIME and SHAP techniques are also applied directly to the prediction function of DRL model. This approach allows for a more direct interpretation of the model's decision-making process, without the need of a supplementary level.

For LIME, the implementation is again done using `LimeTabularExplainer`, but the prediction function is now obtained from the relevant DRL algorithm. For SHAP, the `KernelExplainer` is used, which is a model-agnostic method that randomly samples feature coalitions to approximate SHAP values to reduce computation [80].

Chapter 4

Results

This chapter presents the results of conducting experiments under the methodology proposed in Chapter 3. The experiments were designed to evaluate the performance of the implemented DRL models for portfolio optimisation in changing environment representations and market conditions. Moreover, to analyse the interpretability of the model's decisions, a framework using post-hoc explainability techniques is explored.

4.1 Dataset

Given the general difficulty in finding the appropriate DRL algorithm with a suitable reward function for portfolio optimisation, the five implemented algorithms were tested on five different datasets. Each dataset consists of a different set of financial assets, ranging from three different asset classes. First, three datasets were constructed using the stock constituents of three renowned indexes:

- Dow Jones Industrial Average (DJIA) with 30 stocks,
- Euro Stoxx 50 Index (Euro Stoxx 50) with 50 stocks,

- Financial Times Stock Exchange 100 Index (FTSE 100) with 100 stocks.

The constituents of each of the indexes were retrieved in April 2025 and can be found in Appendix D.1. It is important to note that the datasets were chosen to illustrate different currencies, as this introduces another factor of changing market conditions.

Additionally, two datasets were constructed using commodities and currencies, respectively. The commodities dataset includes six different commodities, which are listed in Appendix D.2. These are a sample of the most traded commodities in the market and were chosen by their availability in the `Yahoo! Finance API` [1]. With regard to the currencies dataset, it includes ten different currency pairs, listed in Appendix D.3. These were selected based on their trading volume and liquidity, with all pairs quoted in United States Dollar (USD).

The datasets are constructed using daily data from January 2016 to July 2025 downloaded using the Python `yfinance` library [81]. The dataset is partitioned into two disjoint sets: training and testing, with the training set containing data from January 2016 to December 2023, and the testing set starting on January 2024 until July 2025. The training set is used to train the DRL models, while the testing set is used to evaluate their performance. For hyper-parameter tuning, the training set is further split into a training and validation set, with the validation set corresponding to the period between January 2023 and December 2023. The validation set is used to evaluate the performance of the models for each hyper-parameter combination. The train-validation-test split is summarised in Table 4.1.

Dataset	**Training Period**	**Validation Period**	**Testing Period**
Dow Jones 30	Jan 2016 - Dec 2022	Jan 2023 - Dec 2023	Jan 2024 - Jul 2025
Euro Stoxx 50	Jan 2016 - Dec 2023	Not applicable	Jan 2024 - Jul 2025

[1]https://uk.finance.yahoo.com

Dataset	Training Period	Validation Period	Testing Period
FTSE 100	Jan 2016 - Dec 2023	Not applicable	Jan 2024 - Jul 2025
Commodities	Jan 2016 - Dec 2022	Jan 2023 - Dec 2023	Jan 2024 - Jul 2025
Currencies	Jan 2016 - Dec 2022	Jan 2023 - Dec 2023	Jan 2024 - Jul 2025

Table 4.1: Train-Validation-Test Split for each dataset

4.2 Experiment Design

To address the challenge of finding a suitable algorithm for portfolio optimisation, the five implemented DRL algorithms were tested on the five datasets described in Section 4.1, with the goal of evaluating the performance of each algorithm in different scenarios and market conditions. Moreover, the environment representation will also be varied to assess the impact of more information on the model's performance. Four environment representations were considered, each with a different number of features:

- Simple dataset: Open, High, Low, Close, Volume (OHLCV) prices of the assets.
- Covariance dataset: To the simple dataset, the covariance matrix of the assets is added to explicitly model the relationships between the assets.
- Indicators dataset: Technical and macroeconomic indicators are added to the simple dataset.
- Complete dataset: The complete dataset includes the simple dataset, the covariance matrix and the technical and macroeconomic indicators.

The strength of DRL algorithms lies in their ability to learn from high-dimensional data, which is why the goal is to evaluate whether a more exhaustive environment representation leads to better performance. However, with higher dimensionality comes a higher computational cost.

Finally, the performance of the algorithms is closely related to the choice of hyper-parameters. Ideally, the hyper-parameters should be tuned to find the optimal configuration for each algorithm and dataset combination. However, it was not feasible to perform tuning for all combinations of algorithms, datasets and environment representations. Consequently, the default hyper-parameters for all the experiments are outlined in Table 4.2. The hyper-parameters were chosen based on those from a testing run that was done on a small dataset of five tickers with indicators as environment representation. Those results can be seen in Appendix E.

Model	**Hyperparameter**	**Values / Range**
A2C	Number of steps	40
	Entropy coefficient	0.0003
	Learning rate	0.003
PPO	Number of steps	512
	Entropy coefficient	0.0005
	Learning rate	0.0015
	Batch size	64
DDPG	Batch size	256
	Buffer size	200000
	Learning rate	0.005
TD3	Batch size	128
	Buffer size	500000
	Learning rate	0.001
SAC	Batch size	64

Table 4.2: Default hyper-parameter configurations.

Model	Hyperparameter	Values / Range
	Buffer size	500000
	Learning rate	0.001
	Learning starts	2000
	Entropy coefficient	"auto_0.1"

Table 4.2: Default hyper-parameter configurations.

Overall, the experiments were designed to evaluate the performance of the implemented DRL algorithms in different scenarios, with the goal of finding the most suitable algorithm for portfolio optimisation. However, testing five algorithms on five distinct datasets with four possible environment representations would result in a total of twenty different experiments per algorithm. Additionally, optimising the parameters for each experiment further expands the experimental space and significantly increases the computational time required. Due to limited computational resources[2], the scope of experiments was adjusted as follows:

- Hyper-parameter tuning was performed only for the Dow Jones 30 dataset with simple and indicators environment representation, as it is the smallest equities dataset and requires less computational time.
- Since the covariance matrix, significantly increases the dimensionality of the environment representation, it was only included in the experiments with the Dow Jones 30, the currencies and the commodities datasets.

The final experimental design consists of 16 experiments, which are summarised in Table

[2]The university did not provide access to a computing cluster; therefore, all experiments were conducted on a personal computer.

4.3, where each row represents a unique combination of dataset, environment representation and whether hyper-parameter tuning is performed for this combination.

Dataset	**Environment Representation**	**Hyper-parameter Tuning**
Dow Jones 30	Simple	Yes
Dow Jones 30	Covariance	No
Dow Jones 30	Indicators	Yes
Dow Jones 30	Complete	No
Euro Stoxx 50	Simple	No
Euro Stoxx 50	Indicators	No
FTSE 100	Simple	No
FTSE 100	Indicators	No
Commodities	Simple	No
Commodities	Covariance	No
Commodities	Indicators	No
Commodities	Complete	No
Currencies	Simple	No
Currencies	Covariance	No
Currencies	Indicators	No
Currencies	Complete	No

Table 4.3: Summary of experiments conducted.

4.3 Evaluation

As outlined in the previous section, the experiments are designed to evaluate the performance of the implemented DRL algorithms in different scenarios and market conditions. The evaluation will focus on key performance metrics, as well as benchmarking against traditional portfolio optimisation techniques.

4.3.1 Performance Metrics

The performance metrics are provided through the `pyfolio` library [82], which includes a `perf_stats` method to calculate various performance metrics of a strategy.

The main metrics for comparison are:

- The cumulative return is the total change in investment price over a period of time, representing the overall percentage gain or loss from the initial investment value. The formula is given by:

$$\text{Cumulative return} = \frac{\text{Final portfolio value} - \text{Initial portfolio value}}{\text{Initial portfolio value}}. \tag{4.1}$$

- The annualised return is the geometric average of the amount of money earned by an investment each year over a given period of time, providing a standardised measure of annual performance. It is calculated as follows:

$$\text{Annualised Return} = \left(\frac{\text{Final portfolio value}}{\text{Initial portfolio value}}\right)^{\frac{1}{\text{Number of years}}} - 1. \tag{4.2}$$

- The annualised volatility is the standard deviation of returns annualised to provide a measure of investment risk on a yearly basis and can be computed with the following formula:

$$\text{Annualised Volatility} = \text{Standard Deviation of Returns} \times \sqrt{\text{Yearly trading days}}, \tag{4.3}$$

where the number of trading days per year is typically assumed to be 252.

- The Sharpe ratio is a measure of risk-adjusted performance that compares the excess return of an investment to a risk-free asset against its volatility. The ratio is given by:

$$\text{Sharpe Ratio} = \frac{R_p - R_f}{\sigma_p}, \tag{4.4}$$

where R_p is the annualised return of the portfolio, R_f is the annualised risk-free rate, and σ_p is the annualised volatility of the portfolio.

- The max drawdown is the maximum percentage loss from a peak to a trough during a specified period, indicating the worst-case scenario for portfolio decline. Its formula is:

$$\text{Max Drawdown} = \frac{\text{Peak Value} - \text{Trough Value}}{\text{Peak Value}}. \tag{4.5}$$

Other metrics available through the `pyfolio` library are outlined in Appendix F.

4.3.2 Benchmark Strategies

Aside from computing relevant performance metrics, the algorithms will be benchmarked against traditional portfolio optimisation methods. The benchmarks are designed to provide a baseline for comparison and to evaluate the performance of the DRL algorithms in relation to established methods. The following benchmark strategies were considered.

- Equal-weighted portfolio: A simple strategy that allocates an equal weight to each asset in the portfolio.

- Mean-variance optimisation: A classic portfolio optimisation method that aims to maximise Sharpe ratio.

- Min-variance portfolio: Another classic portfolio optimisation method that seeks to minimise the portfolio's volatility.

- Momentum portfolio: A strategy that invests in assets with positive momentum, i.e. those that have performed well in the previous time step, and avoids those with negative momentum.

The implementation of the mean-variance and the min-variance portfolio allocation strategies has been done using the `PyPortfolioOpt` Python library [83], whereas the equal-weighted and momentum strategies have been implemented using custom code.

Finally, if the portfolio is made up of equities of a relevant index, the benchmark will also include the index itself, which serves as a reference point for the performance of the portfolio.

4.4 Experiment: Algorithm Comparison

In this section, the results of the experiment to identify the suitability of the implemented DRL algorithms for portfolio optimisation under different market conditions are presented. The algorithms are trained on data with a simple environment representation, which only includes the OHLCV prices of the assets, and evaluated on the five datasets. The table 4.4 summarises the results of the experiment for the A2C algorithm, where each row corresponds to a different dataset and each column to a different performance metric. The results for the other algorithms are presented in Appendix G.1.

Dataset	Cumulative return	Annualised return	Annualised volatility	Sharpe ratio	Max drawdown
Dow Jones 30	0.2165	0.3387	0.1579	1.3203	-0.1649
Euro Stoxx 50	0.1467	0.2293	0.1549	0.9617	-0.1667
FTSE 100	0.1052	0.1623	0.1268	0.8520	-0.1409
Commodities	0.2353	0.3694	0.2041	1.1372	-0.1512
Currencies	-0.0011	-0.0017	0.0462	-0.0018	-0.0665

Table 4.4: Algorithm comparison results for the A2C implementation.

For the case of A2C, the algorithm demonstrates competitive performance particularly for the DowJones30 dataset, achieving a cumulative return of 21.65% and a Sharpe ratio of 1.32. Positive results are also obtained with the commodities dataset, which demonstrates the highest cumulative return of 23.53% and a Sharpe ratio of 1.14. Despite being of the same asset class, the EuroStoxx50 and the FTSE100 datasets show relatively lower performance, with cumulative returns of 14.67% and 10.52%, respectively. With regard to the currencies dataset, the performance of the algorithm is less impressive, with near-zero cumulative returns and Sharpe ratios, indicating that the algorithm struggles to learn a profitable strategy in this asset class.

Although similar observations can be made for the other algorithms, the performance varies significantly across different datasets. Regarding PPO, better performance is achieved in the commodities dataset, with a cumulative return of 25.31% and a Sharpe ratio of 1.26. which is slightly more than 0.1 higher than that of the A2C algorithm. DDPG performs the best in the DowJones30 dataset, achieving similar performance to that of A2C, with a 22.06% cumulative return and a Sharpe ratio of 1.38. However, TD3 surpasses all other algorithms for the DowJones30 dataset and the Commodities datasets, achieving 24.78% and 27.68% in cumulative return, respectively. Finally, SAC

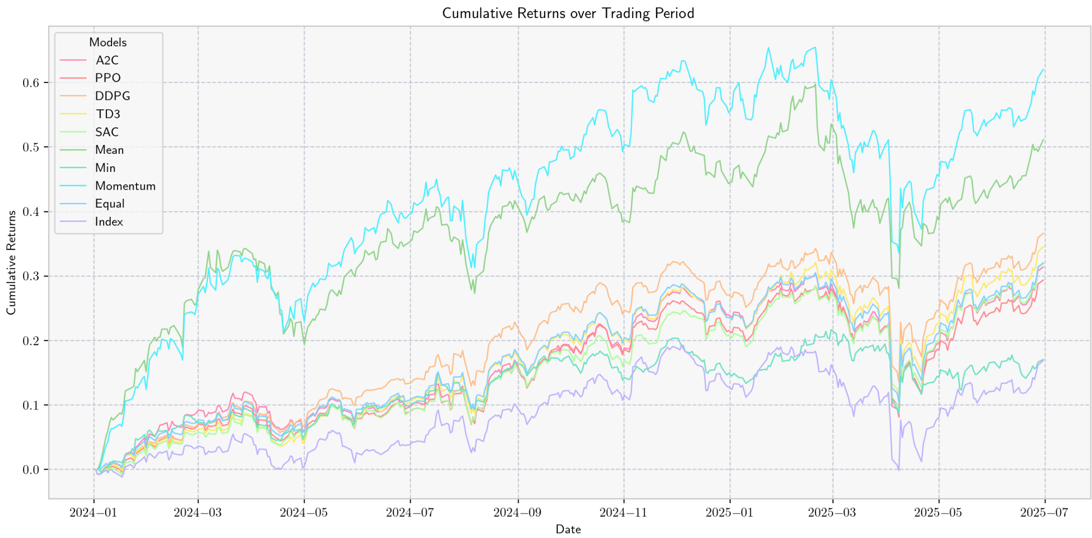

Figure 4.1: Evolution of the Cumulative Returns for the DowJones30 dataset with the OHLCV prices and indicators environment representation.

demonstrates a strong performance in the EuroStoxx50 dataset, with a cumulative return of 17.61% and a Sharpe ratio of 1.14, but it does not outperform the other algorithms in the DowJones30 and Commodities datasets. The algorithm that performs better in the FTSE100 dataset is DDPG, with a cumulative return of 13.36% and a Sharpe ratio of 1.08.

Taking the DowJones30 dataset with an environment representation made up of the OHLCV prices and the indicators, the performance of the algorithms can be benchmarked against traditional strategies and the DJIA Index. The evolution of the cumulative returns over the testing period is shown in Figure 4.1 and the corresponding performance metrics are summarised in Table 4.5.

Dataset	**Cumulative return**	**Annualised return**	**Annualised volatility**	**Sharpe ratio**	**Max drawdown**
A2C	0.2020	0.3139	0.1559	1.2575	-0.1716
PPO	0.1895	0.2937	0.1487	1.2407	-0.1556

Algorithm / Benchmark	Cumulative return	Annualised return	Annualised volatility	Sharpe ratio	Max drawdown
DDPG	0.2341	0.3663	0.1486	1.4896	-0.1546
TD3	0.2214	0.3456	0.1517	1.3938	-0.1609
SAC	0.2050	0.3189	0.1480	1.3340	-0.1538
Mean	0.3218	0.5114	0.1839	1.6096	-0.1983
Min	0.1123	0.1705	0.1157	0.9777	-0.1066
Momentum	0.3855	0.6204	0.1990	1.7388	-0.1929
Equal	0.2066	0.3205	0.1476	1.3461	-0.1541
Index	0.1110	0.1692	0.1532	0.7635	-0.1637

Table 4.5: Algorithm comparison results for the DowJones30 dataset. The colour correspond to the best performing configurations, with blue for the best performing DRL algorithm and green for the best benchmark.

The results show that all the algorithms outperform the performance of the index for all the considered metrics, as well as the min-variance portfolio. Out of all the DRL algorithms, DDPG achieves the highest cumulative return of 23.41% and a Sharpe ratio of 1.49, followed by TD3 with a cumulative return of 22.14% and a Sharpe ratio of 1.39, while PPO has the worst performance of the five with a cumulative return of 18.95% and a Sharpe ratio of 1.24. However, none of these outperform the mean-variance and momentum benchmark strategies, with the latter achieving a Sharpe ratio of 1.74 and a cumulative return of 38.55%, which is significantly higher than the performance of the DRL algorithms.

4.5 Experiment: Environment Representation

Another source of variability in the performance of the algorithms is the environment representation. In this section, the results of comparing the DRL algorithms on different environment representations are presented. For the experiment to be meaningful, it has been performed on the DowJones30 dataset, the currencies and the commodities datasets. This choice provides the ability to compare the performance of the algorithms across different asset classes, while also allowing for a more manageable computational cost. The table 4.6 compares the performance according to the Sharpe ratio and, in Appendix G.2, according to the cumulative return.

Algorithm	**Dataset**	**Simple**	**Indicators**	**Covariance**	**Complete**
A2C	DowJones30	1.3203	1.2575	1.3615	1.3929
	Commodities	1.1373	1.2881	1.0273	1.2079
	Currencies	-0.0018	-0.0571	-0.0674	-0.0053
PPO	DowJones30	1.3410	1.2407	1.3524	1.1890
	Commodities	1.2600	1.1256	1.1559	1.2015
	Currencies	0.0014	0.0699	0.0652	0.1399
DDPG	DowJones30	1.3826	1.4896	1.3562	1.3261
	Commodities	1.1745	0.9871	1.0731	1.1612
	Currencies	0.0528	0.0689	0.0200	0.0115
TD3	DowJones30	1.4911	1.3938	1.1792	1.2084
	Commodities	1.4537	0.9776	1.0462	1.2388
	Currencies	0.0807	-0.0088	-0.0086	-0.0436
SAC	DowJones30	0.9769	1.3340	1.4670	1.3017
	Commodities	1.3727	1.1973	1.1773	1.0500

Table 4.6: Environment representation experiment comparison according to the Sharpe ratio. The colour correspond to the best performing per row, with blue for the DowJones30 dataset and green for the Commodities dataset.

Algorithm	Dataset	Simple	Indicators	Covariance	Complete
	Currencies	0.0806	0.1771	-0.1441	0.2550

Table 4.6: Environment representation experiment comparison according to the Sharpe ratio. The colour correspond to the best performing per row, with blue for the DowJones30 dataset and green for the Commodities dataset.

The results show that the performance of the algorithms varies significantly across different environment representations. For the DowJones30 dataset, the OHLCV prices representation achieves the highest Sharpe ratio of 1.49 for TD3, closely followed by the OHLCV prices with indicators. The OHLCV prices with covariance representation achieves 1.47 for SAC. For the complete feature set, only the A2C algorithm achieves a higher Sharpe ratio of 1.39 than the other algorithms.

For the commodities dataset, the results are completely different to those of the DowJones30. The only coincidence is that TD3 achieves the highest Sharpe ratio and highest cumulative return in the simple feature set. Moreover, when looking at this dataset, explicitly adding the covariance to the environment representation does not lead to better performance in terms of Sharpe ratio and there is only one instance where the cumulative returns are highest, which would be for the DDPG algorithm with the complete feature set.

Finally, for the currencies dataset, the performance of the algorithms is significantly lower, going negative in some cases. A possible reason is the particular set of hyperparameters used. In this experiment, algorithm configuration was not altered or tuned for each specific feature set.

4.6 Experiment: Hyper-parameter Tuning

As has been mentioned in the above experiments, the performance of the algorithms is substantially influenced by the choice of hyper-parameters. Ideally, a systematic approach should be employed to find the optimal hyper-parameters for each algorithm, dataset and environment representation combination. However, due to the computational cost of hyper-parameter tuning, it was only performed for the DowJones30 dataset with the OHLCV and indicators environment representations over five trials. The results for each of the algorithms with the default hyper-parameters versus the tuned hyper-parameters are compared in Table 4.7 and Table 4.8 for the two features sets: simple and with indicators, respectively.

Algorithm	**Metric**	**Default parameters**	**Tuned parameters**
A2C	Cumulative Return	0.3387	0.3491
	Sharpe ratio	1.3203	1.4307
PPO	Cumulative Return	0.3174	0.2844
	Sharpe ratio	1.3410	1.1939
DDPG	Cumulative Return	0.3454	0.27404
	Sharpe ratio	1.3826	1.2088
TD3	Cumulative Return	0.3902	0.2974
	Sharpe ratio	1.4911	1.2354
SAC	Cumulative Return	0.2243	0.3703
	Sharpe ratio	0.9769	1.4521

Table 4.7: Hyper-parameter tuning experiment results for the DowJones30 dataset with simple environment representation.

Algorithm	Metric	Default parameters	Tuned parameters
A2C	Cumulative Return	0.3139	0.3573
	Sharpe ratio	1.2575	1.3994
PPO	Cumulative Return	0.2937	0.3134
	Sharpe ratio	1.2407	1.3679
DDPG	Cumulative Return	0.3663	0.4086
	Sharpe ratio	1.4896	1.5597
TD3	Cumulative Return	0.3456	0.2891
	Sharpe ratio	1.3938	1.2332
SAC	Cumulative Return	0.3189	0.2646
	Sharpe ratio	1.3340	1.1688

Table 4.8: Hyper-parameter tuning experiment results for the DowJones30 dataset with indicators environment representation.

By looking at the data presented in these tables, it is clear that finding the optimal configuration can have a significant impact on the performance of the algorithms. For instance, although the cumulative return for A2C in the simple feature set only improves by 1%, the Sharpe ratio increases from 1.32 to 1.43, meaning that the algorithm learns to better balance risk while maximising returns. Another example is the SAC algorithm, which achieves a better performance than that of the default configuration. However, for the PPO, DDPG and TD3 algorithms, there are no improvements and the default hyper-parameters perform better. Similarly, when looking at the indicators feature set, not all the algorithms show sign of improvements. These means that, for that particular algorithm, dataset and environment representation combination, the optimal hyper-parameter configuration has not been found. Understandably, due to the limited computational resources, the hyper-parameter search was very limited to only five runs, which is not sufficient for a thorough search.

4.7 Explainability Results

A main objective of this thesis is to be able to interpret the decisions made by the DRL algorithms. The following explainability framework is designed to provide insights into the decision-making process of the algorithms and present the most relevant features that influence the portfolio allocation decision at each time step in a visual and human-readable manner. As described in Section 3.4, two approaches were employed: a surrogate model and direct explanations.

For the purposes of visualisation, the results are shown for a sample dataset of five tickers (AAPL, CSCO, HON, MSFT, V) from the DJIA and using only the open, close, high, and low prices as the environment representation. This choice is made because explanations are more easily visualised when the number of assets and features is small. However, the explainability framework itself is general and can be applied to any number of assets and any environment representation. In practice, to support larger portfolios and more complex feature sets, an interactive dashboard could be developed, allowing users to select the relevant assets in the portfolio for which explanations are required. Moreover, only the explanations of the A2C algorithm are presented, as it serves as a representative example of the framework's capabilities.

The surrogate model was implemented following the proposal by de-la-Rica-Escudero et al. (2025) [68]. Although their paper does not explicitly acknowledge the use of a surrogate model nor outline the reasons for its use, it can be inferred that using a simpler transparent algorithm as a proxy provides built-in feature importance, which is

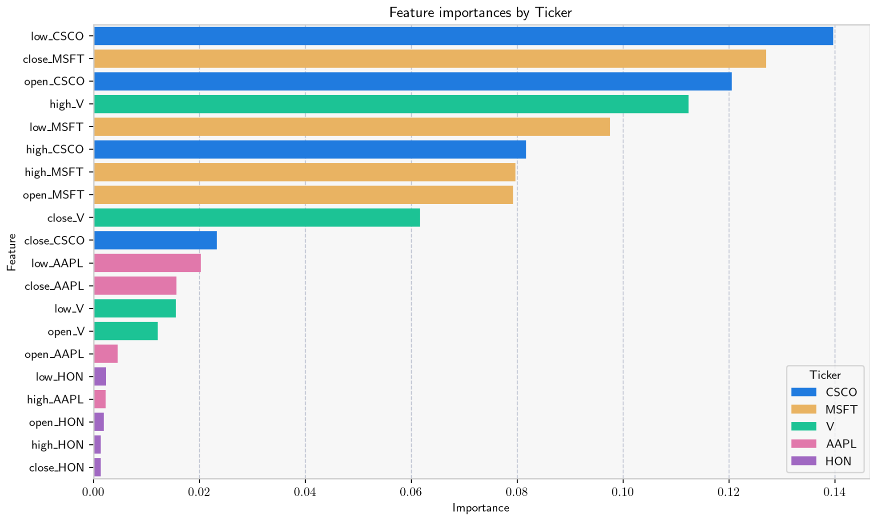

Figure 4.2: Top features from the surrogate model according to feature importance.

a global explainability method. However, there does not seem to be any clear value when using LIME and SHAP as they are both model-agnostic methods capable of providing explanations for any black-box model, given the prediction function of the algorithm.

4.7.1 Feature Importance Results

The feature importance results from the surrogate model are shown in Figure 4.2, where the top 20 features are ranked according to the importance measure. The ranking shows that the low price of CSCO and the close price of MSFT are the two most important features, followed by the open price of CSCO. At the bottom section of the ranking, HON and AAPL features have a lower importance.

This trend is further confirmed by looking at the mean importance of the features for each asset, as shown in Figure 4.3. This indicates that the agent heavily relied on the performance of MSFT and CSCO to guide its portfolio allocation decisions, while the other assets played a less significant role.

Another interesting result about the feature importance shown in Figure 4.2 is that the

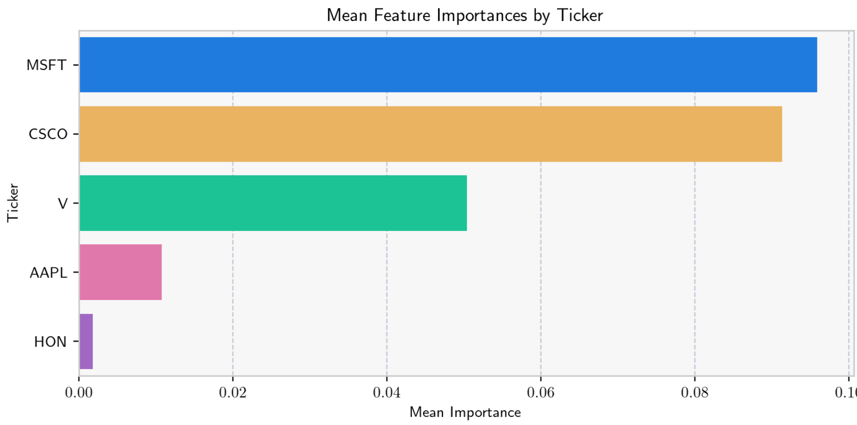

Figure 4.3: Mean feature importance per asset from the surrogate model.

different assets have a different OHLCV feature importance distribution, which suggests that the agent may have developed distinct strategies for each asset. Figure H.1 in the Appendix H.1 shows the top features grouped by asset, where it can be seen more clearly how each asset has a different most important feature. In the case of AAPL, HON and CSCO, the low price played a more critical role in informing the agent's decisions, showing how extreme price changes lead the agent to adjust the portfolio allocation. In contrast, for MSFT, the close price is the most important feature, which might imply the agent is more focused on the end of day activities of this asset. Finally, looking over all the assets, Figure H.2 in Appendix H.1 shows how features corresponding to low and high prices contributed to the agent's decisions. This again indicates that the agent is sensitive to price extremes of the assets.

4.7.2 Local Interpretable Model-agnostic Explanations Results

Moving on to the local explanations of LIME, the direct explanations for the A2C algorithm are provided. The benefit of LIME is its ability to provide explanations at a particular point in time. In this case, the local explanations are provided for the first time step of the test dataset, which corresponds to the 2nd of January 2024. In this

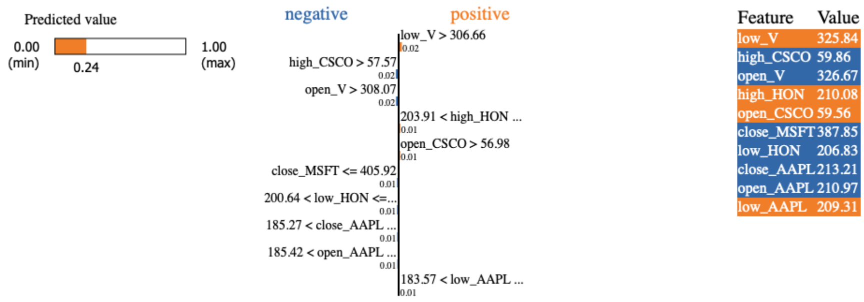

Figure 4.4: LIME explanations for the A2C algorithm at the first time step of the test dataset for the MSFT asset. The orange bars indicate features that contribute positively to the prediction, while the blue bars indicate features that contribute negatively.

section, Figure 4.4 illustrates the local explanations for the MSFT asset, whilst the other assets are found in Appendix H.2. This particular instance is affected by V's low price in the positive terms, meaning that it pushes the price higher, whilst the high price of CSCO and the open of V negatively affect the prediction by pushing it lower. Looking at these explanations over a number of days can help an investor decide whether there exist particular features that consistently contribute to the portfolio allocation decision.

4.7.3 Shapley Additive Explanations Results

Finally, the results of the SHAP analysis are presented. These provide a global view of the feature importance across all time steps and assets, as well as a local interpretation for individual predictions. As with the case of LIME, instead of using a surrogate model, the explanations are extracted from the A2C model directly. Since the predictions of the output are weight allocations across all portfolio assets, the SHAP values presented in this section correspond only to the AAPL asset, but the interpretations can be generalised to other assets and algorithms.

Looking at Figure 4.5, the x-axis represents the SHAP value, which measures the impact of a feature on the model's output; while the y-axis displays the top features. Each point

Figure 4.5: Beeswarm SHAP explanations for the A2C algorithm for the AAPL asset. The x-axis represents the SHAP values, whilst the y-axis represents the top features. The colour indicates the feature value, with magenta being high and blue being low.

in the beeswarm plot represents a single prediction, with the point's colour indicating whether its corresponding feature value is low, coloured in blue, or high, coloured in magenta. The beeswarm plot provides a visual representation of the distribution of SHAP values for each feature, allowing for an easy comparison of their importance. Surprisingly, the most important feature for the AAPL asset is the low price of the MSFT asset and, by visual inspection, the high values of the low price of MSFT push the AAPL allocation lower, while the low values of the low price of MSFT push the AAPL allocation higher. However, this is not quite significant as there is a cluster of data points around the zero, indicating that most frequently, the low price of MSFT does not have an impact on AAPL's allocation.

The `shap` Python library provides numerous visualisations to explain the prediction of a model. An interesting one is the force plot, shown in Figure 4.6, for the AAPL weight

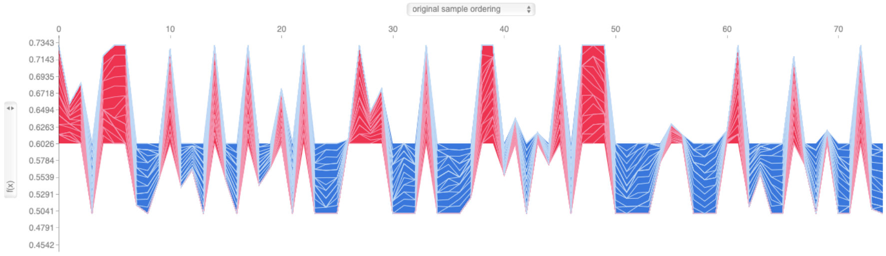

Figure 4.6: SHAP force plot for the weight allocation of the AAPL asset for the A2C algorithm.

allocation and the contribution of all features. The force plot visualises the impact of each feature on the model's weight allocation over the entire test dataset ordered by time. Positive values, visualised in magenta, show feature contributions that push the allocation higher, while negative values, in blue, push it lower. The baseline is the average weight allocation across all time steps.

The `shap` library provides the output in an interactive HyperText Markup Language (HTML) format that allows us to interact with the visualisation and gain deeper insights into the model's behaviour. From the beeswarm plot in Figure 4.5, the most important feature for the AAPL asset is the low price of MSFT. Using the force plot, it is possible to single out the effects of this particular feature over the test period, as shown in Figure 4.7, as well as its isolated effect in one specific prediction. Its impact is very well-defined as for approximately the first 30 samples, it has a positive contribution, increasing the AAPL allocation, while for the rest of the time steps, it has a negative contribution, decreasing the AAPL allocation.

Lundberg and Lee (2017) [52] showed that LIME is a subset of SHAP, it is possible to obtain local explanations using the `shap` library. Figure 4.8 shows the local explanation for the AAPL asset at the first time step of the test dataset, which corresponds to the 2nd of January 2024. Although the visualisation is different, the information it provides is similar to that of LIME. Shown horizontally, the features shown in magenta push the

Figure 4.7: SHAP force plot for the weight allocation of the AAPL asset for the A2C algorithm.

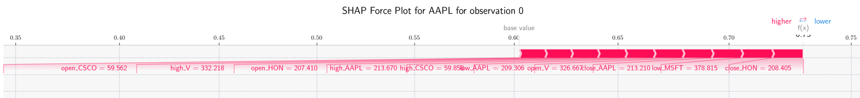

Figure 4.8: SHAP force plot for the weight allocation of the AAPL asset for the A2C algorithm at the first time step of the test dataset.

allocation higher, while those in blue push it lower.

Chapter 5

Legal, Social, Ethical and Professional Issues

The development and deployment of Deep Reinforcement Learning (DRL) algorithms to perform profitable portfolio allocation raises several legal, social, ethical and professional issues that must be considered. The incorporation of post-hoc explainable techniques to understand, interpret and clarify the decision-making process of these algorithms is a crucial step towards addressing these concerns. This chapter explores them in detail, referencing relevant regulatory frameworks and professional codes of conducts, and highlights potential mitigation strategies.

5.1 Legal Issues

When deploying Machine Learning (ML) algorithms in finance, the European Union (EU)'s Artificial Intelligence Act (AI Act) [65] classifies them as high-risk under Article 6 due to their potential impact on an individual's economic well-being. Consequently, high-risk systems must comply with Chapter 3 - Section 2, Articles 9-15, which include the obligations for risk management, data governance, documentation, transparency and human supervision. In particular, Article 13 requires that high-risk AI systems's outputs are interpretable. The use of explainability techniques ensures that the decision-making process of the algorithm can be understood and justified, thus adhering to the legal requirements set forth by the AI Act.

In addition, algorithmic trading systems must adhere to the European Union directive on Markets in Financial Instruments Directive II (MiFID II) [84] and the United Kingdom (UK) Financial Conduct Authority (FCA) guidance on *Algorithmic Trading Compliance in Wholesale Markets*. These regulations require robust governance and oversight frameworks, risk controls and thorough testing infrastructure. In this project, back-testing has been carried out to assess the algorithm's performance under varying market conditions followed by explainability techniques to enable the auditability of the system.

5.2 Social Issues

Despite the growing exposure to Artificial Intelligence (AI) systems with the rise of Large Language Models (LLMs), there is still a significant lack of understanding and trust in these systems. Some of the reasons behind this mistrust include:

- the black box nature of models, making outputs hard to interpret,
- lack of human-like qualities in the models, and
- perceived limitations to adapt to new situations and learn from previous mistakes.

In the context of this work, the integration of explainability techniques addresses the black box behaviour of the implemented technology by offering clear, data-driven justifications. However, transparency must also be accessible. It is essential to provide a user-friendly interface to easily access the explanations and generate plain language descriptions for non-technical audiences.

5.3 Ethical Issues

The ethical implications of DRL in portfolio optimisation are multifaceted. The primary concern is the potential for these systems to make decisions that may not align with human values or ethical standards. For instance, if the algorithm prioritises profit maximisation without considering the social or environmental impact of its investment choices, it could lead to unethical outcomes, such as supporting companies with poor labour practices or those contributing to environmental degradation. To address this concern, the following practices can be put in place:

1. the end-user should have full control over the assets included in their portfolio, allowing them to exclude companies that do not meet their ethical standards; or
2. the environment's representation can be expanded to include ethical dimensions, like social impact or environmental sustainability metrics.

Regarding the environment representation, it was not possible to incorporate such data as it is not readily available nor in a standardised format. This is why, although there was an interest to broaden the environment representation to include other sources outside of the financial domain, it was not feasible to do so given the available resources.

A critical point in portfolio allocation is risk management. In this research, a performance metric that balances return maximisation and risk minimisation is used for hyper-parameter tuning, ensuring that the agent learns to avoid overly risky investments. Additionally, the inclusion of a risk-free asset enables a safe-guard mechanism. The agent can be implemented to allocate all the resources to cash when market volatility exceeds a pre-defined threshold and resume only when said volatility decreases.

Finally, potential biases in the training data or model structure could result in allocations favouring certain industries or assets classes. This thesis mitigates such biases by constructing a portfolio from diversified indices (Appendix D.1) and using explainability techniques to audit the decisions for systematic biases.

5.4 Professional Issues

This work has been conducted in accordance with the British Computer Society (BCS) Code of Conduct [85] and the The Institution of Engineering and Technology (IET) Rules of Conduct [86]. All the work in this report is original unless stated otherwise, and all external contributions are explicitly acknowledged, following the principles of the BCS Code of Conduct [85]. Moreover, all third-party libraries, datasets and code have been explicitly acknowledged and their use complies with the respective licences and guidelines.

Most importantly, the project has been conducted within my own area of expertise: computational finance, Reinforcement Learning, and Explainable Artificial Intelligence. If any aspects of the project were beyond these, proper academic sources have been consulted to ensure the integrity and quality of the work.

Chapter 6

Conclusion

This thesis has explored the application of Deep Reinforcement Learning (DRL) algorithms for optimal portfolio allocation in dynamic financial markets. The primary objective was to develop an explainable model-agnostic framework capable of enhancing the understanding of any DRL algorithm's predictions. Such a tool would provide insights into the decision-making process of these complex models and facilitate auditability.

Five state-of-the-art DRL algorithms were implemented and evaluated on a portfolio management task. To assess the behaviour of these algorithms in different market conditions, a comprehensive experimental setup was designed, involving various datasets and environment representations. The datasets ranged from equities to commodities and currencies, each presenting unique challenges and opportunities for the DRL algorithms. The results demonstrated that DRL algorithms can effectively learn and adapt to dynamic market conditions, achieving competitive performance compared to traditional portfolio management strategies. However, the challenge remains in finding the optimal hyper-parameters, uniquely suited to each algorithm and dataset combination, in order to be able to fully exploit their potential.

Regarding the explainability aspect, the framework developed in this thesis successfully enhances the interpretability, transparency and auditability of these black box models. The incorporation of feature importance through a surrogate model, Local Interpretable Model-agnostic Explanations (LIME) analysis and SHapley Additive exPlanations (SHAP) values provides an exhaustive methodology for understanding both individual predictions and the overall decision-making process of the models over a test set. However, the results highlight the superiority of the SHAP technique, which is capable of providing global explanations and feature importance without the need for an additional layer, in conjunction with local explanations.

6.1 Future Work

Despite the comprehensive and exhaustive nature of this report, there are several areas through which future research could enhance this work. Indisputably, the main limitation of this thesis has been the lack of computation resources. Any future work should aim to fully explore the hyper-parameter space for each of the DRL algorithms, datasets and environment representations. In a similar vein, current research in price prediction has investigated the impact of a smaller feature representation by performing feature engineering techniques, such as feature selection and dimensionality reduction. Such measures might not only improve model performance and reduce over-fitting, but could also reduce computational requirements.

Moreover, even in the case of optimal hyper-parameter configuration, comparing the performance of the models to those of traditional methods left room for improvement. Exploring alternative reward functions, such as Sharpe ratio, incorporating additional constraints, such as transaction costs, or explicitly handling periods of high volatility could lead to more robust and effective trading strategies. Another possibility that led to reduced performance against the benchmarks was the exclusion of a risk-free asset in the portfolio. Future work could include a risk-free asset in the portfolio, which would result in a more diversified and potentially less risky portfolio. In terms of real-world scenarios, portfolios tend to be composed of different asset classes. Although this thesis explored the performance of the algorithms in different classes, forthcoming research could investigate the performance of these algorithms in multi-asset class portfolios.

Lastly, for the explainability aspect, the current framework would benefit from a more user-friendly interface that allows users to provide their dataset and prediction function easily and interactively explore the model's explanations. Another direction is to add intrinsic interpretability methods directly within the DRL algorithms, such as attention-based mechanisms [17], reducing the need for post-hoc explanation techniques.

References

[1] Z. Bodie, A. Kane, and A. J. Marcus, *Investments*. McGraw-Hill Education, 10 ed., 2014.

[2] J. Achiam, "Part 2: Kinds of rl algorithms," 2018.

[3] C. Molnar, *LIME*, ch. 14. Christoph Molnar, 3 ed., 2025.

[4] Wikipedia, "Dow jones industrial average - wikipedia," 7 2025.

[5] Wikipedia, "Euro stoxx 50 - wikipedia," 7 2025.

[6] Wikipedia, "Ftse 100 index - wikipedia," 8 2025.

[7] J. Shen and M. O. Shafiq, "Short-term stock market price trend prediction using a comprehensive deep learning system," *Journal of Big Data*, vol. 7, pp. 1–33, 12 2020.

[8] K. Lei, B. Zhang, Y. Li, M. Yang, and Y. Shen, "Time-driven feature-aware jointly deep reinforcement learning for financial signal representation and algorithmic trading," *Expert Systems with Applications*, vol. 140, p. 112872, 2 2020.

[9] C. Ma, J. Zhang, J. Liu, L. Ji, and F. Gao, "A parallel multi-module deep reinforcement learning algorithm for stock trading," *Neurocomputing*, vol. 449, pp. 290–302, 8 2021.

[10] M. Hasan, M. Z. Abedin, P. Hajek, K. Coussement, M. N. Sultan, and B. Lucey, "A blending ensemble learning model for crude oil price forecasting," *Annals of Operations Research*, pp. 1–31, 1 2024.

[11] I. K. Nti, A. F. Adekoya, and B. A. Weyori, "A comprehensive evaluation of ensemble learning for stock-market prediction," *Journal of Big Data*, vol. 7, pp. 1–40, 12 2020.

[12] J. M. T. Wu, Z. Li, N. Herencsar, B. Vo, and J. C. W. Lin, "A graph-based cnn-lstm stock price prediction algorithm with leading indicators," *Multimedia Systems*, vol. 29, pp. 1751–1770, 6 2023.

[13] J. Moody and M. Saffell, "Learning to trade via direct reinforcement," *IEEE Transactions on Neural Networks*, vol. 12, pp. 875–889, 7 2001.

[14] H. Yang, X. Y. Liu, S. Zhong, and A. Walid, "Deep reinforcement learning for automated stock trading: An ensemble strategy," *ICAIF 2020 - 1st ACM International Conference on AI in Finance*, 10 2020.

[15] X.-Y. Liu, Z. Xiong, S. Zhong, H. Yang, and A. Walid, "Practical deep reinforcement learning approach for stock trading," *NeurIPS 2018 AI in Finance Workshop.*, 11 2018.

[16] M. Guan and X. Y. Liu, "Explainable deep reinforcement learning for portfolio management: An empirical approach," *ICAIF 2021 - 2nd ACM International Conference on AI in Finance*, 11 2021.

[17] D. G. Cortés, E. Onieva, I. Pastor, L. Trinchera, and J. Wu, "Portfolio construction using explainable reinforcement learning," *Expert Systems*, vol. 41, p. e13667, 11 2024.

[18] A. B. Arrieta, N. Díaz-Rodríguez, J. D. Ser, A. Bennetot, S. Tabik, A. Barbado, S. Garcia, S. Gil-Lopez, D. Molina, R. Benjamins, R. Chatila, and F. Herrera, "Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai," *Information Fusion*, vol. 58, pp. 82–115, 10 2019.

[19] D. Gunning, M. Stefik, J. Choi, T. Miller, S. Stumpf, and G. Z. Yang, "Xai—explainable artificial intelligence," *Science Robotics*, vol. 4, 12 2019.

[20] D. Brigo, X. Huang, A. Pallavicini, and H. S. de Ocariz Borde, "Interpretability in deep learning for finance: a case study for the heston model," *SSRN Electronic Journal*, 4 2021.

[21] R. García-Céspedes, F. J. Alias-Carrascosa, and M. Moreno, "On machine learning models explainability in the banking sector: the case of shap," *Journal of the Operational Research Society*, 2025.

[22] Y. Sato, "Model-free reinforcement learning for financial portfolios: A brief survey," *arXiv preprint arXiv:1904.04973*, 4 2019.

[23] B. Bruce and J. Greene, *Chapter 4 - Portfolio Construction*, pp. 133–178. Academic Press, 2014.

[24] X. Li, Y. Li, Y. Zhan, and X.-Y. Liu, "Optimistic bull or pessimistic bear: Adaptive deep reinforcement learning for stock portfolio allocation," *arXiv preprint arXiv:1907.01503*, 6 2019.

[25] H. Markowitz, "Portfolio selection," *The Journal of Finance*, vol. 7, pp. 77–91, 3 1952.

[26] F. M. Clinic, "Mean variance portfolio theory."

[27] IBM, "What is machine learning (ml)?," 9 2021.

[28] V. François-Lavet, P. Henderson, R. Islam, M. G. Bellemare, and J. Pineau, "An introduction to deep reinforcement learning," *Foundations and Trends in Machine Learning*, vol. 11, pp. 219–354, 12 2018.

[29] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning.* MIT Press, 2016. `http://www.deeplearningbook.org`.

[30] C. Tang, B. Abbatematteo, J. Hu, R. Chandra, R. Martín-Martín, and P. Stone, "Deep reinforcement learning for robotics: A survey of real-world successes," *Annual Review of Control, Robotics, and Autonomous Systems*, vol. 8, pp. 153–188, 8 2024.

[31] K. Shao, Z. Tang, Y. Zhu, N. Li, and D. Zhao, "A survey of deep reinforcement learning in video games," *arXiv preprint arXiv:1912.10944*, 12 2019.

[32] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. V. D. Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, "Mastering the game of go with deep neural networks and tree search," *Nature 2016 529:7587*, vol. 529, pp. 484–489, 1 2016.

[33] R. Bellman, "A markovian decision process," *Journal of Mathematics and Mechanics*, vol. 6, pp. 679–684, 1957.

[34] R. Bellman, *Dynamic Programming.* Princeton University Press, 1 1957.

[35] S. Thrun, "Efficient exploration in reinforcement learning," tech. rep., Carnegie Mellon University, 1 1992.

[36] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 12 2013.

[37] M. G. Bellemare, W. Dabney, and R. Munos, "A distributional perspective on reinforcement learning," *34th International Conference on Machine Learning, ICML 2017*, vol. 1, pp. 693–711, 7 2017.

[38] V. Mnih, A. P. Badia, L. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," *33rd International Conference on Machine Learning, ICML 2016*, vol. 4, pp. 2850–2869, 2 2016.

[39] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. K. Openai, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 7 2017.

[40] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *4th International Conference on Learning Representations, ICLR 2016 - Conference Track Proceedings*, 9 2015.

[41] S. Fujimoto, H. V. Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *35th International Conference on Machine Learning, ICML 2018*, vol. 4, pp. 2587–2601, 2 2018.

[42] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," *35th International Conference on Machine Learning, ICML 2018*, vol. 5, pp. 2976–2989, 1 2018.

[43] D. H. G. B. Tokyo and J. Schmidhuber, "Recurrent world models facilitate policy evolution," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[44] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis,

"Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 12 2017.

[45] Y. Wu, E. Mansimov, S. Liao, R. Grosse, and J. Ba, "Scalable trust-region method for deep reinforcement learning using kronecker-factored approximation," *Advances in Neural Information Processing Systems*, vol. 2017-December, pp. 5280–5289, 8 2017.

[46] J. Schulman, S. Levine, P. Moritz, M. Jordan, and P. Abbeel, "Trust region policy optimization," *32nd International Conference on Machine Learning, ICML 2015*, vol. 3, pp. 1889–1897, 2 2015.

[47] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Physical Review*, vol. 36, p. 823, 9 1930.

[48] P. J. Phillips, C. A. Hahn, P. C. Fontana, A. N. Yates, K. Greene, D. A. Broniatowski, and M. A. Przybocki, "Four principles of explainable artificial intelligence," *National Institute of Standards and Technology*, vol. Internal Report 8312, 9 2021.

[49] P. Sequeira and M. Gervasio, "Interestingness elements for explainable reinforcement learning: Understanding agents' capabilities and limitations," *Artificial Intelligence*, vol. 288, p. 103367, 11 2020.

[50] S. Greydanus, A. Koul, J. Dodge, and A. Fern, "Visualizing and understanding atari agents," *Proceedings of the 35th International Conference on Machine Learning*, vol. 80, pp. 1792–1801, 7 2018.

[51] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1135–1144, 8 2016.

[52] S. M. Lundberg and S. I. Lee, "A unified approach to interpreting model predictions," *Advances in Neural Information Processing Systems*, vol. 2017-December, pp. 4766–4775, 5 2017.

[53] B. Amirshahi and S. Lahmiri, "Investigating the effectiveness of twitter sentiment in cryptocurrency close price prediction by using deep learning," *Expert Systems*, vol. 42, p. e13428, 8 2023.

[54] L. Breiman, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, 10 2001.

[55] S. Nembrini, I. R. König, and M. N. Wright, "The revival of the gini importance?," *Bioinformatics (Oxford, England)*, vol. 34, pp. 3711–3718, 11 2018.

[56] L. Shapley, *A value for n-person games*, pp. 307–317. Princeton University Press, 1953.

[57] E. Štrumbelj and I. Kononenko, "A general method for visualizing and explaining black-box regression models," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6594 LNCS, pp. 21–30, 2011.

[58] S. M. Lundberg, G. G. Erion, and S.-I. Lee, "Consistent individualized feature attribution for tree ensembles," *Advances in Neural Information Processing Systems*, 2019.

[59] C. Zhang, N. N. A. Sjarif, and R. Ibrahim, "Deep learning models for price forecasting of financial time series: A review of recent advancements: 2020–2022," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 14, p. e1519, 9 2023.

[60] A. Millea and F. Guijarro, "Deep reinforcement learning for trading—a critical survey," *Data 2021, Vol. 6, Page 119*, vol. 6, p. 119, 11 2021.

[61] R. Chaudhary, "Advanced stock market prediction using long short-term memory networks: A comprehensive deep learning framework," *Journal of Financial Data Science*, 5 2025.

[62] X.-Y. Liu, "Finrl: A deep reinforcement learning library for automated stock trading in quantitative finance," *SSRN Electronic Journal*, 11 2020.

[63] Z. Jiang and J. Liang, "Cryptocurrency portfolio management with deep reinforcement learning," *2017 Intelligent Systems Conference, IntelliSys 2017*, vol. 2018-January, pp. 905–913, 12 2016.

[64] Z. Zhang, S. Zohren, and S. Roberts, "Deep reinforcement learning for trading," *Papers*, 2019.

[65] E. Parliament and C. of the European Union, "Regulation (eu) 2024/1689 of the european parliament and of the council of 13 june 2024 laying down harmonised rules on artificial intelligence and amending certain union legislative acts (artificial intelligence act)," 2024. Official Journal of the European Union, L 2024/1689, 12.7.2024.

[66] Mandeep, A. Agarwal, A. Bhatia, A. Malhi, P. Kaler, and H. S. Pannu, "Machine learning based explainable financial forecasting," *2022 4th International Conference on Computer Communication and the Internet, ICCCI 2022*, pp. 34–38, 2022.

[67] C. A. Zhang, S. Cho, and M. Vasarhelyi, "Explainable artificial intelligence (xai) in auditing," *International Journal of Accounting Information Systems*, vol. 46, p. 100572, 9 2022.

[68] A. de La-Rica-Escudero, E. C. Garrido-Merchán, and M. Coronado-Vaca, "Explainable post hoc portfolio management financial policy of a deep reinforcement learning agent," *PLOS ONE*, vol. 20, p. e0315528, 1 2025.

[69] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Z. Openai, "Openai gym," *arXiv preprint arXiv:1606.01540*, 6 2016.

[70] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, pp. 1–8, 2021.

[71] L. Biewald, "Experiment tracking with weights and biases," 2020. Software available from wandb.com.

[72] W. Sharpe, "The sharpe ratio," *The Journal of Portfolio Management*, vol. Fall, 1994.

[73] W. . Biases, "Stable baselines 3 — weights & biases documentation," 8 2025.

[74] S. Falkner, A. Klein, and F. Hutter, "Bohb: Robust and efficient hyperparameter optimization at scale," *35th International Conference on Machine Learning, ICML 2018*, vol. 4, pp. 2323–2341, 7 2018.

[75] S. Learn, "Randomforestregressor — scikit-learn documentation."

[76] S. Learn, "Halvinggridsearchcv — scikit-learn 1.7.1 documentation."

[77] S. Learn, "1.11.2.5. feature importance evaluation — scikit-learn 1.7.1 documentation."

[78] Lime, "Lime tabular explainer — lime 0.1 documentation."

[79] SHAP, "shap.treeexplainer — shap documentation."

[80] SHAP, "shap.kernelexplainer — shap documentation."

[81] ranaroussi, "finance - python library."

[82] Q. Inc, "pyfolio."

[83] R. A. Martin, "Pyportfolioopt: portfolio optimization in python," *Journal of Open Source Software*, vol. 6, no. 61, p. 3066, 2021.

[84] European Parliament and Council of the European Union, "Directive 2014/65/eu of the european parliament and of the council of 15 may 2014 on markets in financial instruments and amending directive 2002/92/ec and directive 2011/61/eu (recast)," 2014. Official Journal of the European Union, L 173, 12.6.2014, p. 349-496.

[85] B. C. Society, "Bcs code of conduct," 2022.

[86] I. of Engineering and Technology, "Rules of conduct," 2024.

Appendix A

Algorithms

A.1 Deep Reinforcement Learning Algorithms

Algorithm 1 Advantage Actor-Critic (A2C) Pseudo-code

Initialise:
  Global shared policy parameters θ and value parameters θ_v
  Number of parallel workers N
  Global step counter $T \leftarrow 0$
  Hyper-parameters: discount γ, max steps per update $t_{\max}$, max total steps $T_{\max}$, learning rates α_π, α_v
repeat
    Reset gradients: $d\theta \leftarrow 0$, $d\theta_v \leftarrow 0$
    Initialise empty batch storage for all workers
    for worker $i = 1$ to N **do**
        $t_{\text{start}} \leftarrow t$
        Get initial state $s_t^{(i)}$ from worker i
        repeat
            Select action $a_t^{(i)} \sim \pi_\theta(\cdot|s_t^{(i)})$
            Execute $a_t^{(i)}$, observe reward $r_t^{(i)}$ and next state $s_{t+1}^{(i)}$
            Store $(s_t^{(i)}, a_t^{(i)}, r_t^{(i)})$ in worker i's trajectory
            $t \leftarrow t + 1$
        until terminal $s_t^{(i)}$ or $t - t_{\text{start}} == t_{\max}$
        $R^{(i)} = \begin{cases} 0 & \text{if terminal } s_t^{(i)} \\ V_{\theta_v}(s_t^{(i)}) & \text{otherwise} \end{cases}$
        for $j \in \{t-1, \dots, t_{\text{start}}\}$ **do**
            $R^{(i)} \leftarrow r_j^{(i)} + \gamma R^{(i)}$
            Accumulate gradients w.r.t. θ:
              $d\theta \leftarrow d\theta + \nabla_\theta \log \pi_\theta(a_j^{(i)}|s_j^{(i)})(R^{(i)} - V_{\theta_v}(s_j^{(i)}))$
            Accumulate gradients w.r.t. θ_v:
              $d\theta_v \leftarrow d\theta_v + \nabla_{\theta_v}(R^{(i)} - V_{\theta_v}(s_j^{(i)}))^2$
        end for
    end for
    // Synchronous update: wait for all workers to complete
    Average gradients: $d\theta \leftarrow \frac{1}{N} d\theta$, $d\theta_v \leftarrow \frac{1}{N} d\theta_v$
    Update $\theta \leftarrow \theta + \alpha_\pi d\theta$, $\theta_v \leftarrow \theta_v - \alpha_v d\theta_v$
    $T \leftarrow T + N \times t_{\max}$
until $T > T_{\max}$

Algorithm 2 Proximal Policy Optimisation (PPO) Pseudo-code

Initialise:

Policy parameters θ_0 and value function parameters ϕ_0

Global step counter $T \leftarrow 0$

Hyper-parameters: discount γ, GAE parameter λ, clipping parameter ϵ, learning rates α_π, α_v, epochs per update K_{epochs}, minibatch size $N_{\text{minibatch}}$, loss coefficients c_1, c_2

for $k = 0, 1, 2, \dots$ **do**

  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy π_{θ_k} in environment

  Store transitions: $\{(s_t, a_t, r_t, s_{t+1}, \text{done}_t)\}$

  for each trajectory τ in $\mathcal{D}_k$ **do**

    Compute value estimates: $V_t = V_{\phi_k}(s_t)$

    Compute TD residuals: $\delta_t = r_t + \gamma V_{t+1}(1 - \text{done}_t) - V_t$

    Compute GAE advantages: $\hat{A}_t = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$

    Compute returns: $\hat{R}_t = \hat{A}_t + V_t$

  end for

  for epoch $e = 1$ to K_{epochs} **do**

    Shuffle dataset $\mathcal{D}_k$

    for each minibatch $\mathcal{B}$ in $\mathcal{D}_k$ **do**

      $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$

      $L^{\text{CLIP}}(\theta) = \frac{1}{|\mathcal{B}|} \sum_{t \in \mathcal{B}} \min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)$

      $L^{VF}(\phi) = \frac{1}{|\mathcal{B}|} \sum_{t \in \mathcal{B}} \left(V_\phi(s_t) - \hat{R}_t\right)^2$

      $L(\theta, \phi) = L^{\text{CLIP}}(\theta) - c_1 L^{VF}(\phi) + c_2 S[\pi_\theta]$

      $\theta \leftarrow \theta + \alpha_\pi \nabla_\theta L^{\text{CLIP}}(\theta)$

      $\phi \leftarrow \phi - \alpha_v \nabla_\phi L^{VF}(\phi)$

    end for

  end for

  Update policy: $\theta_{k+1} = \theta$

  Update value function: $\phi_{k+1} = \phi$

  $T \leftarrow T + |\mathcal{D}_k|$

end for

Algorithm 3 Deep Deterministic Policy Gradient (DDPG) Pseudo-code

Initialise:

Critic network $Q_{\theta^Q}(s,a)$ and actor $\mu_{\theta^\mu}(s)$ with random weights θ^Q, θ^μ

Target networks: $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$

Replay buffer $\mathcal{B}$

Hyper-parameters: discount γ, soft update rate τ, batch size N, exploration noise process $\mathcal{N}$, learning rates α_Q, α_μ, total episodes M, steps per episode T

for episode = 1 to M **do**
  Initialise random process $\mathcal{N}$ for action exploration
  Receive initial state s_1
  for $t = 1$ to T **do**
    Select action $a_t = \mu_{\theta^\mu}(s_t) + \mathcal{N}_t$
    Execute a_t, observe reward r_t and next state s_{t+1}
    Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{B}$
    Sample mini-batch $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^N$ from $\mathcal{B}$
    Compute target: $y_i = r_i + \gamma Q_{\theta^{Q'}}(s_{i+1}, \mu_{\theta^{\mu'}}(s_{i+1}))$
    Update critic by minimising: $L(\theta^Q) = \frac{1}{N}\sum_i (y_i - Q_{\theta^Q}(s_i, a_i))^2$
    Update actor by policy gradient:
      $\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q_{\theta^Q}(s_i, a)|_{a=\mu_{\theta^\mu}(s_i)} \nabla_{\theta^\mu}\mu_{\theta^\mu}(s_i)$
    Update target networks:
      $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
      $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for

Algorithm 4 Twin Delayed Deep Deterministic Policy Gradient (TD3) Pseudo-code

Initialise:

Critic networks $Q_{\theta_1}(s,a)$ and $Q_{\theta_2}(s,a)$ with random weights θ_1, θ_2

Actor policy $\mu_\phi(s)$ with random weights ϕ

Target networks: $\theta'_1 \leftarrow \theta_1$, $\theta'_2 \leftarrow \theta_2$, $\phi' \leftarrow \phi$

Replay buffer $\mathcal{B}$

Hyper-parameters: discount γ, target policy noise $\tilde{\sigma}$, noise clip c, policy delay d, exploration noise σ, update rate τ, batch size N, total steps T

for $t = 1$ to T **do**

  Observe state s_t

  Sample action with exploration: $a_t = \mu_\phi(s_t) + \epsilon$, $\epsilon \sim \mathcal{N}(0, \sigma)$

  Execute a_t, observe reward r_t and next state s_{t+1}

  Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{B}$

  Sample mini-batch $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^N$ from $\mathcal{B}$

    Sample clipped noise: $\tilde{\epsilon} \sim \text{clip}(\mathcal{N}(0, \tilde{\sigma}), -c, c)$

    Compute target action: $\tilde{a}_{i+1} = \mu_{\phi'}(s_{i+1}) + \tilde{\epsilon}$

    Compute target Q-value: $y_i = r_i + \gamma \min_{j=1,2} Q_{\theta'_j}(s_{i+1}, \tilde{a}_{i+1})$

    Update each critic by minimising $L(\theta_j) = \frac{1}{N}\sum_i \big(y_i - Q_{\theta_j}(s_i, a_i)\big)^2$

  if $t \bmod d = 0$ **then**

    Update actor by policy gradient: $\nabla_\phi J \approx \frac{1}{N}\sum_i \nabla_a Q_{\theta_1}(s_i, a) \mid_{a=\mu_\phi(s_i)} \nabla_\phi \mu_\phi(s_i)$

    Update target networks:

      $\theta'_j \leftarrow \tau\,\theta_j + (1-\tau)\,\theta'_j$ for $j = 1, 2$

      $\phi' \leftarrow \tau\,\phi + (1-\tau)\,\phi'$

  end if

end for

Algorithm 5 Soft Actor-Critic (SAC) Pseudo-code

Initialise:
  Actor network $\pi_\theta(a|s)$ with parameters θ
  Two critic networks $Q_{\phi_1}(s,a)$ and $Q_{\phi_2}(s,a)$ with parameters ϕ_1, ϕ_2
  Target critic networks: $\phi'_1 \leftarrow \phi_1$, $\phi'_2 \leftarrow \phi_2$
  Replay buffer $\mathcal{D}$
  Hyper-parameters: discount γ, temperature α (fixed or learnable), target entropy $\mathcal{H}$ (if α is learnable), batch size N, learning rates $\lambda_Q, \lambda_\pi, \lambda_\alpha$, soft update rate τ
for each training step **do**
  Observe state s_t
  Sample action: $a_t \sim \pi_\theta(\cdot|s_t)$
  Execute a_t, observe reward r_t and next state s_{t+1}
  Store (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$
  if time to update **then**
    Sample mini-batch $\{(s_i, a_i, r_i, s_{i+1})\}_{i=1}^N$ from $\mathcal{D}$
    Update Critics:
    for $j = 1, 2$ **do**
      Sample next actions: $\tilde{a}_{i+1} \sim \pi_\theta(\cdot|s_{i+1})$
      Compute target Q-values:
        $y_i = r_i + \gamma \left(\min_{k=1,2} Q_{\phi'_k}(s_{i+1}, \tilde{a}_{i+1}) - \alpha \log \pi_\theta(\tilde{a}_{i+1}|s_{i+1}) \right)$
      Update critic: $\phi_j \leftarrow \phi_j - \lambda_Q \nabla_{\phi_j} \frac{1}{N} \sum_i (Q_{\phi_j}(s_i, a_i) - y_i)^2$
    end for
    Update Actor:
    Sample actions with reparametrisation: $\tilde{a}_i = f_\theta(\epsilon_i; s_i)$ where $\epsilon_i \sim \mathcal{N}(0, I)$
    Compute policy loss:
      $J(\theta) = \frac{1}{N} \sum_i \left[\alpha \log \pi_\theta(\tilde{a}_i|s_i) - \min_{j=1,2} Q_{\phi_j}(s_i, \tilde{a}_i) \right]$
    Update actor: $\theta \leftarrow \theta - \lambda_\pi \nabla_\theta J(\theta)$
    if α is learnable **then**
      Update Temperature:
      $J(\alpha) = \frac{1}{N} \sum_i \alpha \left(\log \pi_\theta(\tilde{a}_i|s_i) + \mathcal{H} \right)$
      Update temperature: $\alpha \leftarrow \alpha - \lambda_\alpha \nabla_\alpha J(\alpha)$
    end if
    Update Target Networks:
    $\phi'_j \leftarrow \tau \phi_j + (1 - \tau) \phi'_j$ for $j = 1, 2$
  end if
end for

A.2 Explainability Algorithms

Algorithm 6 Shapley Value Approximation

Require: Number of iterations M, instance of interest $\mathbf{x}$, feature index j, data matrix $\mathbf{X}$, model $\hat{f}$

Ensure: Estimated Shapley value $\phi_j(\mathbf{x})$

for $m = 1$ to M **do**

Draw a random instance $\mathbf{z}$ from data matrix $\mathbf{X}$

Choose a random permutation $\mathbf{o}$ of the feature indices

Order $\mathbf{x}$ according to $\mathbf{o}$: $\mathbf{x_o} = \big(x_{(1)}, \ldots, x_{(j)}, \ldots, x_{(p)}\big)$

Order $\mathbf{z}$ according to $\mathbf{o}$: $\mathbf{z_o} = \big(z_{(1)}, \ldots, z_{(j)}, \ldots, z_{(p)}\big)$

Construct two new instances:

$\mathbf{x}_{+j} = \big(x_{(1)}, \ldots, x_{(j-1)}, x_{(j)}, z_{(j+1)}, \ldots, z_{(p)}\big)$

$\mathbf{x}_{-j} = \big(x_{(1)}, \ldots, x_{(j-1)}, z_{(j)}, z_{(j+1)}, \ldots, z_{(p)}\big)$

Compute marginal contribution:

$$\phi_j^{(m)} = \hat{f}(\mathbf{x}_{+j}) - \hat{f}(\mathbf{x}_{-j})$$

end for

Compute average Shapley value:

$$\phi_j(\mathbf{x}) = \frac{1}{M} \sum_{m=1}^{M} \phi_j^{(m)}$$

Appendix B

State Representation

B.1 Technical Indicators

The following technical indicators are used to represent the state of the financial environment. They are calculated based on the historical price data of the assets in the portfolio.

- Simple Moving Average (SMA): Lagging indicator that smooths the price over a period of time. It is the unweighted mean of the previous k data points and is computed following:
$$\mathrm{SMA}_t^k = \frac{1}{k} \sum_{i=t-k+1}^{t} p_i, \tag{B.1}$$
where t is the current time step, k is the look-back period and p_i is the close price at time i. In this thesis, the SMA is calculated for 5, 10 and 20 days.

- Exponential Moving Average (EMA): Weighted moving average that gives more importance to recent prices with the goal of making it more responsive to new information. It is calculated using the formula:

$$\mathrm{EMA}_t^k = \begin{cases} p_t & \text{if } t = 0 \\ \frac{2}{k+1} p_t + \left(1 - \frac{2}{k+1}\right) \mathrm{EMA}_{t-1}^k & \text{if } t > 0 \end{cases}, \tag{B.2}$$

where k is the look-back period, p_t is the close price at time t and $\frac{2}{k+1}$ is the smoothing factor. In this thesis, the EMA is calculated for 5 and 10 days.

- Moving Average Convergence Divergence (MACD): Momentum indicator that shows the relationship between two moving average of an asset's price. The formula is:

$$\mathrm{MACD}_t = \mathrm{EMA}_t^{k_1} - \mathrm{EMA}_t^{k_2}, \tag{B.3}$$

where k_1 and k_2 are the look-back periods for the short-term (12 periods) and long-term (26 periods) EMAs, respectively, and p_t is the close price at time t.

- Relative Strength Index (RSI): Momentum indicator that measures the magnitude of recent changes to identify overbought or oversold conditions in the market. It is computed using smoothed moving averages for the upward change in closing price p_t, defined as $u_t = \max\{p_t - p_{t-1}, 0\}$:

$$\mathrm{SMMA}_t^k(u_t) = \frac{1}{k} u_t + \left(1 - \frac{1}{k}\right) \mathrm{SMMA}_{t-1}^k(u_{t-1}) \tag{B.4}$$

and for the downward change, defined as $d_t = \max\{p_{t-1} - p_t, 0\}$:

$$\mathrm{SMMA}_t^k(d_t) = \frac{1}{k} d_t + \left(1 - \frac{1}{k}\right) \mathrm{SMMA}_{t-1}^k(d_{t-1}). \tag{B.5}$$

Then, the Relative Strength Index (RSI) is given by:

$$\mathrm{RSI}_t = 100 - \frac{100}{1 + \mathrm{RS}_t}, \tag{B.6}$$

where the relative strength RS_t is defined as:

$$\mathrm{RS}_t = \frac{\mathrm{SMMA}_t^k(u_t)}{\mathrm{SMMA}_t^k(d_t)}. \tag{B.7}$$

- Commodity Channel Index (CCI): Momentum indicator that measures the deviation of the price from its historical average price over a period of time. It is calculated as:

$$\mathrm{CCI}_t^k = \frac{\mathrm{TP}_t^k - \mathrm{MA}_t^k}{0.015 \cdot \mathrm{MD}_t}, \tag{B.8}$$

  where k is the number of periods (14 days), the typical price TP_t is:

$$\mathrm{TP}_t = \sum_{i=t-k+1}^{t} \frac{p_i + h_i + l_i}{3}, \tag{B.9}$$

  with p_t, h_t and l_t being the close, high and low prices at time step t, respectively, and the moving average MA_t^k is:

$$\mathrm{MA}_t^k = \frac{1}{k} \sum_{i=t-k+1}^{t} \mathrm{TP}_i \tag{B.10}$$

  and the mean deviation MD_t is:

$$\mathrm{MD}_t = \frac{1}{k} \sum_{i=t-k+1}^{t} \left| \mathrm{TP}_i - \mathrm{MA}_i^k \right|. \tag{B.11}$$

- Bollinger Bands: Volatility indicator that defines the trend-line for high and low prices based on the deviation of the asset from the moving average. The upper bands is calculated as:

$$\mathrm{BOLLUB}_t^k = \mathrm{MA}_t^k + m \cdot \mathrm{SD}_t^k, \tag{B.12}$$

  where k is the number of periods (20 days), m is the number of standard deviations away from the moving average (2 standard deviations) and SD_t^k is the standard deviation of the typical price over the same period. Similarly, the lower band is calculated as:

$$\mathrm{BOLLDB}_t^k = \mathrm{MA}_t^k - m \cdot \mathrm{SD}_t^k. \tag{B.13}$$

- Average True Range (ATR): Volatility indicator that measures the average range of price movement over a period of time, usually 14 days. It is calculated as:

$$\mathrm{ATR}_t^k = \frac{1}{k} \sum_{i=t-k+1}^{t} \mathrm{TR}_i, \tag{B.14}$$

where the true range TR_i is defined as:

$$\mathrm{TR}_i = \max \{h_i - l_i, |h_i - p_{i-1}|, |l_i - p_{i-1}|\}, \tag{B.15}$$

where h_i and l_i are the high and low prices of the asset at time i, respectively, and p_{i-1} is the close price of the asset at time $i-1$. The true range finds the maximum of the following three:

  - most recent period high minus most recent period low,
  - absolute value of the most recent period high minus the previous close, and
  - absolute value of the most recent period low minus the previous close.

- Average Directional Index (ADX): Trend indicator used to measure the strength of a trend by quantifying the price movement. It is calculated as:

$$\mathrm{ADX}_t^k = \frac{1}{k} \sum_{i=t-k+1}^{t} \mathrm{DX}_i, \tag{B.16}$$

where the Directional Movement Index (DX) is defined as:

$$\mathrm{DX}_t = \frac{100 \cdot |\mathrm{PDI}_t - \mathrm{MDI}_t|}{\mathrm{PDI}_t + \mathrm{MDI}_t} \tag{B.17}$$

with the Positive Directional Index (PDI) and Negative Directional Index (MDI) calculated as

$$\mathrm{PDI}_t = \frac{100 \cdot \mathrm{SMMA}_t^k(\mathrm{DM}^+)}{\mathrm{ATR}_t^k} \tag{B.18}$$

and

$$\mathrm{MDI}_t = \frac{100 \cdot \mathrm{SMMA}_t^k(\mathrm{DM}^-)}{\mathrm{ATR}_t^k}, \tag{B.19}$$

where DM^+ and DM^- are the positive and negative directional movements, respectively, calculated as:

$$\mathrm{DM}^+ = \max\left(0, h_t - h_{t-1}\right) \tag{B.20}$$

and

$$\mathrm{DM}^- = \max\left(0, l_{t-1} - l_t\right), \tag{B.21}$$

where h_t and l_t are the high and low prices of the asset at time t, respectively.

- Rate of Change (ROC): Momentum indicator that measures the percentage change in price between the current price p_t and the price p_{t-k} periods ago. It is given by the formula:

$$\mathrm{ROC}_t^k = \frac{p_t - p_{t-k}}{p_{t-k}} \cdot 100. \tag{B.22}$$

In this thesis, the ROC is calculated for $k = 10$ days.

B.2 Macroeconomic Indicators

The following macroeconomic indicators are used to represent the state of the financial environment. They provide additional context about the market conditions and are calculated based on external data sources.

- Volatility Index (VIX) measures the market's expectation of future volatility based on options prices. This is only available for US markets.
  - VIX is calculated using the implied volatility of Standard and Poor's 500 Index (S&P 500) options.
  - VXD is calculated using the implied volatility of Dow Jones 30 Index (DW30) options.
- Currency index captures the impact from the monetary market on the stock market.
  - U.S. Dollar Index (DXY): United States (U.S.) dollar's value relative to a basket of foreign currencies.
  - Euro Index (EXY): Euro's value relative to a basket of foreign currencies.
  - British Pound Currency Index (BXY): British pound's value relative to a basket of foreign currencies.
- Interest rates reflect the cost of borrowing money and the return on savings. This is only available for US markets.
  - 3-Month Treasury Yield (IRX): Reflects the return on investment for a 3-month government bond.
  - 5-Year Treasury Yield (FVX): Reflects the return on investment for a 5-year government bond.

- 10-Year Treasury Yield (TNX): Reflects the return on investment for a 10-year government bond.

Appendix C

Hyper-parameter tuning

For the five implemented Deep Reinforcement Learning algorithms, the following table summarises the hyper-parameters that were tuned during the training process.

Model	Hyperparameter	Values / Range
A2C	Number of steps	{5, 10, 20, 30, 40}
	Entropy coefficient	Uniform[1×10^{-8}, 1×10^{-3}]
	Learning rate	Uniform[1×10^{-5}, 1×10^{-2}]
PPO	Number of steps	{128, 256, 512, 1024, 2048}
	Entropy coefficient	Uniform[1×10^{-8}, 1×10^{-3}]
	Learning rate	Uniform[1×10^{-5}, 1×10^{-2}]
	Batch size	{32, 64, 128, 256, 512}
DDPG	Batch size	{64, 128, 256}
	Buffer size	{50000, 100000, 200000, 500000}
	Learning rate	Uniform[1×10^{-5}, 1×10^{-2}]
TD3	Batch size	{64, 100, 128, 256}
	Buffer size	{500000, 1000000, 2000000}

Model	**Hyperparameter**	**Values / Range**
	Learning rate	Uniform[$1{\times}10^{-5}$, $1{\times}10^{-2}$]
SAC	Batch size	{32, 64, 128}
	Buffer size	{100000, 500000, 1000000, 2000000}
	Learning rate	Uniform[$1{\times}10^{-5}$, $1{\times}10^{-2}$]
	Learning starts	{500, 1000, 2000, 5000}
	Entropy coefficient	{"auto", "auto_0.1", "auto_0.01"}

Table C.1: Hyper-parameter tuning configurations for different RL algorithms.

Appendix D

Datasets

D.1 Equities

D.1.1 Dow Jones 30

The Dow Jones Industrial Average (DJIA) is a stock market index with 30 companies listed on United States (U.S.) stock exchanges. The trading symbol is ^DJI and as of April 2025, its constituents are:

Symbol	**Name**	**Sector**	**Industry**
AXP	American Express	Financial Services	Credit Services
AMGN	Amgen	Healthcare	Drug Manufacturers - General
AAPL	Apple	Technology	Consumer Electronics
AMZN	Amazon	Consumer Cyclical	Internet Retail
BA	Boeing	Industrials	Aerospace & Defense

Symbol	Name	Sector	Industry
CAT	Caterpillar	Industrials	Farm & Heavy Construction Machinery
CRM	Salesforce	Technology	Software - Application
CSCO	Cisco	Technology	Communication Equipment
CVX	Chevron	Energy	Oil & Gas Integrated
DIS	Walt Disney	Communication Services	Entertainment
GS	Goldman Sachs	Financial Services	Capital Markets
HD	Home Depot	Consumer Cyclical	Home Improvement Retail
HON	Honeywell	Industrials	Conglomerates
IBM	IBM	Technology	Information Technology Services
JNJ	Johnson & Johnson	Healthcare	Drug Manufacturers - General
JPM	JP Morgan Chase	Financial Services	Banks - Diversified
KO	Coca-Cola	Consumer Defensive	Beverages - Non-Alcoholic
MCD	McDonald's	Consumer Cyclical	Restaurants
MMM	3M	Industrials	Conglomerates
MRK	Merck	Healthcare	Drug Manufacturers - General
MSFT	Microsoft	Technology	Software - Infrastructure

Symbol	Name	Sector	Industry
NKE	Nike	Consumer Cyclical	Footwear & Accessories
NVDA	NVIDIA	Technology	Semiconductors
PG	Procter & Gamble	Consumer Defensive	Household & Personal Products
SHW	Sherwin-Williams	Basic Materials	Specialty Chemicals
TRV	Travelers	Financial Services	Insurance - Property & Casualty
UNH	UnitedHealth	Healthcare	Healthcare Plans
V	Visa	Financial Services	Credit Services
VZ	Verizon	Communication Services	Telecom Services
WMT	Walmart	Consumer Defensive	Discount Stores

Table D.1: Constituents of the Dow Jones Industrial Average index as of April 2025. [4]

D.1.2 Euro Stoxx 50

The Euro Stoxx 50 Index (Euro Stoxx 50) is a stock market index that represents 50 of the largest companies in the Eurozone. The trading symbol is `^STOXX50E` and as of April 2025, its constituents are:

Symbol	Name	Sector	Industry
ADS.DE	adidas	Consumer Cyclical	Footwear & Accessories
ADYEN.AS	Adyen	Technology	Software - Infrastructure

Symbol	**Name**	**Sector**	**Industry**
AD.AS	Koninklijke Ahold Delhaize	Consumer Defensive	Grocery Stores
AI.PA	L'Air Liquide	Basic Materials	Specialty Chemicals
AIR.PA	Airbus	Industrials	Aerospace & Defense
ALV.DE	Allianz	Financial Services	Insurance - Diversified
ABI.BR	Anheuser-Busch InBev	Consumer Defensive	Beverages - Brewers
ASML.AS	ASML Holding	Technology	Semiconductor Equipment & Materials
CS.PA	AXA	Financial Services	Insurance - Diversified
BAS.DE	BASF	Basic Materials	Chemicals
BAYN.DE	Bayer	Healthcare	Drug Manufacturers - General
BBVA.MC	BBVA	Financial Services	Banks - Diversified
SAN.MC	Banco Santander	Financial Services	Banks - Diversified
BMW.DE	BMW	Consumer Cyclical	Auto Manufacturers
BNP.PA	BNP Paribas	Financial Services	Banks - Regional
BN.PA	Danone	Consumer Defensive	Packaged Foods
DB1.DE	Deutsche Börse	Financial Services	Financial Data & Stock Exchanges
DHL.DE	Deutsche Post	Industrials	Integrated Freight & Logistics
DTE.DE	Deutsche Telekom	Communication Services	Telecom Services
ENEL.MI	Enel	Utilities	Utilities - Diversified

Symbol	**Name**	**Sector**	**Industry**
ENI.MI	Eni	Energy	Oil & Gas Integrated
EL.PA	EssilorLuxottica	Healthcare	Medical Instruments & Supplies
RACE.MI	Ferrari	Consumer Cyclical	Auto Manufacturers
FLTR.L	Flutter Entertainment	Consumer Cyclical	Gambling
RMS.PA	Hermès International	Consumer Cyclical	Luxury Goods
IBE.MC	Iberdrola	Utilities	Utilities - Diversified
ITX.MC	Industria de Diseño Textil	Consumer Cyclical	Apparel Retail
IFX.DE	Infineon Technologies	Technology	Semiconductors
INGA.AS	ING Groep	Financial Services	Banks - Diversified
ISP.MI	Intesa Sanpaolo	Financial Services	Banks - Regional
KER.PA	Kering	Consumer Cyclical	Luxury Goods
OR.PA	L'Oréal	Consumer Defensive	Household & Personal Products
MC.PA	LVMH Moët Hennessy - Louis Vuitton	Consumer Cyclical	Luxury Goods
MBG.DE	Mercedes-Benz Group AG	Consumer Cyclical	Auto Manufacturers

Symbol	**Name**	**Sector**	**Industry**
MUV2.DE	Münchener Rückversicherungs-Gesellschaft	Financial Services	Insurance - Reinsurance
NOKIA.HE	Nokia	Technology	Communication Equipment
NDA-FI.HE	Nordea Bank	Financial Services	Banks - Regional
RI.PA	Pernod Ricard	Consumer Defensive	Beverages - Wineries & Distilleries
PRX.AS	Prosus	Communication Services	Internet Content & Information
SAF.PA	Safran	Industrials	Aerospace & Defense
SGO.PA	Compagnie de Saint-Gobain	Industrials	Building Products & Equipment
SAN.PA	Sanofi	Healthcare	Drug Manufacturers - General
SAP.DE	SAP	Technology	Software - Application
SU.PA	Schneider Electric	Industrials	Specialty Industrial Machinery
SIE.DE	Siemens	Industrials	Specialty Industrial Machinery
STLAM.MI	Stellantis	Consumer Cyclical	Auto Manufacturers
TTE.PA	TotalEnergies	Energy	Oil & Gas Integrated
DG.PA	Vinci	Industrials	Engineering & Construction

Symbol	Name	Sector	Industry
UCG.MI	UniCredit	Financial Services	Banks - Regional
VOW.DE	Volkswagen	Consumer Cyclical	Auto Manufacturers

Table D.2: Constituents of the Euro Stoxx 50 index as of April 2025. [5]

D.1.3 FTSE 100

The Financial Times Stock Exchange 100 Index (FTSE 100) is a stock market index that represents 100 of the largest companies listed on the London Stock Exchange. The trading symbol is `^FTSE` and as of April 2025, its constituents are:

Symbol	Name	Industry
III.L	3i Group	Financial Services
ADM.L	Admiral Group	Insurance
AAF.L	Airtel Africa	Telecommunications Services
ALW.L	Alliance Witan	Investment Trusts
AAL.L	Anglo American	Mining
ANTO.L	Antofagasta	Mining
AHT.L	Ashtead Group	Support Services
ABF.L	Associated British Foods	Food & Tobacco
AZN.L	AstraZeneca	Pharmaceuticals & Biotechnology
AUTO.L	Auto Trader Group	Media
AV.L	Aviva	Life Insurance
BAB.L	Babcock International Group	Aerospace & Defence

Symbol	Name	Industry
BA.L	BAE Systems	Aerospace & Defence
BARC.L	Barclays	Banks
BTRW.L	Barratt Redrow	Household Goods & Home Construction
BEZ.L	Beazley	Insurance
BKG.L	The Berkeley Group Holdings	Household Goods & Home Construction
BP.L	BP	Oil & Gas Producers
BATS.L	British American Tobacco	Tobacco
BT-A.L	BT Group	Telecommunications Services
BNZL.L	Bunzl	Support Services
CNA.L	Centrica	Multiline Utilities
CCEP.L	Coca-Cola Europacific Partners	Beverages
CCH.L	Coca-Cola HBC	Beverages
CPG.L	Compass Group	Support Services
CTEC.L	ConvaTec Group	Health Care Equipment & Services
CRDA.L	Croda International	Chemicals
DCC.L	DCC	Support Services
DGE.L	Diageo	Beverages
DPLM.L	Diploma	Industrial Support Services
EDV.L	Endeavour Mining	Precious Metals and Mining
ENT.L	Entain	Travel & Leisure
EZJ.L	easyJet	Travel & Leisure

Symbol	Name	Industry
EXPN.L	Experian	Support Services
FCIT.L	F&C Investment Trust	Collective Investments
FRES.L	Fresnillo	Mining
GAW.L	Games Workshop Group	Leisure Goods
GLEN.L	Glencore	Mining
GSK.L	GSK	Pharmaceuticals & Biotechnology
HLN.L	Haleon	Pharmaceuticals & Biotechnology
HLMA.L	Halma	Electronic Equipment & Parts
HIK.L	Hikma Pharmaceuticals	Pharmaceuticals & Biotechnology
HSX.L	Hiscox	Non-life insurance
HWDN.L	Howden Joinery Group	Homebuilding & Construction Supplies
HSBA.L	HSBC Holdings	Banks
IHG.L	InterContinental Hotels Group	Travel & Leisure
IMI.L	IMI	Industrial Engineering
IMB.L	Imperial Brands	Tobacco
INF.L	Informa	Media
ICG.L	ICG	Financial Services
IAG.L	International Consolidated Airlines Group	Travel & Leisure
ITRK.L	Intertek Group	Support Services
JD.L	JD Sports Fashion	General Retailers
KGF.L	Kingfisher	Retailers

Symbol	Name	Industry
LAND.L	Land Securities	Real Estate Investment Trusts
LGEN.L	Legal & General	Life Insurance
LLOY.L	Lloyds Banking	Banks
LMP.L	LondonMetric Property	Real Estate Investment Trusts
LSEG.L	London Stock Exchange Group	Financial Services
MNG.L	M&G	Financial Services
MKS.L	Marks and Spencer	Food & Drug Retailing
MRO.L	Melrose Industries	Aerospace & Defence
MNDI.L	Mondi	Containers & Packaging
NG.L	National Grid	Multiline Utilities
NWG.L	NatWest	Banks
NXT.L	NEXT	General Retailers
PSON.L	Pearson	Media
PSH.L	Pershing Square Holdings	Financial Services
PSN.L	Persimmon	Household Goods & Home Construction
PHNX.L	Phoenix Group	Life Insurance
PCT.L	Polar Capital Technology	Investment Trusts
PRU.L	Prudential	Life Insurance
RKT.L	Reckitt Benckiser	Household Goods & Home Construction
REL.L	RELX	Media

Symbol	Name	Industry
RTO.L	Rentokil Initial	Support Services
RMV.L	Rightmove	Media
RIO.L	Rio Tinto	Mining
RR.L	Rolls-Royce Holdings	Aerospace & Defence
SGE.L	The Sage Group	Software & Computer Services
SBRY.L	J Sainsbury	Food & Drug Retailing
SDR.L	Schroders	Financial Services
SMT.L	Scottish Mortgage	Collective Investments
SGRO.L	SEGRO	Real Estate Investment Trusts
SVT.L	Severn Trent	Multiline Utilities
SHEL.L	Shell	Oil & Gas Producers
SMIN.L	Smiths Group	Industrial Engineering
SN.L	Smith & Nephew	Health Care Equipment & Services
SPX.L	Spirax Group	Industrial Engineering
SSE.L	SSE	Electrical Utilities & Independent Power Producers
STAN.L	Standard Chartered	Banks
STJ.L	St. James's Place	Financial Services
TW.L	Taylor Wimpey	Household Goods & Home Construction
TSCO.L	Tesco	Food & Drug Retailing
ULVR.L	Unilever	Personal Goods
UU.L	United Utilities	Multiline Utilities
UTG.L	Unite Group	Real Estate Investment Trusts
VOD.L	Vodafone Group	Mobile Telecommunications

Symbol	**Name**	**Industry**
WEIR.L	The Weir Group	Industrial Goods and Services
WTB.L	Whitbread	Retail Hospitality
WPP.L	WPP	Media

Table D.3: Constituents of the FTSE100 index as of April 2025. [6]

D.2 Commodities

The commodities market includes a variety of physical goods that are traded on exchanges. The following table lists a sample of 6 commodities:

Symbol	**Name**
CL=F	Crude Oil
NG=F	Natural Gas
GC=F	Gold
SI=F	Silver
ALI=F	Aluminum
HG=F	Copper

Table D.4: 6 Commodities Futures Contracts

D.3 Currencies

The currencies market includes a variety of currency pairs that are traded on exchanges. The following table lists a sample of 10 currency pairs, selected for their trading volume. Note that the trading symbols all have United States Dollar (USD) as the quote currency.

Symbol	**Name**
EURUSD=X	Euro (EUR)/USD
GBPUSD=X	British Pound (GBP)/USD
JPYUSD=X	Japanese Yen (JPY)/USD
AUDUSD=X	Australian Dollar (AUD)/USD
CADUSD=X	Canadian Dollar (CAD)/USD
CNYUSD=X	Chinese Yuan (CNY)/USD
CHFUSD=X	Swiss Franc (CHF)/USD
HKDUSD=X	Hong Kong Dollar (HKD)/USD
KRWUSD=X	South Korean Won (KRW)/USD
INRUSD=X	Indian Rupee (INR)/USD

Table D.5: 10 Currency Pairs

Appendix E

Default Hyper-parameter Selection

The choice of hyper-parameters is crucial for the performance of DRL algorithms. However, it is computationally expensive to perform hyper-parameter tuning for all combinations of algorithms, datasets and environment representations. Therefore, the default hyper-parameters used in the experiments are outlined in Table 4.2. These hyper-parameters were selected based on preliminary tests conducted on a small dataset of five tickers (AAPL, CSCO, HON, MSFT, V) sampled from the DJIA and only the open, close, high and low prices as the environment representation.

The hyper-parameters were tuned according to the specifications in Table C.1 and the results were used to inform the default settings, outlined in Table 4.2. For each of the five DRL algorithms, the hyper-parameter search was performed using Bayesian optimisation using the `wandb` library and a maximum number of runs set to 20. The training set was data from January 2016 to December 2022, the validation set started in January 2023 and ended in December 2023, and the test set was between January 2024 and June 2025. The dataset split is visualised in Figure E.1.

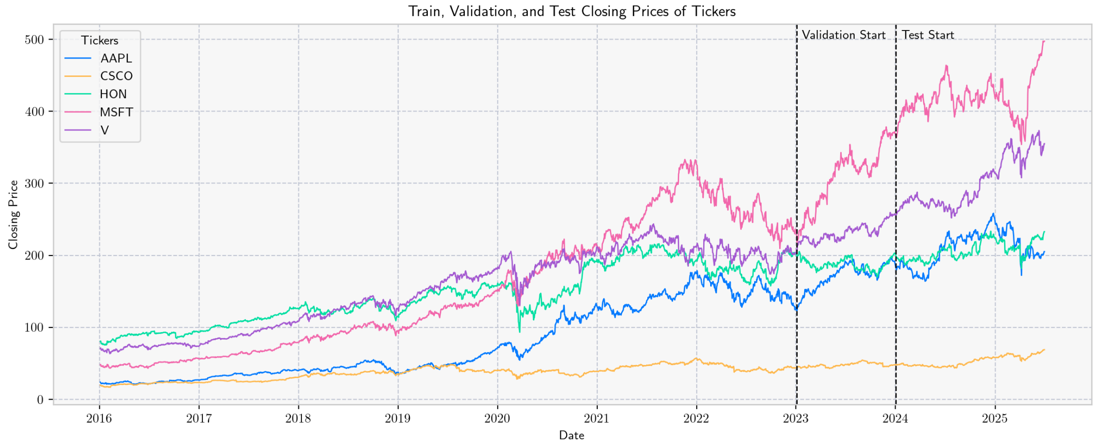

Figure E.1: Train-Validation-Test Split for the Hyper-parameter tuning on a sample of five assets from the Dow Jones Industrial Average index.

The `wandb` library provides an interactive website to visualise the results of the hyper-parameters in terms of the metric chosen for the optimisation and, if applicable, any other metrics that were chosen to be reported. In this case, the optimisation metric was the Sharpe ratio, as it balances the trade-off between risk and return, and additionally, the cumulative return was also reported. Figure E.2 shows the reported results for the hyper-parameter tuning of the A2C algorithm. Within the figure, there are three charts where the left one shows the Sharpe ratio over all sweeps, the middle one shows the cumulative return, and the right one shows the hyper-parameters that were tuned and the resulting optimisation metric.

Figure E.2: Hyper-parameter tuning results for the A2C algorithm.

The hyper-parameter tuning process was repeated for the other four algorithms, resulting in the hyper-parameters shown in Table 4.2. Since the search had been done in the validation dataset, the best-performing models were evaluated and benchmarked against the test dataset to assess their generalisation performance, whose results are presented in Table E.1.

Dataset	**Cumulative return**	**Annualised return**	**Annualised volatility**	**Sharpe ratio**	**Max drawdown**
A2C	0.2598	0.1679	0.1968	0.8859	-0.2186
PPO	0.3375	0.2158	0.1697	1.2364	-0.1630
DDPG	0.3494	0.2231	0.1763	1.2300	-0.1684
TD3	0.3401	0.2174	0.1731	1.2227	-0.1598
SAC	0.2659	0.1717	0.1762	0.9866	-0.1825
Mean	0.1503	0.0989	0.1896	0.5923	-0.1972
Min	0.3609	0.2308	0.1681	1.3193	-0.1559
Momentum	0.1635	0.1074	0.1868	0.6396	-0.2025
Equal	0.3123	0.2010	0.1758	1.1290	-0.1762

Table E.1: Results of hyper-parameter tuning on the test dataset.

Appendix F

Evaluation Metrics

The following list outlines additional evaluation metrics that can be used to assess the performance of trading strategies.

- Calmar ratio: A risk-adjusted performance measure that compares the average annual compounded rate of return to the maximum drawdown, providing insight into how much return an investor receives for each unit of risk taken.

$$\text{Calmar Ratio} = \frac{\text{Average Annualised Return}}{\text{Max Drawdown}} \tag{F.1}$$

- Stability: A measure that determines the R-square of a linear fit to the cumulative log returns, to indicate how well returns follow a linear trend over time.

$$\text{Stability} = R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{F.2}$$

where y_i is the observed cumulative log return, $\hat{y}_i$ is the predicted cumulative log return from the linear fit, $\bar{y}$ is the mean of the observed cumulative log returns and n is the number of observations.

- Omega ratio: A probability-weighted ratio that compares the gains versus losses relative to a threshold return target, either set to zero or the risk-free rate. It is a proxy for the likelihood of achieving returns above a certain threshold compared to the likelihood of experiencing losses below that threshold.

$$\text{Omega Ratio} = \Omega\left(\theta\right) = \frac{\int_{\theta}^{\infty}\left(1 - F(r)\right)dr}{\int_{-\infty}^{\theta} F(r)dr} \tag{F.3}$$

where θ is the threshold return and $F(r)$ is the cumulative distribution function of returns.

- Sortino ratio: A modification of the Sharpe ratio that only considers downside risk, measuring the risk-adjusted return of an investment by comparing the excess return to the downside deviation.

$$\text{Sortino Ratio} = \frac{R_p - R_f}{\sigma_d} \tag{F.4}$$

where R_p is the annualised return of the portfolio, R_f is the annualised risk-free rate, and σ_d is the downside deviation of the portfolio returns, i.e. the standard deviation of only the negative returns.

- Skewness: A measure of the asymmetry of the return distribution, indicating whether returns have a tendency toward positive or negative extremes.

$$\text{Skewness} = \frac{\sum_{i=1}^{n}(R_i - \mu)^3}{(n-1)\cdot\sigma^3} \tag{F.5}$$

where R_i is the return of the portfolio at time i, μ is the mean return, σ is the standard deviation of returns, and n is the number of observations.

- Kurtosis: A measure of the "tailedness" of the return distribution, indicating the frequency of extreme values compared to a normal distribution.

$$\text{Kurtosis} = \frac{\sum_{i=1}^{n}(R_i - \mu)^4}{(n-1)\cdot\sigma^4} \tag{F.6}$$

where R_i is the return of the portfolio at time i, μ is the mean return, σ is the standard deviation of returns, and n is the number of observations.

- Tail ratio: A measure of the relative magnitude of extreme positive returns compared to extreme negative returns, calculated as the ratio between the right (95th percentile) and the left (5th percentile) tails of the distribution of returns.

$$\text{Tail Ratio} = \frac{\text{Right Tail (95th Percentile)}}{\text{Left Tail (5th Percentile)}} \tag{F.7}$$

- Daily Value at Risk (VaR): A measure of the potential loss that could occur in a portfolio over a single day at a specified confidence level, typically 95%. The computation is done using the variance-covariance calculation:

$$\text{Daily VaR} = (R_p - z\cdot\sigma_p)\times V_p \tag{F.8}$$

where R_p is the expected daily return, z is the z-score of the desired level of confidence, σ_p is the daily standard deviation of portfolio returns, and V_p is the value of the portfolio.

Appendix G

Experiment Results

In this Appendix, the results of the experiments conducted to evaluate the performance of the implemented algorithms under changing conditions are presented. Only the tables that do not fit in the main Results chapter 4 are included here.

G.1 Experiment: Algorithm Comparison

The following tables summarise the results of the algorithm comparison experiment conducted on the 5 datasets with OHLCV prices as the environment representation. The results for the A2C algorithm are presented in Table 4.4 in Section 4.4, whereas for the PPO, DDPG, TD3 and SAC algorithms, the results are presented in the following tables.

Dataset	**Cumulative return**	**Annualised return**	**Annualised volatility**	**Sharpe ratio**	**Max drawdown**
Dow Jones 30	0.2035	0.3174	0.1461	1.3410	-0.1503
Euro Stoxx 50	0.1557	0.2438	0.1549	1.0117	-0.1706

Dataset	Cumulative return	Annualised return	Annualised volatility	Sharpe ratio	Max drawdown
FTSE 100	0.1246	0.1931	0.1244	1.0063	-0.1315
Commodities	0.2531	0.3990	0.1941	1.2600	-0.1522
Currencies	-0.0011	-0.0017	0.0493	0.0014	-0.0726

Table G.1: Algorithm comparison results for the PPO implementation.

Dataset	Cumulative return	Annualised return	Annualised volatility	Sharpe ratio	Max drawdown
Dow Jones 30	0.2206	0.3454	0.1526	1.3826	-0.1560
Euro Stoxx 50	0.1292	0.2011	0.1556	0.8592	-0.1774
FTSE 100	0.1336	0.2076	0.1229	1.0822	-0.1248
Commodities	0.2168	0.3391	0.1811	1.1745	-0.1237
Currencies	0.0014	0.0021	0.0559	0.0528	-0.0802

Table G.2: Algorithm comparison results for the DDPG implementation.

Dataset	Cumulative return	Annualised return	Annualised volatility	Sharpe ratio	Max drawdown
Dow Jones 30	0.2478	0.3902	0.1567	1.4911	-0.1567
Euro Stoxx 50	0.1547	0.2423	0.1487	1.0420	-0.1645
FTSE 100	0.1325	0.2058	0.1231	1.0729	-0.1263
Commodities	0.2768	0.4386	0.1792	1.4537	-0.1224
Currencies	0.0026	0.0038	0.0438	0.0807	-0.0639

Table G.3: Algorithm comparison results for the TD3 implementation.

Dataset	Cumulative return	Annualised return	Annualised volatility	Sharpe ratio	Max drawdown
Dow Jones 30	0.1457	0.2243	0.1508	0.9769	-0.1750
Euro Stoxx 50	0.1761	0.2771	0.1520	1.1437	-0.1718
FTSE 100	0.1319	0.2048	0.1279	1.0326	-0.1404
Commodities	0.0027	0.0041	0.0480	0.0806	-0.0704
Currencies	0.2563	0.4044	0.1778	1.3727	-0.1356

Table G.4: Algorithm comparison results for the SAC implementation.

G.2 Experiment: Environment Representation

The table G.5 summarises the performance, according to the cumulative return, of the five algorithms in the four possible different environment representations: OHLCV prices, OHLCV prices with indicators, OHLCV prices with covariance, and OHLCV prices with indicators and covariance. The results are presented for three datasets: DowJones30, Commodities, and Currencies.

Algorithm	Dataset	Simple	Indicators	Covariance	Complete
A2C	DowJones30	0.3387	0.3139	0.3464	0.3239
	Commodities	0.3695	0.4133	0.3883	0.4098
	Currencies	-0.0017	-0.0071	-0.0069	-0.0023
PPO	DowJones30	0.3174	0.2937	0.3221	0.2713
	Commodities	0.3990	0.3565	0.3689	0.3751
	Currencies	-0.0017	0.0033	0.0029	0.0083

Table G.5: Environment representation experiment comparison according to the cumulative return. The colour correspond to the best performing per row, with blue for the DowJones30 dataset and green for the Commodities dataset.

Algorithm	**Dataset**	**Simple**	**Indicators**	**Covariance**	**Complete**
DDPG	DowJones30	0.3454	0.3663	0.3128	0.3301
	Commodities	0.3391	0.3339	0.3616	0.4211
	Currencies	0.0021	0.0032	-0.0002	-0.0006
TD3	DowJones30	0.3902	0.3456	0.2571	0.2787
	Commodities	0.4386	0.3632	0.3292	0.4312
	Currencies	0.0038	-0.0022	-0.0023	-0.005
SAC	DowJones30	0.2243	0.3189	0.3774	0.3162
	Commodities	0.4044	0.3371	0.3958	0.3025
	Currencies	0.0041	0.0114	-0.0117	0.0179

Table G.5: Environment representation experiment comparison according to the cumulative return. The colour correspond to the best performing per row, with blue for the DowJones30 dataset and green for the Commodities dataset.

Appendix H

Explainability Results

H.1 Feature Importance

Figure H.1 shows the top features grouped by asset according to the feature importance of the surrogate model.

Figure H.2 shows the mean feature importance for each type of feature, i.e. open, close, high and low prices, over the assets.

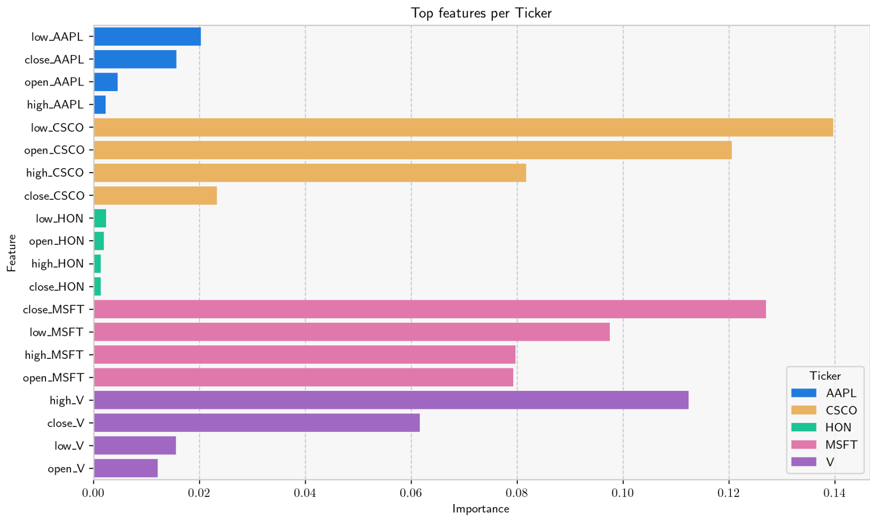

Figure H.1: Top features grouped by asset from the surrogate model according to feature importance.

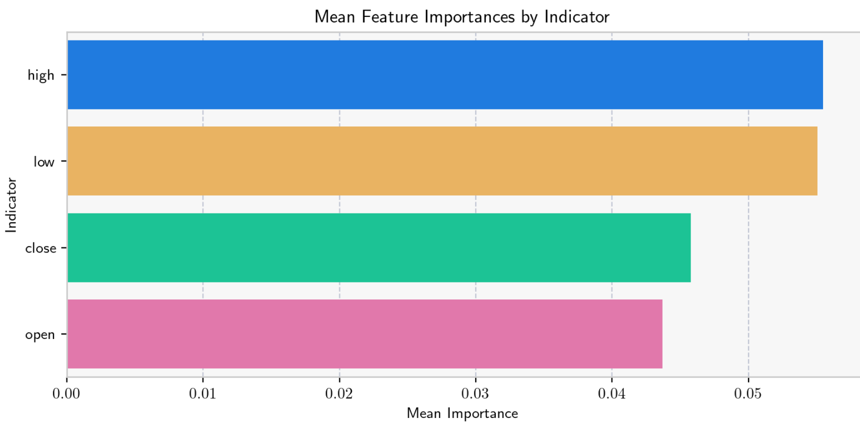

Figure H.2: Mean feature importance per type of feature from the surrogate model.

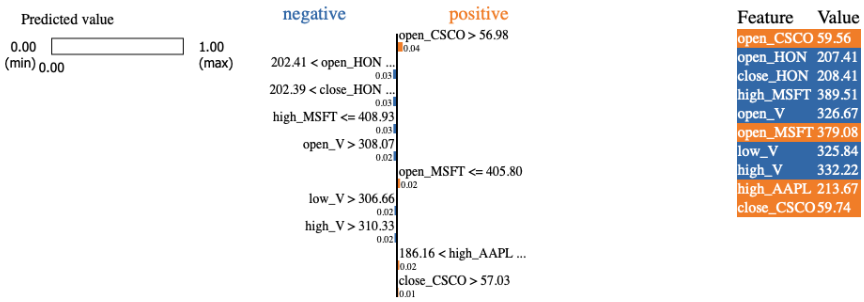

Figure H.3: LIME explanations for the A2C algorithm at the first time step of the test dataset for the AAPL asset.

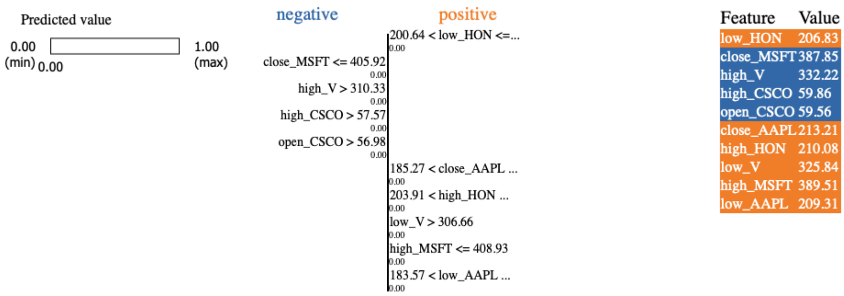

Figure H.4: LIME explanations for the A2C algorithm at the first time step of the test dataset for the CSCO asset.

H.2 Local Importance Model-Agnostic Explanation Results

With the LIME analysis, the explanations for the A2C algorithm are provided. Since these are local observations, the following figures provide the explanation for the first time step on the test dataset of the assets not included in the main text. Values coloured in orange represent features that contribute positively to the prediction, while blue values indicate features that contribute negatively.

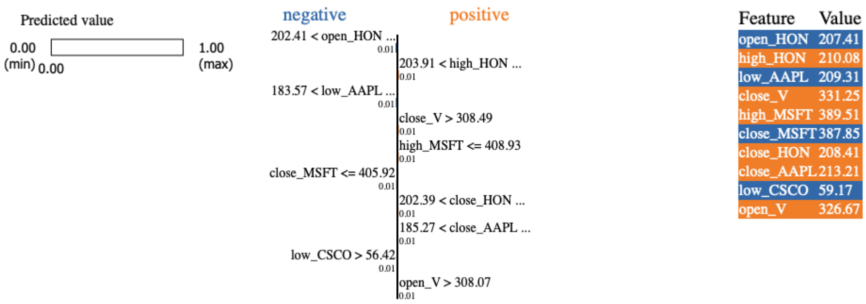

Figure H.5: LIME explanations for the A2C algorithm at the first time step of the test dataset for the HON asset.

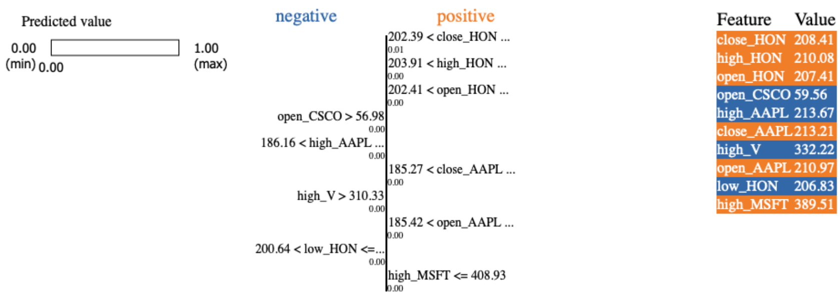

Figure H.6: LIME explanations for the A2C algorithm at the first time step of the test dataset for the V asset.

Appendix I

Code

The following section illustrates the code implementation of the project. It includes the main components of the codebase such as data download and pre-processing, model training and evaluation, benchmarking and explainability techniques.

The code is structured in a modular way, where each module corresponds to a specific functionality. The main components are:

- **Configuration**: This module manages the configuration settings that define the list of assets, the training-test periods, and other hyper-parameters.
- **Data Download and Pre-processing:** This module handles the downloading of datasets, cleaning, and pre-processing steps necessary for training the models.
- **Agents**: This module contains the implementation of the DRL agents capable of interacting with the environment, learning from it, and making decisions based on the observations received.
- **Environments**: This module defines the various environments in which the agents operate, including the state and action spaces, as well as the reward functions.

- **Benchmarking**: This module contains the implementation of the benchmarking strategies as well as the evaluation metrics used to assess the performance of the agents.
- **Explainability**: This module implements the explainability techniques used to interpret the agents' decisions and understand their behaviour.
- **Visualiser**: This module provides visualisations tools for the data, agents' performance, benchmarking comparisons and explainability results.
- **Optimisation**: This module contains the integration with hyper-parameter tuning libraries to find optimal configurations and track the experiments.

The modules are implemented as Python packages, each containing the necessary code and resources for its specific functionality. Additionally, the project contains a folder titled `examples`, where the main Jupyter notebooks to perform each of the separate tasks of this project are implemented. The directory structure is organized as follows:

```
xdl-portfolio/
|
+-- agents/
|   +-- __init__.py
|   +-- drl_agent.py
|   +-- tb_callback.py
|
+-- config/
|   +-- __init__.py
|   +-- config.py
|   +-- config_indicators.py
|   +-- config_tickers.py
```

```
|   +-- config_models.py
|
+-- environments/
|   +-- __init__.py
|   +-- env_portfolio_optimisation.py
|
+-- examples/
|   +-- findownloader.ipynb
|   +-- finpreprocessor.ipynb
|   +-- portfolio_optimisation.ipynb
|   +-- backtesting.ipynb
|   +-- hyperparameter_tuning.ipynb
|   +-- explainability.ipynb
|
+-- explainability/
|   +-- __init__.py
|   +-- explainability.py
|   +-- feature_importance.py
|   +-- lime_explainability.py
|   +-- shap_explainability.py
|
+-- optimisation/
|   +-- __init__.py
|   +-- wandb_opt.py
|
+-- benchmark/
|   +-- __init__.py
```

```
|   +-- portfolio_benchmark.py
|
+-- preprocessor/
|   +-- __init__.py
|   +-- findata_downloader.py
|   +-- findata_preprocessor.py
|
+-- visualisation/
|   +-- __init__.py
|   +-- benchmark_visualiser.py
|   +-- findata_visualiser.py
|   +-- model_visualiser.py
|   +-- shap_visualiser.py
|   +-- style.mplstyle
+-- README.md
+-- requirements.txt
```

The code is available as a GitHub repository at XDL-Portfolio.

I.1 Configuration Module

Listing 1: config.py

```python
from config import config_indicators, config_tickers

START_DATE = "2016-01-01"
END_DATE = "2025-07-01"

DATA_DIR = "data"
PLOT_DIR = "figures"
RESULTS_DIR = "results"
MODELS_DIR = "models"
LOGS_DIR = "logs"

```

```python
INITIAL_AMOUNT = 100000  # Initial capital for the portfolio

TEST_NAME = "test"
DOW_30_NAME = "dow30"
EURO_STOXX_50_NAME = "eurostoxx50"
FTSE_100_NAME = "ftse100"
SP_500_NAME = "sp500"
FX_NAME = "currencies"
COMMODITY_NAME = "futures"

DOW_30_INDEX = "^DJI"
EURO_STOXX_50_INDEX = "^STOXX50E"
FTSE_100_INDEX = "^FTSE"
SP_500_INDEX = "^GSPC"

EXCHANGE_NYSE = "XNYS"  # New York Stock Exchange
EXCHANGE_LSE = "XLON"  # London Stock Exchange
EXCHANGE_DAX = "XFRA"  # Frankfurt Stock Exchange

TRAIN_START_DATE = START_DATE
TRAIN_END_DATE = "2022-12-31"
VAL_START_DATE = "2023-01-01"
VAL_END_DATE = "2023-12-31"
TEST_START_DATE = "2024-01-01"
TEST_END_DATE = END_DATE

# Configuration for indicators
USE_TECHNICAL_INDICATORS = True
USE_MACROECONOMIC_INDICATORS = True
# Configuration for covariance features
USE_COVARIANCE_FEATURES = False

# Environment representation columns
ENVIRONMENT_COLUMNS = [
    "open",
    "high",
    "low",
    "close",
    "volume",
]

if USE_TECHNICAL_INDICATORS:
    ENVIRONMENT_COLUMNS += list(config_indicators.TECHNICAL_INDICATORS.keys())

if USE_MACROECONOMIC_INDICATORS:
    MACROECONOMIC_INDICATORS = (
        config_indicators.MACROECONOMIC_INDICATORS_DEFAULT
    )

    # Remove non-letter characters from string
    ENVIRONMENT_COLUMNS += list(
        map(
            lambda x: "".join(
                ch
                for ch in x.split(".", 1)[0].lower()
```

```python
                if ch.isalnum() or ch == "_"
            ),
            list(MACROECONOMIC_INDICATORS.keys()),
        )
    )

if USE_COVARIANCE_FEATURES:
    ENVIRONMENT_COLUMNS += ["covariance"]

# Tickers configurations
TICKERS = config_tickers.TEST_TICKERS
INDEX = None
TICKERS_NAME = TEST_NAME
EXCHANGE = EXCHANGE_NYSE

# Set dataset name based on the configuration
if (
    USE_TECHNICAL_INDICATORS
    and USE_MACROECONOMIC_INDICATORS
    and USE_COVARIANCE_FEATURES
):
    DATASET_NAME = "dataset-indicators-covariance"
elif USE_TECHNICAL_INDICATORS and USE_MACROECONOMIC_INDICATORS:
    DATASET_NAME = "dataset-indicators"
elif USE_COVARIANCE_FEATURES:
    DATASET_NAME = "dataset-covariance"
else:
    DATASET_NAME = "simple-dataset"

WANDB_ENTITY = "xdl-team"
WANDB_PROJECT = "xdl-portfolio"

SWEEP_CONFIG = {
    "method": "bayes",
    "metric": {"name": "sharpe_ratio", "goal": "maximize"},
    "early_terminate": {
        "type": "hyperband",
        "s": 2,
        "eta": 2,
        "max_iter": 10,
    },
}
```

Listing 2: config_tickers.py

```python
TEST_TICKERS = [
    "AAPL",
    "CSCO",
    "MSFT",
    "HON",
    "V",
]
```

```
DOW_30_TICKERS = [
    "AXP",
    "AMGN",
    "AAPL",
    "AMZN",
    "BA",
    "CAT",
    "CRM",
    "CSCO",
    "CVX",
    "DIS",
    "GS",
    "HD",
    "HON",
    "IBM",
    "JNJ",
    "JPM",
    "KO",
    "MCD",
    "MMM",
    "MRK",
    "MSFT",
    "NKE",
    "NVDA",
    "PG",
    "SHW",
    "TRV",
    "UNH",
    "V",
    "VZ",
    "WMT",
]

EURO_STOXX_50_TICKERS = [
    "ADS.DE",
    "ADYEN.AS",
    "AD.AS",
    "AI.PA",
    "AIR.PA",
    "ALV.DE",
    "ABI.BR",
    "ASML.AS",
    "CS.PA",
    "BAS.DE",
    "BAYN.DE",
    "BBVA.MC",
    "SAN.MC",
    "BMW.DE",
    "BNP.PA",
    "BN.PA",
    "DB1.DE",
    "DHL.DE",
    "DTE.DE",
    "ENEL.MI",
```

```
    "ENI.MI",
    "EL.PA",
    "RACE.MI",
    "FLTR.L",
    "RMS.PA",
    "IBE.MC",
    "ITX.MC",
    "IFX.DE",
    "INGA.AS",
    "ISP.MI",
    "KER.PA",
    "OR.PA",
    "MC.PA",
    "MBG.DE",
    "MUV2.DE",
    "NOKIA.HE",
    "NDA-FI.HE",
    "RI.PA",
    "PRX.AS",
    "SAF.PA",
    "SGO.PA",
    "SAN.PA",
    "SAP.DE",
    "SU.PA",
    "SIE.DE",
    "STLAM.MI",
    "TTE.PA",
    "DG.PA",
    "UCG.MI",
    "VOW.DE",
]

FTSE_100_TICKERS = [
    "III.L",
    "ADM.L",
    "AAF.L",
    "ALW.L",
    "AAL.L",
    "ANTO.L",
    "AHT.L",
    "ABF.L",
    "AZN.L",
    "AUTO.L",
    "AV.L",
    "BAB.L",
    "BA.L",
    "BARC.L",
    "BTRW.L",
    "BEZ.L",
    "BKG.L",
    "BP.L",
    "BATS.L",
    "BT-A.L",
    "BNZL.L",
    "CNA.L",
```

```
    "CCEP.L",
    "CCH.L",
    "CPG.L",
    "CTEC.L",
    "CRDA.L",
    "DCC.L",
    "DGE.L",
    "DPLM.L",
    "EDV.L",
    "ENT.L",
    "EZJ.L",
    "EXPN.L",
    "FCIT.L",
    "FRES.L",
    "GAW.L",
    "GLEN.L",
    "GSK.L",
    "HLN.L",
    "HLMA.L",
    "HIK.L",
    "HSX.L",
    "HWDN.L",
    "HSBA.L",
    "IHG.L",
    "IMI.L",
    "IMB.L",
    "INF.L",
    "ICG.L",
    "IAG.L",
    "ITRK.L",
    "JD.L",
    "KGF.L",
    "LAND.L",
    "LGEN.L",
    "LLOY.L",
    "LMP.L",
    "LSEG.L",
    "MNG.L",
    "MKS.L",
    "MRO.L",
    "MNDI.L",
    "NG.L",
    "NWG.L",
    "NXT.L",
    "PSON.L",
    "PSH.L",
    "PSN.L",
    "PHNX.L",
    "PCT.L",
    "PRU.L",
    "RKT.L",
    "REL.L",
    "RTO.L",
    "RMV.L",
    "RIO.L",
```

```python
    "RR.L",
    "SGE.L",
    "SBRY.L",
    "SDR.L",
    "SMT.L",
    "SGRO.L",
    "SVT.L",
    "SHEL.L",
    "SMIN.L",
    "SN.L",
    "SPX.L",
    "SSE.L",
    "STAN.L",
    "STJ.L",
    "TW.L",
    "TSCO.L",
    "ULVR.L",
    "UU.L",
    "UTG.L",
    "VOD.L",
    "WEIR.L",
    "WTB.L",
    "WPP.L",
]

FX_TICKERS = [
    "EURUSD=X",
    "GBPUSD=X",
    "JPYUSD=X",
    "AUDUSD=X",
    "CADUSD=X",
    "CNYUSD=X",
    "CHFUSD=X",
    "HKDUSD=X",
    "KRWUSD=X",
    "INRUSD=X",
]

COMMODITY_TICKERS = [
    "CL=F",  # Crude oil
    "NG=F",  # Natural gas
    "GC=F",  # Gold
    "SI=F",  # Silver
    "ALI=F",  # Aluminum
    "HG=F",  # Copper
]
```

Listing 3: config_indicators.py

```python
# List of technical indicators
TECHNICAL_INDICATORS = {
    "close_5_sma": "Simple Moving Average over 5 days (1 week)",
```

```python
    "close_10_sma": "Simple Moving Average over 10 days (2 weeks)",
    "close_20_sma": "Simple Moving Average over 20 days (1 month)",
    "close_5_ema": "Exponential Moving Average over 5 days (1 week)",
    "close_10_ema": "Exponential Moving Average over 10 days (2 weeks)",
    "macd": "Moving Average Convergence Divergence",
    "rsi": "Relative Strength Index over 14 days",
    "cci": "Commodity Channel Index",
    "boll_ub": "Bollinger Bands Upper Band",
    "boll_lb": "Bollinger Bands Lower Band",
    "atr": "Average True Range",
    "adx": "Average Directional Index",
    "close_10_roc": "Rate of Change over 10 days (2 weeks)",
}

# List of macroeconomic indicators
MACROECONOMIC_INDICATORS_DEFAULT = {
    "^VIX": "Volatility Index (VIX)",
    "DX-Y.NYB": "US Dollar Index (DXY)",
    "^IRX": "3-Month Treasury Yield (IRX)",
    "^FVX": "5-Year Treasury Yield (FVX)",
    "^TNX": "10-Year Treasury Yield (TNX)",
}

MACROECONOMIC_INDICATORS_DW30 = {
    "^VXD": "Volatility Index (VXD)",
    "DX-Y.NYB": "US Dollar Index (DXY)",
    "^IRX": "3-Month Treasury Yield (IRX)",
    "^FVX": "5-Year Treasury Yield (FVX)",
    "^TNX": "10-Year Treasury Yield (TNX)",
}

MACROECONOMIC_INDICATORS_EUROSTOXX50 = {
    "^XDE": "Euro Currency Index (EXY)",
}

MACROECONOMIC_INDICATORS_FTSE100 = {
    "^XDB": "British Pound Currency Index (BXY)",
}
```

Listing 4: config_models.py

```python
# Stable Baselines DRL Models
from stable_baselines3 import A2C, DDPG, PPO, SAC, TD3

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

```

```python
# Training parameters for stock trading task
MODEL_KWARGS_STOCK = {
    "a2c": {
        "n_steps": 40,
        "ent_coef": 0.0003,
        "learning_rate": 0.003,
    },
    "ppo": {
        "n_steps": 512,
        "ent_coef": 0.0005,
        "learning_rate": 0.0015,
        "batch_size": 64,
    },
    "ddpg": {
        "batch_size": 256,
        "buffer_size": 200000,
        "learning_rate": 0.005,
    },
    "td3": {
        "batch_size": 128,
        "buffer_size": 500000,
        "learning_rate": 0.001,
    },
    "sac": {
        "batch_size": 64,
        "buffer_size": 500000,
        "learning_rate": 0.001,
        "learning_starts": 2000,
        "ent_coef": "auto_0.1",
    },
}

# Training parameters for portfolio optimisation task
MODEL_KWARGS_PORTFOLIO = {
    "a2c": {
        "n_steps": 10,
        "ent_coef": 0.005,
        "learning_rate": 0.0004,
    },
    "ppo": {
        "n_steps": 2048,
        "ent_coef": 0.005,
        "learning_rate": 0.001,
        "batch_size": 128,
    },
    "ddpg": {
        "batch_size": 128,
        "buffer_size": 50000,
        "learning_rate": 0.001,
    },
    "td3": {
        "batch_size": 100,
        "buffer_size": 1000000,
        "learning_rate": 0.001,
    },
```

```
    "sac": {
        "batch_size": 128,
        "buffer_size": 1000000,
        "learning_rate": 0.0003,
        "learning_starts": 100,
        "ent_coef": "auto_0.1",
    },
}

# Training visualisation config
train_visualisation_config = {
    "a2c": {
        "x": "time/iterations",
        "y": [
            "train/reward",
            "train/policy_loss",
            "train/entropy_loss",
            "train/value_loss",
        ],
        "title": [
            "Iterations vs Reward",
            "Iterations vs Policy Loss",
            "Iterations vs Entropy loss",
            "Iterations vs Value loss",
        ],
    },
    "ppo": {
        "x": "time/iterations",
        "y": [
            "train/reward",
            "train/policy_gradient_loss",
            "train/entropy_loss",
            "train/value_loss",
        ],
        "title": [
            "Iterations vs Reward",
            "Iterations vs Policy Gradient Loss",
            "Iterations vs Entropy loss",
            "Iterations vs Value loss",
        ],
    },
    "ddpg": {
        "x": "time/episodes",
        "y": [
            "train/reward",
            "train/actor_loss",
            "train/critic_loss",
        ],
        "title": [
            "Episodes vs Reward",
            "Episodes vs Actor Loss",
            "Episodes vs Critic Loss",
        ],
    },
    "td3": {
```

```
        "x": "time/episodes",
        "y": [
            "train/reward",
            "train/actor_loss",
            "train/critic_loss",
        ],
        "title": [
            "Episodes vs Reward",
            "Episodes vs Actor Loss",
            "Episodes vs Critic Loss",
        ],
    },
    "sac": {
        "x": "time/episodes",
        "y": [
            "train/reward",
            "train/actor_loss",
            "train/critic_loss",
            "train/ent_coef_loss",
        ],
        "title": [
            "Episodes vs Reward",
            "Episodes vs Actor Loss",
            "Episodes vs Critic Loss",
            "Episodes vs Entropy Coefficient Loss",
        ],
    },
}

# Model hyperparameter sweep configuration
MODEL_SWEEP_CONFIG = {
    "a2c": {
        "n_steps": {"values": [5, 10, 20, 30, 40]},
        "ent_coef": {
            "distribution": "uniform",
            "min": 1e-8,
            "max": 1e-3,
        },
        "learning_rate": {
            "distribution": "uniform",
            "min": 1e-5,
            "max": 1e-2,
        },
    },
    "ppo": {
        "n_steps": {"values": [128, 256, 512, 1024, 2048]},
        "ent_coef": {
            "distribution": "uniform",
            "min": 1e-8,
            "max": 1e-3,
        },
        "learning_rate": {
            "distribution": "uniform",
            "min": 1e-5,
            "max": 1e-2,
```

```
        },
        "batch_size": {"values": [32, 64, 128, 256, 512]},
    },
    "ddpg": {
        "batch_size": {"values": [64, 128, 256]},
        "buffer_size": {"values": [50000, 100000, 200000, 500000]},
        "learning_rate": {
            "distribution": "uniform",
            "min": 1e-5,
            "max": 1e-2,
        },
    },
    "td3": {
        "batch_size": {"values": [64, 100, 128, 256]},
        "buffer_size": {"values": [500000, 1000000, 2000000]},
        "learning_rate": {
            "distribution": "uniform",
            "min": 1e-5,
            "max": 1e-2,
        },
    },
    "sac": {
        "batch_size": {"values": [32, 64, 128]},
        "buffer_size": {"values": [100000, 500000, 1000000, 2000000]},
        "learning_rate": {
            "distribution": "uniform",
            "min": 1e-5,
            "max": 1e-2,
        },
        "learning_starts": {"values": [500, 1000, 2000, 5000]},
        "ent_coef": {"values": ["auto", "auto_0.1", "auto_0.01"]},
    },
}
```

I.2 Data Download and Pre-processing Module

Listing 5: findata_downloader.py

```
from typing import List

import pandas as pd
import yfinance as yf


class FinancialDataDownloader:

    def __init__(self, start_date: str, end_date: str) -> None:
        """
        Initialise the FinancialDataDownloader with a date range.
        :param start_date: Start date for data download in 'YYYY-MM-DD' format.
        :param end_date: End date for data download in 'YYYY-MM-DD' format.
```

```python
        """
        self.start_date = start_date
        self.end_date = end_date
        self.data = None

    def download_data(self, tickers: List[str]) -> pd.DataFrame:
        """
        Download financial data for the specified tickers within the date range.
        :param tickers: List of stock tickers to download data for.
        :return: DataFrame containing the downloaded financial data.
        :raises Exception: If an error occurs during data download.
        :raises ValueError: If no data is returned from the download.
        """
        if self.data is not None:
            print("Data already downloaded. Returning existing data.")
            return self.data

        try:
            data = yf.download(
                tickers,
                start=self.start_date,
                end=self.end_date,
                group_by="Ticker",
                auto_adjust=True,
                progress=False,
            )
        except Exception as e:
            raise Exception(f"An error occurred while downloading data: {e}")

        if data is None:
            raise ValueError(
                "No data returned from the download. Please check the tickers and date range."
            )

        # Flatten the MultiIndex columns
        data = (
            data.stack(level=0, future_stack=True)
            .rename_axis(["Date", "Ticker"])
            .reset_index(level=1)
        )

        # Reset index to have a clean DataFrame
        data.reset_index(inplace=True)

        # Rename index columns
        data.columns.rename("", inplace=True)

        # Convert column names to lowercase
        data.columns = [col.lower() for col in data.columns]

        # Rename ticker column to 'tic'
        data.rename(columns={"ticker": "tic"}, inplace=True)

```

```python
        self.data = data
        print(
            f"Data downloaded for {len(tickers)} tickers from {self.start_date} to
            ↪  {self.end_date}."
        )
        return data

    def save_data(self, directory: str, filename: str) -> None:
        """
        Save the downloaded data to a CSV file.
        :param directory: Directory where the CSV file will be saved.
        :param filename: Name of the CSV file (without extension).
        :raises ValueError: If there is no data to save.
        """
        if self.data is None or self.data.empty:
            raise ValueError("No data to save. Please download data first.")

        file_path = f"{directory}/{filename}.csv"
        self.data.to_csv(file_path, index=False)
        print(f"Data saved to {file_path}")

    def load_data(self, directory: str, filename: str) -> pd.DataFrame:
        """
        Load financial data from a CSV file.
        :param directory: Directory where the CSV file is located.
        :param filename: Name of the CSV file (without extension).
        :return: DataFrame containing the loaded financial data.
        :raises FileNotFoundError: If the specified file does not exist.
        :raises Exception: If an error occurs while loading the data.
        """
        file_path = f"{directory}/{filename}.csv"

        try:
            data = pd.read_csv(file_path)

            # Convert date columns to datetime format if they exist
            data["date"] = pd.to_datetime(data["date"])

            # Sort the data by date, tic
            data.sort_values(by=["date", "tic"], inplace=True)
            # Reset index after sorting
            data.reset_index(drop=True, inplace=True)

            print(f"Data loaded from {file_path}")
            self.data = data

            return data

        except FileNotFoundError:
            raise FileNotFoundError(f"No data file found at {file_path}")

        except Exception as e:
            raise Exception(f"An error occurred while loading data: {e}")
```

Listing 6: `findata_preprocessor.py`

```python
import itertools
from datetime import datetime
from typing import List, Optional

import exchange_calendars as ecals
import pandas as pd
from stockstats import StockDataFrame

from preprocessor.findata_downloader import FinancialDataDownloader


class FinancialDataPreprocessor:
    def __init__(self, start_date: str, end_date: str) -> None:
        """
        Initialses the FinancialDataPreprocessor with a date range.
        :param start_date: Start date for the data processing in 'YYYY-MM-DD' format.
        :param end_date: End date for the data processing in 'YYYY-MM-DD' format.
        """
        self.start_date = start_date
        self.end_date = end_date

    def preprocess(
        self,
        data: pd.DataFrame,
        exchange: str,
        use_tech_indicators: bool = False,
        tech_indicators: Optional[List[str]] = None,
        use_macro_indicators: bool = False,
        macro_indicators: Optional[List[str]] = None,
        use_covariance: bool = False,
        lookback_period: int = 252,
    ) -> pd.DataFrame:
        """
        Preprocess financial data by cleaning it and adding additional features.
        :param data: DataFrame containing financial data with columns ['date', 'tic', 'open', 'high', 'low', 'close', 'volume'].
        :param exchange: The stock exchange for which the data is being processed (e.g., 'NYSE', 'LSE').
        :param use_tech_indicators: Whether to add technical indicators (Default is False).
        :param tech_indicators: List of technical indicators to add.
        :param use_macro_indicators: Whether to add macroeconomic indicators (Default is False).
        :param macro_indicators: List of macroeconomic indicators to add.
        :param use_covariance: Whether to add covariance features (Default is False).
        :param lookback_period: The lookback period for calculating covariance features (Default is 252).
        :return: Processed DataFrame with additional features.
        """
        df = data.copy()

        df = self.__clean_data(df, exchange)
```

```python

        # Add day of the week column
        df["day"] = df["date"].dt.dayofweek

        if use_tech_indicators and tech_indicators:
            df = self.__add_technical_indicators(df, tech_indicators)

        if use_macro_indicators and macro_indicators:
            df = self.__add_macroeconomic_indicators(df, macro_indicators)

        if use_covariance:
            df = self.__add_covariance_features(df, lookback_period)

        df = self.__rename_columns(df)

        return df

    def __clean_data(self, data: pd.DataFrame, exchange: str) -> pd.DataFrame:
        """
        Clean the financial data by ensuring all trading days are represented
        and filling missing values appropriately.
        :param data: DataFrame containing financial data with columns ['date', 'tic', 'open', 'high', 'low', 'close', 'volume'].
        :param exchange: The stock exchange for which the data is being processed (e.g., 'NYSE', 'LSE').
        :return: Cleaned DataFrame with all trading days represented and missing values filled.
        """
        trading_days = self.__get_trading_dates(exchange)
        tickers = data["tic"].unique()

        # Create a DataFrame with all combinations of trading dates and tickers
        index = list(itertools.product(trading_days, tickers))
        df = pd.DataFrame(index, columns=["date", "tic"]).merge(
            data, on=["date", "tic"], how="left"
        )

        df = df.sort_values(by=["tic", "date"])

        # Fill missing values with forward fill method and then backward fill
        df["open"] = df.groupby("tic")["open"].ffill().bfill()
        df["close"] = df.groupby("tic")["close"].ffill().bfill()
        df["high"] = df.groupby("tic")["high"].ffill().bfill()
        df["low"] = df.groupby("tic")["low"].ffill().bfill()

        # Fill missing volumes with 0 to indicate no trading activity
        df["volume"] = df["volume"].fillna(0)

        return df

    def __get_trading_dates(self, exchange: str) -> List[datetime]:
        """
        Get trading dates for a given exchange within a specified date range.
        :param exchange: The stock exchange for which the trading dates are needed (e.g., 'NYSE', 'LSE').
```

```
        :return: List of trading dates as datetime objects.
        """
        dates = (
            ecals.get_calendar(exchange)
            .sessions_in_range(start=self.start_date, end=self.end_date)
            .to_list()
        )

        # Convert to datetime objects
        return [d.to_pydatetime() for d in dates]

    def __add_technical_indicators(
        self, data: pd.DataFrame, indicators: List[str]
    ) -> pd.DataFrame:
        """
        Add technical indicators to the DataFrame based on the specified indicators.
        :param data: DataFrame containing financial data with columns ['date', 'tic', 'open', 'high', 'low', 'close', 'volume'].
        :param indicators: List of technical indicators to add (e.g., ['macd', 'rsi', 'cci']).
        :return: DataFrame with additional technical indicators.
        :raises Exception: If there is an error processing a specific indicator for a ticker.
        """
        # Convert the dataframe to StockDataFrame
        stock_df = StockDataFrame.retype(data.copy())
        tickers = data["tic"].unique()

        # Iterate over the indicators
        for indicator in indicators:
            indicator_df = pd.DataFrame()

            # Iterate over each ticker
            for ticker in tickers:

                # Extract the indicator data for the ticker
                try:
                    ind_df = stock_df[stock_df["tic"] == ticker][indicator]
                    ind_df = pd.DataFrame(ind_df)
                    ind_df["tic"] = ticker
                    ind_df["date"] = data[data["tic"] == ticker]["date"].values

                    indicator_df = pd.concat(
                        [indicator_df, ind_df], ignore_index=True
                    )
                except Exception as e:
                    print(
                        f"Error processing indicator '{indicator}' for ticker '{ticker}': {e}"
                    )

            data = data.merge(indicator_df, on=["tic", "date"], how="left")

        data.fillna(0, inplace=True)  # Fill NaN values with 0
```

```python

        return data

    def __add_macroeconomic_indicators(
        self, data: pd.DataFrame, indicators: List[str]
    ) -> pd.DataFrame:
        """
        Add macroeconomic indicators to the DataFrame based on the specified indicators.
        :param data: DataFrame containing financial data with columns ['date', 'tic',
        ↪ 'open', 'high', 'low', 'close', 'volume'].
        :param indicators: List of macroeconomic indicators to add (e.g., ['^VIX']).
        :return: DataFrame with additional macroeconomic indicators.
        """

        for indicator in indicators:
            findownloader = FinancialDataDownloader(
                self.start_date, self.end_date
            )
            ind_df = findownloader.download_data([indicator])
            indicator_df = ind_df[["date", "close"]].rename(
                columns={"close": indicator}
            )

            data = data.merge(indicator_df, on=["date"])

        return data

    def __add_covariance_features(
        self, data: pd.DataFrame, lookback_period: int = 252
    ) -> pd.DataFrame:
        """
        Add covariance features to the DataFrame.
        :param data: DataFrame containing financial data.
        :param lookback_period: The lookback period for calculating covariance features.
        :return: DataFrame with additional covariance features.
        """
        covariance_df = pd.DataFrame()

        dates = data.date.unique()
        tics = data.tic.unique()

        for i in range(len(dates)):
            # If the date is smaller than the lookback period,
            # do not calculate the covariance and set it to 0
            if i < lookback_period:
                cov_df = pd.DataFrame(
                    {
                        "date": dates[i],
                        "tic": tics,
                        "covariance": [
                            [0] * len(tics) for _ in range(len(tics))
                        ],
                    }
                )
```

```python
            # Calculate the covariance features for the current lookback period
            else:
                # Get the start and end dates for the lookback period
                start_date = dates[i - lookback_period]
                end_date = dates[i]

                # Retrieve the relevant data for the lookback period
                window = data[
                    (data.date >= start_date) & (data.date < end_date)
                ]
                # Pivot the table to get the price data for the lookback period
                price_lookback = window.pivot_table(
                    index="date", columns="tic", values="close"
                )

                # Compute the covariance matrix
                cov_matrix = price_lookback.cov()
                cov_df = pd.DataFrame(
                    {
                        "date": dates[i],
                        "tic": tics,
                        "covariance": cov_matrix.values.tolist(),
                    }
                )

            # Append to the covariance dataframe
            covariance_df = pd.concat(
                [covariance_df, cov_df], ignore_index=True
            )

        # Merge the data on date and tic
        data = data.merge(covariance_df, on=["date", "tic"])

        return data

    def __rename_columns(self, data: pd.DataFrame) -> pd.DataFrame:
        """
        Rename columns to a consistent format to work with FinRL library.
        :param data: DataFrame containing financial data with columns ['date', 'tic',
        ↪  'open', 'high', 'low', 'close', 'volume'].
        :return: DataFrame with renamed columns.
        """

        # Convert column names
        # 1. Split by dot and take the first part
        # 2. Convert to lowercase
        # 3. Remove non-alphanumeric characters
        data.columns = (
            data.columns.str.split(".")
            .str[0]
            .str.lower()
            .str.replace(r"[^a-z0-9_]", "", regex=True)
        )

```

```python
        return data

    def __set_index(self, data: pd.DataFrame) -> pd.DataFrame:
        """
        Set the index of the DataFrame to a number from 0 to the number of distinct dates in the dataset.
        This is useful for ensuring that the index is consistent with the FinRL library's expectations.
        :param data: DataFrame containing financial data.
        :return: DataFrame with the index set to a number from 0 to the number of distinct dates.
        """

        data.sort_values(by=["date", "tic"], inplace=True)

        # Set the index to a number from 0 to number of distinct dates in the dataset
        data.set_index(data["date"].astype("category").cat.codes, inplace=True)

        return data

    def split_train_test(
        self, data: pd.DataFrame, test_start_date: pd.Timestamp | str
    ) -> tuple[pd.DataFrame, pd.DataFrame]:
        """
        Split the data into training and testing sets based on the given test start date.
        :param data: DataFrame containing financial data.
        :param test_start_date: The start date for the testing set. Data after this date will be used for testing.
        :return: A tuple containing the training DataFrame and the testing DataFrame.
        """

        # Ensure test start date is part of the DataFrame
        test_start_date = pd.to_datetime(test_start_date)
        if "date" not in data.columns:
            raise ValueError("Data must contain a 'date' column for splitting.")
        if (
            test_start_date < data["date"].min()
            or test_start_date > data["date"].max()
        ):
            raise ValueError(
                f"The test start date is outside of the 'date' column values."
            )

        # Split the data into training and testing sets
        train_data = data[data["date"] < test_start_date].copy()
        test_data = data[data["date"] >= test_start_date].copy()

        train_data = self.__set_index(train_data)
        test_data = self.__set_index(test_data)

        return train_data, test_data

    def save_train_test_data(
```

```python
        self,
        train_data: pd.DataFrame,
        test_data: pd.DataFrame,
        directory: str,
        filename: str,
    ) -> None:
        """
        Save the training and testing data to CSV files.
        :param train_data: DataFrame containing training data.
        :param test_data: DataFrame containing testing data.
        :param directory: Directory where the CSV files will be saved.
        :param filename: Base filename for the CSV files (without extension).
        """
        train_file_path = f"{directory}/{filename}_train.csv"
        test_file_path = f"{directory}/{filename}_trade.csv"

        train_data.to_csv(train_file_path, index=False)
        test_data.to_csv(test_file_path, index=False)

        print(f"Train data saved to {train_file_path}")
        print(f"Test data saved to {test_file_path}")

    def __load_file(self, filepath: str) -> pd.DataFrame:
        """
        Load a CSV file into a DataFrame.
        :param filepath: Path to the CSV file.
        :return: DataFrame containing the data from the CSV file.
        :raises FileNotFoundError: If the file does not exist.
        :raises ValueError: If the 'tic' column is missing from the data.
        """
        try:
            data = pd.read_csv(filepath)
            # Convert date columns to datetime format if they exist
            if "date" in data.columns:
                data["date"] = pd.to_datetime(data["date"])

            # Ensure the 'tic' column exists
            if "tic" not in data.columns:
                raise ValueError("The 'tic' column is missing from the data.")

            # Set the index
            data = self.__set_index(data)

            return data

        except FileNotFoundError:
            raise FileNotFoundError(f"No file found at {filepath}")

    def load_train_test_data(
        self, directory: str, filename: str
    ) -> tuple[pd.DataFrame, pd.DataFrame]:
        """
        Load the training and testing data from CSV files.
        :param directory: Directory where the CSV files are located.
        :param filename: Base filename for the CSV files (without extension).
```

```python
        :return: A tuple containing the training DataFrame and the testing DataFrame.
        """

        train_data = self.__load_file(f"{directory}/{filename}_train.csv")
        test_data = self.__load_file(f"{directory}/{filename}_trade.csv")

        return train_data, test_data
```

I.3 Agents Module

Listing 7: drl_agent.py

```python
from typing import Literal, Optional, Tuple

import pandas as pd
from stable_baselines3.common.base_class import BaseAlgorithm
from stable_baselines3.common.logger import configure
from stable_baselines3.common.vec_env import DummyVecEnv
from wandb.integration.sb3 import WandbCallback
from wandb.sdk.wandb_run import Run

from agents.tb_callback import TensorboardCallback
from config import config_models


class DRLAgent:
    def __init__(self, run: Optional[Run] = None):
        """
        Initialises the DRL agent.
        """
        self.run = run

    def get_model(
        self,
        model_name: str,
        environment: DummyVecEnv,
        use_case: Literal["stock-trading", "portfolio-optimisation"],
        directory: Optional[str] = None,
        model_kwargs: Optional[dict] = None,
        policy: str = "MlpPolicy",
        policy_kwargs: Optional[dict] = None,
        seed: Optional[int] = None,
        verbose: int = 1,
    ) -> BaseAlgorithm:
        """
        Returns a DRL model based on the specified model name and parameters.
        :param model_name: The name of the DRL model to create.
        :param environment: The environment in which the model will be trained.
        :param use_case: The use case for the model (e.g., stock trading, portfolio
        ↪  optimization).
        :param directory: The directory where the tensorboard logs will be saved.
```

```python
        :param model_kwargs: Additional keyword arguments for the model.
        :param policy: The policy to use for the model.
        :param policy_kwargs: Additional keyword arguments for the policy.
        :param seed: Random seed for reproducibility.
        :param verbose: Verbosity level for the model.
        :return: An instance of the specified DRL model.
        :raises ValueError: If the model name is not supported.
        """

        if model_name not in config_models.MODELS:
            raise ValueError(
                f"Model {model_name} is not supported. Choose from
                ↪  {config_models.MODELS.keys()}."
            )

        if model_kwargs is None:
            if use_case == "stock-trading":
                model_kwargs = config_models.MODEL_KWARGS_STOCK[model_name]
            elif use_case == "portfolio-optimisation":
                model_kwargs = config_models.MODEL_KWARGS_PORTFOLIO[model_name]

        if verbose:
            print(f"Model arguments: {model_kwargs}")

        if not self.run:
            if not directory:
                raise ValueError(
                    "Directory must be specified if run is not provided."
                )

            tensorboard_log = f"{directory}/{model_name}"

            logger = configure(tensorboard_log, ["csv", "tensorboard"])

            model = config_models.MODELS[model_name](
                policy=policy,
                env=environment,
                verbose=verbose,
                tensorboard_log=tensorboard_log,
                seed=seed,
                policy_kwargs=policy_kwargs,
                **model_kwargs,
            )

            model.set_logger(logger)

        else:
            model = config_models.MODELS[model_name](
                policy=policy,
                env=environment,
                verbose=verbose,
                seed=seed,
                policy_kwargs=policy_kwargs,
                **model_kwargs,
```

```
            )

        return model

    def train(
        self,
        model: BaseAlgorithm,
        tb_log_name: str,
        log_interval: int = 20,
        total_timesteps: int = 100000,
        callback: TensorboardCallback | WandbCallback = TensorboardCallback(),
    ) -> BaseAlgorithm:
        """
        Trains the DRL model with the specified parameters.
        :param model: The DRL model to train.
        :param tb_log_name: The name for the tensorboard log.
        :param log_interval: The interval for logging training progress.
        :param total_timesteps: The total number of timesteps for training.
        :param callback: Callback function for additional logging or actions during
        ↪  training.
        :return: The trained DRL model.
        """

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

        return model

    def predict(
        self,
        model: BaseAlgorithm,
        environment,
        deterministic: bool = True,
    ) -> Tuple[pd.DataFrame, pd.DataFrame]:
        """
        Makes a prediction given the provided model and environment.
        :param model: The trained DRL model.
        :param environment: The environment in which to run the prediction.
        :param deterministic: Whether to use deterministic actions.
        :return: The account memory and actions memory after the prediction.
        """
        test_env, test_obs = environment.get_sb_env()
        account_memory = []
        actions_memory = []

        test_env.reset()
        max_steps = environment.df.index.nunique()

        for i in range(max_steps):
            action, _ = model.predict(test_obs, deterministic=deterministic)
```

```python

            # Last step: Save account and actions memory
            if i == max_steps - 1:
                account_memory = test_env.env_method(
                    method_name="save_asset_memory"
                )
                actions_memory = test_env.env_method(
                    method_name="save_action_memory"
                )

            test_obs, _, dones, _ = test_env.step(action)

            # If the environment is done, break the loop
            if dones[0]:
                break

        return account_memory[0], actions_memory[0]

    def save_model(
        self,
        model: BaseAlgorithm,
        model_name: str,
        directory: str,
    ):
        """
        Saves the trained model to a specified directory.
        :param model: The trained DRL model.
        :param model_name: The name of the model.
        :param directory: The directory where the model will be saved.
        """
        model_path = f"{directory}/{model_name}"
        model.save(model_path)
        print(f"Model saved to {model_path}")

    def load_model(
        self,
        model_name: str,
        directory: str,
    ) -> BaseAlgorithm:
        """
        Loads a trained model from a specified directory.
        :param model_name: The name of the model to load.
        :param directory: The directory where the model is saved.
        :return: The loaded DRL model.
        :raises ValueError: If the model name is not supported.
        """

        if model_name not in config_models.MODELS:
            raise ValueError(
                f"Model {model_name} is not supported. Choose from
                ↪ {config_models.MODELS.keys()}."
            )

        model_path = f"{directory}/{model_name}"
```

```
        model = config_models.MODELS[model_name].load(model_path)

        print(f"Model successfully loaded from {model_path}")

        return model
```

Listing 8: `tb_callback.py`

```
from stable_baselines3.common.callbacks import BaseCallback


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

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

    def _on_step(self) -> bool:
        try:
            self.logger.record(
                key="train/reward", value=self.locals["rewards"][0]
            )
        except KeyError:
            self.logger.record(
                key="train/reward", value=self.locals["reward"][0]
            )
        except Exception as e:
            print(f"Error recording reward: {e}")
        return True
```

I.4 Environments Module

Listing 9: `env_portfolio_optimisation.py`

```
from datetime import datetime
from typing import Dict, List, Literal, Optional, Tuple

import gymnasium as gym
import numpy as np
import pandas as pd
from gymnasium import spaces
from gymnasium.utils import seeding
from stable_baselines3.common.vec_env import DummyVecEnv
from stable_baselines3.common.vec_env.base_vec_env import VecEnvObs

from config import config
from pbenchmark.portfolio_benchmark import PortfolioBenchmark
```

```python


class PortfolioOptimisationEnv(gym.Env):
    def __init__(
        self,
        data: pd.DataFrame,
        stock_dimension: int,
        initial_amount: float,
        reward_scaling: float,
        state_space: int,
        action_space: int,
        state_columns: List[str],
        normalisation_strategy: Literal["sum", "softmax"],
        verbose: int = 10,
        day: int = 0,
        seed: Optional[int] = None,
    ):
        """
        Initialise the portfolio optimisation environment.
        :param data: DataFrame containing stock data.
        :param stock_dimension: Number of stocks in the environment.
        :param initial_amount: Initial cash available for portfolio allocation.
        :param reward_scaling: Scaling factor for the reward.
        :param state_space: Dimension of the state space.
        :param action_space: Dimension of the action space.
        :param state_columns: List of columns in the dataframe to represent the state
        ↪  space.
        :param normalisation_strategy: Strategy (softmax, sum) to normalise the actions
        ↪  to sum to 1.
        :param verbose: Verbosity level for logging.
        :param day: Current day in the trading data.
        :param seed: Random seed for reproducibility.
        """
        self.df = data
        self.day = day
        self.data = self.df.loc[self.day, :]
        self.stock_dimension = stock_dimension
        self.initial_amount = initial_amount
        self.reward_scaling = reward_scaling
        self.state_space = state_space
        self.action_space = spaces.Box(low=0, high=1, shape=(action_space,))
        self.observation_space = spaces.Box(
            low=-np.inf,
            high=np.inf,
            shape=(
                self.state_space
                + (
                    self.stock_dimension - 1
                    if "covariance" in state_columns
                    else 0
                ),
                self.stock_dimension,
            ),
        )
```

```
        self.state_columns = state_columns
        self.terminal = False
        self.verbose = verbose
        self.normalisation_strategy = normalisation_strategy

        # Initialise state
        self.state = self.__get_state()
        self.portfolio_value = self.initial_amount
        self.weights = [1 / self.stock_dimension] * self.stock_dimension

        # Initialise reward
        self.reward = 0.0

        # Initialise counter variables
        self.episode = 0

        # Initialise memory variables
        self.asset_memory = [self.initial_amount]
        self.rewards_memory = [self.reward]
        self.return_memory = [0]
        self.actions_memory = [self.weights]
        self.date_memory = [self.__get_date()]

        # Initialise the benchmark
        self.benchmark = PortfolioBenchmark()

        # Set the random seed for reproducibility
        self._seed(seed)

    def __get_state(self) -> List:
        """
        Get the current state representation from the data.
        :return: Current state as a list of values for each column in state_columns.
        """
        if "covariance" in self.state_columns:
            covariance = np.array(
                self.data["covariance"]
                .apply(lambda x: np.array(eval(x)))  # type: ignore
                .values.tolist()
            )
            return np.append(
                covariance,
                [
                    self.data[col].values.tolist()
                    for col in self.state_columns
                    if col != "covariance"
                ],
                axis=0,
            ).tolist()
        else:
            return [
                self.data[col].values.tolist() for col in self.state_columns
            ]

    def __get_date(self) -> datetime:
```

```python
        """
        Get the current date from the data.
        :return: Current date.
        """

        return self.data.date.unique()[0]

    def step(self, action: np.ndarray) -> Tuple[List, float, bool, bool, Dict]:
        """
        Execute one time step within the environment.
        :param action: List of actions to take for each stock.
        :return: Tuple containing the next state, reward, done flag, truncated flag, and
        ↪  additional info.
        """
        self.terminal = self.day >= len(self.df.index.unique()) - 1

        if self.terminal:
            df_return = self.save_asset_memory()

            # Print information if verbose
            if self.verbose and self.episode % self.verbose == 0:
                print("=================================")
                print(f"day: {self.day}, episode: {self.episode}")
                print(f"begin_total_asset:{self.asset_memory[0]:.2f}")
                print(f"end_total_asset:{self.portfolio_value:.2f}")
                sharpe = self.benchmark.compute_sharpe_ratio(df_return)
                print(f"sharpe_ratio: {sharpe:.2f}")
                print("=================================")

        else:
            # Normalise actions
            weights = self.__normalise_actions(
                action, self.normalisation_strategy  # type: ignore
            )
            self.actions_memory.append(weights.tolist())

            # Retrieve previous day's prices
            prev_prices = np.array(self.data.close.values)

            # Update the state with the new actions
            self.day += 1
            self.data = self.df.loc[self.day, :]
            self.state = self.__get_state()

            # Retrieve current day's prices
            curr_prices = np.array(self.data.close.values)

            # Compute the portfolio returns
            portfolio_return = np.dot(
                ((curr_prices / prev_prices) - 1), weights
            )

            # Update portfolio value
            new_portfolio_value = self.portfolio_value * (1 + portfolio_return)
```

```python

            # Update the reward: Change in portfolio value
            self.reward = new_portfolio_value - self.portfolio_value
            self.portfolio_value = new_portfolio_value
            self.rewards_memory.append(self.reward)
            self.reward *= self.reward_scaling

            # Update the memory
            self.return_memory.append(portfolio_return)
            self.date_memory.append(self.__get_date())
            self.asset_memory.append(new_portfolio_value)

        # The fourth element in the tuple is always False for this environment
        # Corresponds to whether the environment is truncated or not
        return self.state, self.reward, self.terminal, False, {}

    def reset(
        self,
        *,
        seed=None,
        options=None,
    ) -> Tuple[List, Dict]:
        """
        Resets the environment and returns a new state representation.
        :param seed: Random seed for reproducibility.
        :param options: Options for resetting the environment.
        :return: Tuple containing the initial state and an empty dictionary for
        ↪  additional info.
        """
        self.day = 0
        self.data = self.df.loc[self.day, :]
        self.state = self.__get_state()
        self.portfolio_value = self.initial_amount

        self.terminal = False

        self.weights = [1 / self.stock_dimension] * self.stock_dimension
        self.asset_memory = [self.initial_amount]
        self.return_memory = [0]
        self.rewards_memory = [0.0]
        self.actions_memory = [self.weights]
        self.date_memory = [self.__get_date()]

        self.episode += 1

        return self.state, {}

    def render(self, mode: str = "human", close=False) -> List:
        """
        Render the current state of the environment.
        :param mode: The rendering mode (default is "human").
        :param close: Whether to close the rendering window (not used in this
        ↪  environment).
        :return: Current state.
```

```python
        """
        return self.state

    def __normalise_actions(
        self,
        actions: np.ndarray,
        strategy: Literal["sum", "softmax"],
    ) -> np.ndarray:
        """
        Apply normalisation to the actions.
        :param actions: Actions to be normalised.
        :param strategy: Normalisation strategy to use.
        :return: Normalised actions.
        """
        if strategy == "sum":
            total_sum = np.sum(actions)
            if total_sum != 0:
                return actions / total_sum
            else:
                # Explicitly handle zero-sum case by returning uniform distribution
                return np.ones_like(actions) / len(actions)
        else:
            exp_actions = np.exp(actions)
            return exp_actions / np.sum(exp_actions)

    def save_asset_memory(self) -> pd.DataFrame:
        """
        Returns a DataFrame containing the asset memory.
        :return: DataFrame with asset memory.
        """
        df_asset = pd.DataFrame(
            {"date": self.date_memory, "account_value": self.asset_memory}
        )

        # Add daily return
        df_asset["daily_return"] = (
            df_asset["account_value"].pct_change().fillna(0)
        )

        # Add cumulative return
        df_asset["cumulative_return"] = (
            1 + df_asset["daily_return"]
        ).cumprod() - 1

        return df_asset

    def save_action_memory(self) -> pd.DataFrame:
        """
        Returns a DataFrame containing the action memory.
        :return: DataFrame with actions memory.
        """
        df_actions = pd.DataFrame(
            self.actions_memory,
            columns=self.data.tic.values,
        )
```

```
        df_actions["date"] = self.date_memory

        return df_actions

    def _seed(self, seed: Optional[int] = None) -> List[int]:
        """
        Set the random seed for the environment.
        :param seed: Random seed for reproducibility.
        :return: List containing the seed used.
        """
        self.np_random, seed = seeding.np_random(seed)
        return [seed]

    def get_sb_env(self) -> Tuple[DummyVecEnv, VecEnvObs]:
        """
        Get the stable-baselines environment.
        :return: Stable-baselines environment and observation space.
        """
        e = DummyVecEnv([lambda: self])
        obs = e.reset()
        return e, obs


class PortfolioOptimisationEnvWrapper:
    def __init__(
        self,
        train_data: pd.DataFrame,
        trade_data: pd.DataFrame,
        state_columns: List[str] = ["close"],
        initial_amount: float = config.INITIAL_AMOUNT,
        reward_scaling: float = 1e-1,
        normalisation_strategy: Literal["sum", "softmax"] = "softmax",
        verbose: int = 10,
    ):
        """
        Initialises the trading environment.
        :param train_data: DataFrame containing training data.
        :param trade_data: DataFrame containing trading data.
        :param state_columns: List of columns to represent the state space.
        :param initial_amount: Initial cash available for trading.
        :param reward_scaling: Scaling factor for the reward.
        :param normalisation_strategy: Strategy (softmax, sum) to normalise the actions
        ↪  to sum to 1.
        :param verbose: Verbosity level for logging.
        """
        self.stock_dim = train_data.tic.nunique()
        self.state_space = len(state_columns)
        self.train_data = train_data
        self.trade_data = trade_data

        self.env_args = {
            "initial_amount": initial_amount,
            "state_space": self.state_space,
            "stock_dimension": self.stock_dim,
```

```
            "action_space": self.stock_dim,
            "reward_scaling": reward_scaling,
            "state_columns": state_columns,
            "normalisation_strategy": normalisation_strategy,
            "verbose": verbose,
        }

        if verbose:
            print(
                f"Environment successfully created with \n\tStock dimension:
                ↪ {self.stock_dim} \n\tState space: {self.state_space}"
            )

    def get_train_env(self) -> DummyVecEnv:
        """
        Creates and returns the training environment.
        :return: Training environment instance.
        """
        self.train_env = PortfolioOptimisationEnv(
            data=self.train_data, **self.env_args
        )
        return self.train_env.get_sb_env()[0]

    def get_trade_env(
        self,
    ) -> Tuple[PortfolioOptimisationEnv, Tuple[DummyVecEnv, VecEnvObs]]:
        """
        Creates and returns the trading environment.
        :return: Trading environment instance and its stable-baselines environment.
        """
        self.trade_env = PortfolioOptimisationEnv(
            data=self.trade_data, **self.env_args
        )
        return self.trade_env, self.trade_env.get_sb_env()
```

I.5 Benchmarking Module

Listing 10: portfolio_benchmark.py

```
from typing import Dict, Literal

import pandas as pd
from pyfolio import timeseries
from pypfopt import EfficientFrontier, expected_returns, risk_models

from config import config
from preprocessor.findata_downloader import FinancialDataDownloader


class PortfolioBenchmark:
    def __init__(self) -> None:
```

```
        """
        Initialise the PortfolioBenchmark class.
        """
        pass

    def convert_daily_return(
        self,
        df_account: pd.DataFrame,
    ) -> pd.Series:

        df_returns = df_account.copy()

        df_returns["date"] = pd.to_datetime(df_returns["date"])
        df_returns.set_index("date", drop=True, inplace=True)
        df_returns.index = df_returns.index.tz_localize("UTC")  # type: ignore
        return pd.Series(
            df_returns["daily_return"].values, index=df_returns.index
        )

    def set_data(
        self,
        train_data: pd.DataFrame,
        trade_data: pd.DataFrame,
        lookback: int = 252,
    ) -> None:
        """
        Combine the training and testing data into a single DataFrame and
        filter the data for the specified time period.
        :param train_data: DataFrame containing training data.
        :param trade_data: DataFrame containing trading data.
        :param lookback: Lookback period in days for the test start date.
        """
        data = pd.concat([train_data, trade_data], ignore_index=True)

        # Find the test start date or the next available trading date
        test_start = pd.to_datetime(config.TEST_START_DATE)
        if test_start not in data["date"].values:
            print("Test start date is not a trading date in the dataset.")
            next_date = test_start
            for lookahead in range(1, 6):
                next_date = test_start + pd.Timedelta(days=lookahead)
                if next_date in data["date"].values:
                    print(
                        f"Using next available trading date: {next_date.date()}"
                    )
                    test_start = next_date
                    break
            else:
                raise ValueError(
                    "No available trading date found after the test start date."
                )

        data.set_index(data["date"].astype("category").cat.codes, inplace=True)

        # Filter data for the test period plus lookback window
```

```python
        start_idx = (
            data[data["date"] == test_start].index.unique().values[0] - lookback
        )
        end_idx = data.index.max()

        self.data = data.loc[start_idx:end_idx, :]
        self.lookback = lookback

    def optimise_portfolio(
        self,
        strategy: Literal["mean", "min", "momentum", "equal"],
        initial_amount: int = config.INITIAL_AMOUNT,
    ) -> pd.DataFrame:
        """
        Optimises the portfolio depending on the specified strategy:
        - 'mean': Mean-variance optimisation
        - 'min': Minimum variance optimisation
        - 'momentum': Momentum-based strategy
        - 'equal': Equal weight strategy
        :param strategy: Type of optimisation
        :param initial_amount: Initial capital for the portfolio, default is 100000.
        :return: DataFrame with date, account_value, and daily_return columns.
        """
        # Pivot data: date as index, tics as columns
        price_df = self.data.pivot(
            index="date", columns="tic", values="close"
        ).sort_index()

        # Calculate daily returns
        returns_df = price_df.pct_change().dropna()

        # Initialise memory
        weights_record = {}
        portfolio_returns = [0.0]
        portfolio_values = [initial_amount]
        portfolio_value = initial_amount

        dates = returns_df.index

        for i in range(self.lookback + 1, len(returns_df)):
            current_date = dates[i]

            window_prices = price_df.iloc[i - self.lookback : i + 1]

            if strategy == "momentum":
                weights = self.__momentum_strategy(window_prices)
            elif strategy == "equal":
                weights = self.__equal_weight_strategy(
                    price_df.columns.tolist()
                )
            elif strategy == "mean":
                weights = self.__mean_variance_strategy(window_prices)
            elif strategy == "min":
                weights = self.__min_variance_strategy(window_prices)
            else:
```

```
                raise ValueError(
                    "Invalid strategy. Choose from 'mean', 'min', 'momentum', or 'equal'."
                )

            weights_record[current_date] = weights

            # Calculate daily return of the portfolio
            daily_return = sum(
                weights.get(tic, 0) * returns_df.iloc[i][tic]
                for tic in price_df.columns
            )
            portfolio_returns.append(daily_return)

            # Calculate daily portfolio value
            portfolio_value *= 1 + daily_return
            portfolio_values.append(portfolio_value)

        # Convert to pandas DataFrame with date column, account_value and daily_return columns
        portfolio = pd.DataFrame(
            {
                "date": dates[self.lookback :],
                "account_value": portfolio_values,
                "daily_return": portfolio_returns,
            }
        )

        return portfolio

    def __momentum_strategy(
        self, prices: pd.DataFrame, lookback: int = 252
    ) -> Dict:
        """
        Implements a simple momentum strategy.
        :param prices: DataFrame of prices.
        :return: Dictionary of weights based on momentum.
        """

        returns = prices.pct_change(lookback).iloc[-1]

        # Clipping negative returns to zero
        returns = returns.clip(lower=0)

        # If all returns are zero, return equal weights
        if returns.sum() == 0:
            return {tic: 1 / len(prices.columns) for tic in prices.columns}
        # Normalise weights to sum to 1
        weights = returns / returns.sum()
        return weights.to_dict()

    def __equal_weight_strategy(self, ticker_list: list) -> Dict:
        """
        Implements an equal weight strategy.
```

```
        :param ticker_list: List of asset tickers.
        :return: Dictionary of equal weights for each asset.
        """
        num_assets = len(ticker_list)
        return {tic: 1 / num_assets for tic in ticker_list}

    def __mean_variance_strategy(self, prices: pd.DataFrame) -> Dict:
        """
        Implements a mean-variance optimisation strategy.
        :param prices: DataFrame of prices.
        :return: Dictionary of weights based on mean-variance optimisation.
        """
        mu = expected_returns.mean_historical_return(prices)
        S = risk_models.sample_cov(prices)

        ef = EfficientFrontier(mu, S)
        _ = ef.max_sharpe()
        return ef.clean_weights()

    def __min_variance_strategy(self, prices: pd.DataFrame) -> Dict:
        """
        Implements a minimum variance optimisation strategy.
        :param prices: DataFrame of prices.
        :return: Dictionary of weights based on minimum variance optimisation.
        """
        mu = expected_returns.mean_historical_return(prices)
        S = risk_models.sample_cov(prices)

        ef = EfficientFrontier(mu, S)
        _ = ef.min_volatility()
        return ef.clean_weights()

    def get_index_performance(
        self, start_date: str, end_date: str, index_ticker: str
    ):
        findownloader = FinancialDataDownloader(start_date, end_date)
        index_df = findownloader.download_data([index_ticker])
        index_df = index_df[["date", "close"]].copy()

        # Compute daily returns
        index_df["daily_return"] = index_df["close"].pct_change().fillna(0)

        # Compute the cumulative returns
        index_df["cumulative_return"] = (
            1 + index_df["daily_return"]
        ).cumprod() - 1

        # Compute the account value
        index_df["account_value"] = (
            1 + index_df["daily_return"]
        ).cumprod() * config.INITIAL_AMOUNT
        # Set the model name and drop price column
        index_df["model"] = "Index"
        index_df.drop(columns=["close"], inplace=True)

```

```
        return index_df

    def compute_perf_stats(self, df_account: pd.DataFrame) -> pd.Series:
        """
        Computes performance statistics for the portfolio using PyFolio.
        :param df_account: DataFrame containing account values with 'date',
        'account_value', and 'daily_return' columns.
        :return: Series containing performance statistics.
        """
        pf_returns = self.convert_daily_return(df_account)
        perf_stats_alg = timeseries.perf_stats(
            returns=pf_returns,
            factor_returns=pf_returns,
            positions=None,
            transactions=None,
            turnover_denom="AGB",
        )
        return perf_stats_alg

    def compute_sharpe_ratio(self, df_account: pd.DataFrame) -> float:
        """
        Computes the Sharpe ratio for the portfolio.
        :param df_account: DataFrame containing account values with 'date',
            'account_value', and 'daily_return' columns.
        :return: The Sharpe ratio as a float.
        """
        perf_stats = self.compute_perf_stats(df_account)
        return perf_stats.get("Sharpe ratio", 0)

    def compute_cum_returns(self, df_account: pd.DataFrame) -> float:
        """
        Computes the cumulative returns for the portfolio.
        :param df_account: DataFrame containing account values with 'date',
            'account_value', and 'daily_return' columns.
        :return: The cumulative returns as a float.
        """
        perf_stats = self.compute_perf_stats(df_account)
        return perf_stats.get("Cumulative returns", 0)
```

I.6 Explainability Module

Listing 11: explainability.py

```
from typing import Tuple

import pandas as pd
from sklearn.ensemble import RandomForestRegressor

# https://scikit-learn.org/stable/modules/generated/sklearn.experimental.enable_halving
↪ _search_cv.html
from sklearn.experimental import enable_halving_search_cv
```

```python
from sklearn.model_selection import HalvingGridSearchCV  # type: ignore
from sklearn.model_selection import train_test_split


class Explainer:
    """
    Base class for explainers in the portfolio optimization environment.
    This class provides methods to build state and action spaces, split data,
    and build a proxy explanation model.
    """

    def __init__(self):
        self.state_space = None
        self.action_space = None

    def build_state_space(
        self, data: pd.DataFrame, columns: list
    ) -> pd.DataFrame:
        """
        Build the state space dataframe from a given DataFrame.
        :param data: DataFrame containing trade data with 'date' and 'tic' columns.
        :param columns: List of columns to include in the state space.
        :return: DataFrame representing the state space columns.
        """
        # Filter the trade data to include only the relevant columns
        pivot_df = data.pivot(index="date", columns="tic", values=columns)
        # Flatten the multi-index columns
        pivot_df.columns = [
            "_".join(col).strip() for col in pivot_df.columns.values
        ]

        state_space = pivot_df.reset_index()

        # Drop the 'date' column
        state_space.drop(columns=["date"], inplace=True)

        return state_space

    def build_action_space(self, data: pd.DataFrame) -> pd.DataFrame:
        """
        Build the action space dataframe from a given DataFrame.
        :param data: DataFrame containing trade data with 'date' and 'tic' columns.
        :return: DataFrame representing the action space columns.
        """
        # Filter the trade data to include only the relevant columns
        action_space = data.drop(columns=["date"])

        return action_space

    def split_data(
        self,
        state_space: pd.DataFrame,
        action_space: pd.DataFrame,
        test_size: float = 0.2,
    ) -> Tuple[pd.DataFrame, pd.DataFrame, pd.DataFrame, pd.DataFrame]:
```

```python
        """
        Split the state and action space data into training and testing sets.
        :param state_space: DataFrame representing the state space.
        :param action_space: DataFrame representing the action space.
        :param test_size: Proportion of the dataset to include in the test split.
        :return: Tuple containing training and testing sets for state and action spaces.
        """
        X_train, X_test, y_train, y_test = train_test_split(
            state_space, action_space, test_size=test_size, shuffle=False
        )
        return X_train, X_test, y_train, y_test

    def build_proxy_explanation_model(
        self,
        X_train: pd.DataFrame,
        y_train: pd.DataFrame,
        find_best_params: bool = True,
    ) -> RandomForestRegressor:
        """
        Build and train a RandomForestRegressor model to explain the actions
        taken by the agent based on the state space.
        :param X_train: DataFrame containing the training state space data.
        :param y_train: DataFrame containing the training action space data.
        :param find_best_params: Whether to perform hyperparameter tuning.
        :return: The trained RandomForestRegressor model.
        """

        if find_best_params:

            param_grid = {
                "n_estimators": [100, 200, 300],
                "max_depth": [10, 20, 30],
                "min_samples_split": [2, 5, 10],
                "min_samples_leaf": [1, 2, 4],
            }

            grid_search = HalvingGridSearchCV(
                estimator=RandomForestRegressor(),
                param_grid=param_grid,
                cv=3,
            ).fit(X_train, y_train)
            print("Best parameters found: ", grid_search.best_params_)
            print(f"Best score: {grid_search.best_score_:.4f}")

            best_model = grid_search.best_estimator_
        else:
            best_model = RandomForestRegressor(
                n_estimators=200,
                max_depth=10,
                min_samples_split=5,
                min_samples_leaf=4,
            )
            best_model.fit(X_train, y_train)

```

```
        return best_model
```

Listing 12: feature_importance.py

```
from typing import List, Literal

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
from sklearn.ensemble import RandomForestRegressor


class FeatureImportance:
    def __init__(
        self,
        model: RandomForestRegressor,
        columns: List[str],
        directory: str,
        model_name: str,
    ):
        """
        Initialize the FeatureImportance class.
        :param model: Trained Random Forest model.
        :param columns: List of feature names.
        :param directory: Directory where plots will be saved.
        :param model_name: Name of the DRL model to which the feature importances belong
         to.
        """
        self.model = model
        self.columns = columns
        self.directory = directory
        self.model_name = model_name

    def get_feature_importances(self) -> pd.DataFrame:
        """
        Calculate feature importances from the trained model.
        :return: DataFrame containing feature names and their importances.
        """
        importances = self.model.feature_importances_
        feature_importances = pd.DataFrame(
            {
                "feature": self.columns,
                "importance": importances,
            }
        ).sort_values(by="importance", ascending=False)

        feature_importances.reset_index(drop=True, inplace=True)
        feature_importances["ticker"] = feature_importances["feature"].apply(
            lambda x: x.split("_")[-1] if "_" in x else ""
        )
        feature_importances["indicator"] = (
            feature_importances["feature"]
```

```python
            .apply(lambda x: x.split("_")[:-1] if "_" in x else x)
            .apply(lambda x: "_".join(x) if len(x) > 1 else x[0])
        )
        self.feature_importances = feature_importances

        return feature_importances

    def plot_feature_importances(
        self,
        max_features: int = 20,
    ) -> None:
        """
        Plot the feature importances
        :param max_features: Maximum number of features to display. Defaults to 20.
        :return: None
        """
        plt.figure(figsize=(10, 6))
        sns.barplot(
            data=self.feature_importances[:max_features],
            y="feature",
            x="importance",
            orient="h",
        )
        plt.title("Feature importances")
        plt.xlabel("Importance")
        plt.ylabel("Feature")
        plt.tight_layout()
        plt.savefig(
            f"{self.directory}/{self.model_name}_feature_importance.png"
        )
        plt.show()

    def plot_feature_importances_by_ticker(
        self,
        max_features: int = 20,
    ) -> None:
        """
        Plot the feature importances by ticker.
        :return: None
        """
        plt.figure(figsize=(10, 6))
        sns.barplot(
            data=self.feature_importances[:max_features],
            y="feature",
            x="importance",
            orient="h",
            hue="ticker",
        )
        plt.title("Feature importances by Ticker")
        plt.xlabel("Importance")
        plt.ylabel("Feature")
        plt.legend(title="Ticker")
        plt.tight_layout()
        plt.savefig(
            f"{self.directory}/{self.model_name}_feature_importance_by_ticker.png"
```

```
        )
        plt.show()

    def add_comparison(
        self,
        statistic: Literal["mean", "median", "q1", "q3"] = "mean",
    ) -> pd.DataFrame:
        """
        Add a column to the feature importances DataFrame indicating whether
        the importance is above or below the provided statistical function.
        :param statistic: Statistical function to compare against (e.g., mean, median, q1, q3).
        :return: Updated DataFrame with a new column indicating comparison.
        """
        if statistic == "mean":
            threshold = self.feature_importances["importance"].mean()
        elif statistic == "median":
            threshold = self.feature_importances["importance"].median()
        elif statistic == "q1":
            threshold = self.feature_importances["importance"].quantile(0.25)
        elif statistic == "q3":
            threshold = self.feature_importances["importance"].quantile(0.75)
        else:
            raise ValueError(
                "Invalid statistic. Use 'mean', 'median', 'q1', or 'q3'."
            )

        self.feature_importances[statistic] = self.feature_importances[
            "importance"
        ].apply(
            lambda x: (
                f"Above {statistic}" if x > threshold else f"Below {statistic}"
            )
        )

        return self.feature_importances

    def plot_feature_importance_comparison(
        self,
        max_features: int = 20,
        statistic: Literal["mean", "median", "q1", "q3"] = "mean",
    ) -> None:
        """
        Plot the feature importances with comparison to a statistical threshold.
        :param max_features: Maximum number of features to display. Defaults to 20.
        :param statistic: Statistical function to compare against (e.g., mean, median, q1, q3).
        :return: None
        """
        feature_importances = self.add_comparison(statistic)

        plt.figure(figsize=(10, 6))
        sns.barplot(
            data=feature_importances[:max_features],
```

```
            y="feature",
            x="importance",
            orient="h",
            hue=statistic,
        )
        plt.title(f"Feature importances with {statistic} comparison")
        plt.xlabel("Importance")
        plt.ylabel("Feature")
        plt.legend(title="")
        plt.tight_layout()
        plt.savefig(
            f"{self.directory}/{self.model_name}_feature_importance_{statistic}_compari
            ↪  son.png"
        )
        plt.show()

    def plot_feature_importance_by_indicator(
        self,
        max_features: int = 20,
    ) -> None:
        """
        Plot the feature importances by indicator.
        :param max_features: Maximum number of features to display. Defaults to 20.
        :return: None
        """
        plt.figure(figsize=(10, 6))
        sns.barplot(
            data=self.feature_importances[:max_features],
            y="feature",
            x="importance",
            orient="h",
            hue="indicator",
        )
        plt.title("Feature importances by Indicator")
        plt.xlabel("Importance")
        plt.ylabel("Feature")
        plt.legend(title="Indicator")
        plt.tight_layout()
        plt.savefig(
            f"{self.directory}/{self.model_name}_feature_importance_by_indicator.png"
        )
        plt.show()

    def plot_top_features_per_ticker(
        self,
        top_features: int = 5,
    ) -> None:
        """
        Plot the top features per ticker.
        :param directory: Directory where the plot will be saved.
        :param filename: Base filename for the saved plot.
        :param top_features: Number of top features to display per ticker. Defaults to 5
        :return: None
        """
```

```

        # Group by ticker and get the top features by importance
        grouped_features = (
            self.feature_importances.groupby("ticker")[
                ["ticker", "feature", "importance"]
            ]
            .apply(lambda x: x.nlargest(top_features, "importance"))
            .reset_index(drop=True)
        )

        plt.figure(figsize=(10, 6))
        sns.barplot(
            data=grouped_features,
            x="importance",
            y="feature",
            hue="ticker",
        )
        plt.title("Top features per Ticker")
        plt.xlabel("Importance")
        plt.ylabel("Feature")
        plt.legend(title="Ticker")
        plt.tight_layout()
        plt.savefig(
            f"{self.directory}/{self.model_name}_top_features_per_ticker.png"
        )
        plt.show()

    def plot_top_features_per_indicator(
        self,
        top_features: int = 5,
    ) -> None:
        """
        Plot the top features per indicator.
        :param top_features: Number of top features to display per indicator. Defaults
        ↪  to 5
        :return: None
        """

        # Select the top indicators based on their total importance
        top_indicators = (
            self.feature_importances.groupby("indicator")["importance"]
            .sum()
            .nlargest(top_features)
            .index
        )

        # Filter the feature importances to include only the top indicators
        filtered_features = self.feature_importances[
            self.feature_importances["indicator"].isin(top_indicators)
        ].copy()

        # Sort values by indicator column given the order of top_indicators
        filtered_features["indicator"] = pd.Categorical(
            filtered_features["indicator"],
```

```python
            categories=top_indicators,
            ordered=True,
        )

        # Sort the filtered features by indicator and importance
        filtered_features = filtered_features.sort_values(
            by=["indicator", "importance"], ascending=[True, False]
        )

        plt.figure(figsize=(10, 6))
        sns.barplot(
            data=filtered_features,
            x="importance",
            y="feature",
            hue="indicator",
        )
        plt.title("Top features per Indicator")
        plt.xlabel("Importance")
        plt.ylabel("Feature")
        plt.legend(title="Indicator")
        plt.tight_layout()
        plt.savefig(
            f"{self.directory}/{self.model_name}_top_features_per_indicator.png"
        )
        plt.show()

    def plot_mean_importance_by_ticker(
        self,
    ) -> None:
        """
        Plot the mean feature importances by ticker.
        :return: None
        """
        mean_importances = (
            self.feature_importances.groupby("ticker")["importance"]
            .mean()
            .reset_index()
            .sort_values(by="importance", ascending=False)
        )

        plt.figure(figsize=(8, 4))
        sns.barplot(
            data=mean_importances,
            x="importance",
            y="ticker",
            orient="h",
            hue="ticker",
        )
        plt.title("Mean Feature Importances by Ticker")
        plt.xlabel("Mean Importance")
        plt.ylabel("Ticker")
        plt.tight_layout()
        plt.savefig(
            f"{self.directory}/{self.model_name}_mean_importance_by_ticker.png"
        )
```

```python
        plt.show()

    def plot_mean_importance_by_indicator(
        self,
    ) -> None:
        """
        Plot the mean feature importances by indicator.
        :return: None
        """
        mean_importances = (
            self.feature_importances.groupby("indicator")["importance"]
            .mean()
            .reset_index()
            .sort_values(by="importance", ascending=False)
        )

        plt.figure(figsize=(8, 4))
        sns.barplot(
            data=mean_importances,
            x="importance",
            y="indicator",
            orient="h",
            hue="indicator",
        )
        plt.title("Mean Feature Importances by Indicator")
        plt.xlabel("Mean Importance")
        plt.ylabel("Indicator")
        plt.tight_layout()
        plt.savefig(
            f"{self.directory}/{self.model_name}_mean_importance_by_indicator.png"
        )
        plt.show()
```

Listing 13: `lime_explainability.py`

```python
from typing import Callable, List

import pandas as pd
from lime.lime_tabular import LimeTabularExplainer


class LimeExplainer:
    def __init__(self, directory: str, model_name: str) -> None:
        """
        Initialize the LimeExplainer with a directory to save explanation plots.
        :param directory: Directory where plots will be saved.
        :param model_name: Name of the DRL model to which the feature importances belong
        ↪ to.
        """
        self.directory = directory
        self.model_name = model_name

```

```python
    def build_lime_explainer(
        self,
        X_train: pd.DataFrame,
    ) -> LimeTabularExplainer:
        """
        Build a LIME explainer for the given training data.
        :param X_train: DataFrame containing the training data.
        :return: LIME explainer object.
        """
        explainer = LimeTabularExplainer(
            X_train.values, mode="regression", feature_names=X_train.columns
        )
        self.explainer = explainer
        return explainer

    def explain_instance(
        self,
        instance: pd.Series,
        predict_fn: Callable,
    ) -> List:
        """
        Explain a single instance using the LIME explainer.
        Output the explanation in a notebook and return it as a list.
        :param instance: Series representing the instance to explain.
        :param predict_fn: Function to predict actions based on states.
        :return: List containing the explanation.
        """
        explanation = self.explainer.explain_instance(
            instance.values, predict_fn
        )
        explanation.show_in_notebook(show_table=True, show_all=False)
        return explanation.as_list()

    def explain_portfolio(
        self,
        instance: pd.Series,
        columns: List[str],
        predict_fn: Callable,
        filename: str,
    ) -> None:
        """
        Explain the portfolio using the LIME explainer and save the explanations as HTML files.
        :param explainer: LIME explainer object.
        :param instance: Series representing the instance to explain.
        :param columns: List of asset names to explain.
        :param predict_fn: Function to predict actions based on states.
        :param filename: Base filename for the HTML files.
        """
        # Explain each output separately
        for index, column in enumerate(columns):
            print(f"Explaining output for asset: {column}")

            # Define a predict function for the specific asset
```

```
            def predict(x):
                return predict_fn(x)[:, index]

            # Explain the instance for the specific output
            exp = self.explainer.explain_instance(
                instance.values, predict, num_features=10
            )

            # Save the explanation as HTML
            html = exp.as_html()
            with open(
                f"{self.directory}/{self.model_name}_{filename}_lime_single_obs_{column
                ↪   }.html",
                "w",
            ) as f:
                f.write(html)

            exp.show_in_notebook(show_table=True)
```

Listing 14: `shap_explainability.py`

```
from typing import Callable

import numpy as np
import pandas as pd
import shap
from sklearn.ensemble import RandomForestRegressor


class ShapExplainer:
    def __init__(self):
        """
        Initialize the SHAP explainer for the portfolio optimization environment.
        """
        pass

    def build_proxy_explainer(
        self, model: RandomForestRegressor
    ) -> shap.TreeExplainer:
        """
        Build a proxy SHAP explainer for the given model and training data.
        :param model: The trained RandomForestRegressor model.
        :param X_train: DataFrame containing the training data.
        :return: SHAP explainer object.
        """
        explainer = shap.TreeExplainer(model)
        return explainer

    def build_kernel_explainer(
        self, predict_fn: Callable, X_train: pd.DataFrame
    ) -> shap.KernelExplainer:
        """
```

```
        Build a Kernel SHAP explainer for the given prediction function and training
        ↪ data.
        :param predict_fn: Function to predict actions based on states.
        :param X_train: DataFrame containing the training data.
        :return: SHAP KernelExplainer object.
        """
        explainer = shap.KernelExplainer(predict_fn, X_train)
        return explainer

    def compute_shap_values(
        self, explainer: shap.Explainer, X_test: pd.DataFrame
    ) -> shap.Explanation:
        """
        Compute SHAP values for the test data using the given explainer.
        :param explainer: SHAP explainer object.
        :param X_test: DataFrame containing the test data.
        :return: SHAP explanation object containing the SHAP values.
        """
        shap_values = explainer(X_test)
        return shap_values

    def compute_shap_interaction_values(
        self, explainer: shap.Explainer, X_test: pd.DataFrame
    ) -> np.ndarray:
        """
        Compute SHAP interaction values for the test data using the given explainer.
        :param explainer: SHAP explainer object.
        :param X_test: DataFrame containing the test data.
        :return: SHAP explanation object containing the SHAP interaction values.
        """
        shap_interaction_values = explainer.shap_interaction_values(X_test)  # type:
        ↪ ignore
        return shap_interaction_values
```

I.7 Visualisation Module

Listing 15: `__init__.py`

```
import os

import matplotlib.pyplot as plt

# Use custom matplotlib style
style = os.path.dirname(os.path.abspath(__file__)) + "/style.mplstyle"
plt.style.use(style)
```

Listing 16: `benchmark_visualiser.py`

```
import matplotlib.pyplot as plt
```

```python
import pandas as pd
import seaborn as sns


class BenchmarkVisualiser:
    def __init__(
        self,
        directory: str,
    ):
        """
        Initializes the BenchmarkVisualiser with a directory to save plots.
        :param directory: Directory where the plots will be saved.
        """
        self.directory = directory

    def compare_account_value(
        self,
        data: pd.DataFrame,
    ) -> None:
        """
        Visualises the account value over time for different models.
        :param data: DataFrame containing account values with 'date', 'model', and 'account_value' columns.
        """
        plt.figure(figsize=(12, 6))
        sns.lineplot(data=data, x="date", y="account_value", hue="model")
        plt.title("Portfolio Value over Trading Period")
        plt.xlabel("Date")
        plt.ylabel("Account Value")
        plt.legend(title="Models")
        plt.tight_layout()
        plt.savefig(f"{self.directory}/account_value.png")
        plt.show()

    def compare_daily_returns(self, data: pd.DataFrame) -> None:
        """
        Visualises the daily returns over time for different models.
        :param data: DataFrame containing returns with 'date', 'model', and 'daily_return' columns.
        """
        plt.figure(figsize=(12, 6))
        sns.lineplot(data=data, x="date", y="daily_return", hue="model")
        plt.title("Daily Returns over Trading Period")
        plt.xlabel("Date")
        plt.ylabel("Daily Returns")
        plt.legend(title="Models")
        plt.tight_layout()
        plt.savefig(f"{self.directory}/daily_returns.png")
        plt.show()

    def compare_cum_returns(self, data: pd.DataFrame) -> None:
        """
        Visualises the cumulative returns over time for different models.
        :param data: DataFrame containing returns with 'date', 'model', and 'cumulative_return' columns.
```

```python
        """
        plt.figure(figsize=(12, 6))
        sns.lineplot(data=data, x="date", y="cumulative_return", hue="model")
        plt.title("Cumulative Returns over Trading Period")
        plt.xlabel("Date")
        plt.ylabel("Cumulative Returns")
        plt.legend(title="Models")
        plt.tight_layout()
        plt.savefig(f"{self.directory}/cumulative_returns.png")
        plt.show()
```

Listing 17: findata_visualiser.py

```python
from typing import Optional

import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns


class FinancialDataVisualiser:
    def __init__(self, directory: str) -> None:
        """
        Initialise the FinancialDataVisualiser.
        """
        self.directory = directory

    def plot_close_prices(
        self, data: pd.DataFrame, filename: Optional[str] = "close_prices"
    ) -> None:
        """
        Plot closing prices of tickers in the data.
        :param data: DataFrame containing financial data with 'date', 'tic', and 'close'
        ↪ columns.
        """
        # Sample the first 10 tickers if there are more than 10 unique tickers
        if data["tic"].nunique() > 10:
            sample_tickers = data["tic"].unique()[:10]
            data = data[data["tic"].isin(sample_tickers)].copy()

        # Sort the data by 'tic' to ensure consistent plotting
        data.sort_values(by="tic", inplace=True)

        plt.figure(figsize=(12, 5))
        sns.lineplot(data=data, x="date", y="close", hue="tic")
        plt.title("Closing Prices of Tickers")
        plt.xlabel("Date")
        plt.ylabel("Closing Price")
        plt.legend(title="Tickers")
        plt.tight_layout()
        plt.savefig(f"{self.directory}/{filename}.png")
        plt.show()
```

```python

    def plot_technical_indicators(
        self,
        data: pd.DataFrame,
        indicators: dict[str, str],
    ) -> None:
        """
        Plot technical indicators for each ticker in the data.
        :param data: DataFrame containing financial data with 'date', 'tic', and
        ↪  technical indicators.
        :param indicators: Dictionary mapping technical indicator names to their
        ↪  descriptions.
        """

        # Sample the first 10 tickers if there are more than 10 unique tickers
        if data["tic"].nunique() > 10:
            sample_tickers = data["tic"].unique()[:10]
            data = data[data["tic"].isin(sample_tickers)].copy()

        # Sort the data by 'tic' to ensure consistent plotting
        data.sort_values(by="tic", inplace=True)

        # Sample the first 5 indicators if there are more than 5 unique indicators
        ind_size = n if (n := len(indicators)) < 5 else 5

        if ind_size == 1:
            plt.figure(figsize=(12, 5))
            indicator, name = list(indicators.items())[0]
            if indicator in data.columns:
                sns.lineplot(data=data, x="date", y=indicator, hue="tic")
                plt.title(name)
                plt.xlabel("Date")
                plt.ylabel(indicator)
                plt.legend(title="Tickers")
                plt.savefig(f"{self.directory}/technical_indicators.png")
                plt.show()
            else:
                print(f"Technical indicator '{indicator}' not found in data.")
        else:
            _, ax = plt.subplots(
                ind_size, 1, figsize=(12, 5 * ind_size), sharex=True
            )

            # Iterate over indicators to ind_size
            for i, (indicator, name) in enumerate(
                list(indicators.items())[:ind_size]
            ):
                if indicator in data.columns:
                    sns.lineplot(
                        data=data, x="date", y=indicator, hue="tic", ax=ax[i]
                    )
                    ax[i].set_title(name)
                    ax[i].set_xlabel("Date")
                    ax[i].set_ylabel(indicator)
```

```python
                    ax[i].tick_params(labelbottom=True)
                    ax[i].legend(title="Tickers")
                else:
                    print(
                        f"Technical indicator '{indicator}' not found in data."
                    )

        plt.tight_layout()
        plt.savefig(f"{self.directory}/technical_indicators.png")
        plt.show()

    def plot_macroeconomic_indicators(
        self,
        data: pd.DataFrame,
        indicators: dict[str, str],
    ) -> None:
        """
        Plot macroeconomic indicators.
        :param data: DataFrame containing financial data with 'date' and macroeconomic
        ↪ indicators.
        :param indicators: Dictionary mapping macroeconomic indicator names to their
        ↪ descriptions.
        """

        # Sample the first 10 tickers if there are more than 10 unique tickers
        if data["tic"].nunique() > 10:
            sample_tickers = data["tic"].unique()[:10]
            data = data[data["tic"].isin(sample_tickers)].copy()

        # Sort the data by 'tic' to ensure consistent plotting
        data.sort_values(by="tic", inplace=True)

        # Sample the first 5 indicators if there are more than 5 unique indicators
        ind_size = n if (n := len(indicators)) < 5 else 5

        if ind_size == 1:
            plt.figure(figsize=(12, 5))
            indicator, name = list(indicators.items())[0]
            # Convert indicator to alphanumeric
            indicator = "".join(
                ch
                for ch in indicator.split(".", 1)[0].lower()
                if ch.isalnum() or ch == "_"
            )
            if indicator in data.columns:
                # Take the date and the indicator column
                ind_df = data[["date", indicator]]
                # Remove duplicate dates
                ind_df = ind_df.drop_duplicates(subset="date")
                sns.lineplot(data=ind_df, x="date", y=indicator)
                plt.title(name)
                plt.xlabel("Date")
                plt.ylabel(indicator)
                plt.savefig(f"{self.directory}/macroeconomic_indicators.png")
```

```
                    plt.show()
                else:
                    print(
                        f"Macroeconomic indicator '{indicator}' not found in data."
                    )
            else:
                _, ax = plt.subplots(
                    ind_size, 1, figsize=(12, 5 * ind_size), sharex=True
                )

                colors = sns.color_palette().as_hex()[
                    :ind_size
                ]  # Use distinct colors

                # Iterate over indicators to ind_size
                for i, (indicator, name) in enumerate(
                    list(indicators.items())[:ind_size]
                ):
                    # Convert indicator to alphanumeric
                    indicator = "".join(
                        ch
                        for ch in indicator.split(".", 1)[0].lower()
                        if ch.isalnum() or ch == "_"
                    )
                    if indicator in data.columns:
                        # Take the date and the indicator column
                        ind_df = data[["date", indicator]]
                        # Remove duplicate dates
                        ind_df = ind_df.drop_duplicates(subset="date")
                        sns.lineplot(
                            data=ind_df,
                            x="date",
                            y=indicator,
                            ax=ax[i],
                            color=colors[i],
                        )
                        ax[i].set_title(name)
                        ax[i].set_xlabel("Date")
                        ax[i].set_ylabel(indicator)
                        ax[i].tick_params(labelbottom=True)
                    else:
                        print(
                            f"Macroeconomic indicator '{indicator}' not found in data."
                        )

                plt.tight_layout()
                plt.savefig(f"{self.directory}/macroeconomic_indicators.png")
                plt.show()

        def plot_train_test_close_prices(
            self,
            train_data: pd.DataFrame,
            test_data: pd.DataFrame,
        ) -> None:
            """
```

```
        Plot closing prices for train and test datasets.
        :param train_data: DataFrame containing training data with 'date', 'tic', and 'close' columns.
        :param test_data: DataFrame containing testing data with 'date', 'tic', and 'close' columns.
        :param directory: Directory where the plot will be saved.
        :param filename: Name of the file to save the plot (without extension).
        """

        # Sample the first 10 tickers if there are more than 10 unique tickers
        if train_data["tic"].nunique() > 10:
            sample_tickers = train_data["tic"].unique()[:10]
            train_data = train_data[
                train_data["tic"].isin(sample_tickers)
            ].copy()
            test_data = test_data[test_data["tic"].isin(sample_tickers)].copy()

        # Sort the data by 'tic' to ensure consistent plotting
        train_data.sort_values(by="tic", inplace=True)
        test_data.sort_values(by="tic", inplace=True)

        _, ax = plt.subplots(2, 1, figsize=(12, 10), sharex=True, sharey=True)

        sns.lineplot(data=train_data, x="date", y="close", hue="tic", ax=ax[0])
        ax[0].set_title("Train data set")
        ax[0].set_xlabel("Date")
        ax[0].set_ylabel("Closing Price")
        ax[0].tick_params(labelbottom=True)
        ax[0].legend(title="Tickers")

        sns.lineplot(data=test_data, x="date", y="close", hue="tic", ax=ax[1])
        ax[1].set_title("Test data set")
        ax[1].set_xlabel("Date")
        ax[1].set_ylabel("Closing Price")
        ax[1].legend(title="Tickers")

        plt.suptitle("Train and Test Closing Prices of Tickers")
        plt.tight_layout()
        plt.savefig(f"{self.directory}/train_test_close_prices.png")
        plt.show()

    def plot_train_val_test_close_prices(
        self,
        train_data: pd.DataFrame,
        val_data: pd.DataFrame,
        test_data: pd.DataFrame,
    ) -> None:
        """
        Plot closing prices for train, validation, and test datasets.
        :param train_data: DataFrame containing training data with 'date', 'tic', and 'close' columns.
        :param val_data: DataFrame containing validation data with 'date', 'tic', and 'close' columns.
        :param test_data: DataFrame containing testing data with 'date', 'tic', and 'close' columns.
```

```
        """

        # Concatenate dataframes
        data = pd.concat([train_data, val_data, test_data])

        if data.tic.nunique() > 10:
            sample_tickers = data["tic"].unique()[:10]
            data = data[data["tic"].isin(sample_tickers)].copy()

        # Sort the data by 'tic' to ensure consistent plotting
        data.sort_values(by="tic", inplace=True)

        plt.figure(figsize=(12, 5))
        sns.lineplot(data=data, x="date", y="close", hue="tic")

        # Add vertical lines to indicate the split points
        plt.axvline(
            x=val_data["date"].min(),
            color="black",
            linestyle="--",
        )
        plt.axvline(x=test_data["date"].min(), color="black", linestyle="--")

        # Add labels for the split points
        plt.text(
            val_data["date"].min() + pd.Timedelta(days=20),
            data["close"].max(),
            "Validation Start",
            horizontalalignment="left",
            verticalalignment="bottom",
            color="black",
        )
        plt.text(
            test_data["date"].min() + pd.Timedelta(days=20),
            data["close"].max(),
            "Test Start",
            horizontalalignment="left",
            verticalalignment="bottom",
            color="black",
        )

        plt.title("Train, Validation, and Test Closing Prices of Tickers")
        plt.xlabel("Date")
        plt.ylabel("Closing Price")
        plt.legend(title="Tickers")
        plt.tight_layout()
        plt.savefig(f"{self.directory}/train_val_test_close_prices.png")
        plt.show()
```

Listing 18: model_visualiser.py

```
from typing import List

```

```python
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns


class ModelVisualiser:
    def __init__(
        self,
        directory: str,
    ):
        """
        Initializes the ModelVisualiser with a directory to save plots.
        :param directory: Directory where the plots will be saved.
        """
        self.directory = directory

    def evaluate_training(
        self,
        model_name: str,
        x: str,
        y: List[str],
        title: List[str],
        logs_dir: str,
    ) -> None:
        """
        Visualises the training progress of an agent by plotting specified variables against a common
        x-axis variable.
        :param model_name: Name of the model being evaluated (e.g., 'a2c').
        :param x: The name of the x-axis variable (e.g., 'step').
        :param y: List of variable names to be plotted on the y-axis.
        :param title: List of titles for each subplot corresponding to the y variables.
        :param logs_dir: Directory where the training logs are stored.
        """

        num_variables = len(y)
        _, ax = plt.subplots(
            num_variables, 1, figsize=(12, 5 * num_variables), sharex=True
        )

        data = pd.read_csv(f"{logs_dir}/{model_name}/progress.csv")

        colors = sns.color_palette("husl", num_variables)

        # Iterate over the variables to plot
        for i, variable in enumerate(y):
            if variable in data.columns:
                sns.lineplot(
                    data=data, x=x, y=variable, ax=ax[i], color=colors[i]
                )
                ax[i].set_title(title[i])
                ax[i].set_xlabel(x.split("/")[-1].capitalize())
                ax[i].set_ylabel(
                    " ".join(variable.split("/")[-1].split("_")).capitalize()
```

```python
                )
                ax[i].tick_params(labelbottom=True)

        plt.suptitle(f"Training Progress of {model_name.upper()} Agent", y=1)
        plt.tight_layout()
        plt.savefig(f"{self.directory}/{model_name}_train_evaluation.png")
        plt.show()

    def evaluate_testing(
        self,
        model_name: str,
        account_data: pd.DataFrame,
        actions_data: pd.DataFrame,
    ) -> None:
        """
        Visualises the testing results of an agent by plotting account value and actions
        ↪ over time.
        :param model_name: Name of the model being evaluated (e.g., 'a2c').
        :param account_data: DataFrame containing account values with a 'date' column.
        :param actions_data: DataFrame containing actions with a 'date' column.
        """

        _, ax = plt.subplots(2, 1, figsize=(12, 10), sharex=True)

        # Plot account value
        sns.lineplot(
            data=account_data,
            x="date",
            y="account_value",
            ax=ax[0],
        )
        ax[0].set_title("Account Value Over Time")
        ax[0].set_xlabel("Date")
        ax[0].set_ylabel("Account Value")
        ax[0].tick_params(labelbottom=True)

        # Format the actions_data
        if "date" in actions_data.columns:
            actions_data = actions_data.reset_index(drop=True).melt(
                id_vars="date", var_name="tic", value_name="action"
            )
        else:
            actions_data = actions_data.reset_index().melt(
                id_vars="date", var_name="tic", value_name="action"
            )
        actions_data.sort_values(by=["date", "tic"]).reset_index(
            drop=True, inplace=True
        )

        # Sample the first 10 tickers if there are more than 10 unique tickers
        if actions_data["tic"].nunique() > 10:
            sample_tickers = actions_data["tic"].unique()[:10]
            actions_data = actions_data[
                actions_data["tic"].isin(sample_tickers)
```

```
            ]

        # Plot actions
        sns.lineplot(
            data=actions_data,
            x="date",
            y="action",
            hue="tic",
            ax=ax[1],
        )
        ax[1].set_title("Actions Over Time")
        ax[1].set_xlabel("Date")
        ax[1].set_ylabel("Actions")
        ax[1].legend(title="Ticker")

        plt.suptitle(f"Testing Results of {model_name.upper()} Agent", y=1)
        plt.tight_layout()
        plt.savefig(f"{self.directory}/{model_name}_test_evaluation.png")
        plt.show()
```

Listing 19: shap_visualiser.py

```
from typing import Optional

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import shap

shap.initjs()


class ShapVisualiser:
    def __init__(
        self,
        shap_values: shap.Explanation,
        action_space: pd.DataFrame,
        X_test: pd.DataFrame,
        directory: str,
        filename: str,
        model_name: str,
        shap_interaction_values: Optional[np.ndarray] = None,
    ) -> None:
        """
        Initializes the SHAP visualiser with SHAP values and action space.
        :param shap_values: SHAP explanation object containing the SHAP values.
        :param action_space: DataFrame containing the action space columns.
        :param X_test: DataFrame containing the test state space data.
        :param directory: Directory where plots will be saved.
        :param filename: Base filename for the saved plots.
        :param model_name: Name of the DRL model to which the feature importances belong
        ↪ to.
```

```python
        :param shap_interaction_values: SHAP interaction values for the features.
        """
        self.shap_values = shap_values
        self.action_space = action_space
        self.X_test = X_test
        self.directory = directory
        self.filename = filename
        self.model_name = model_name
        self.shap_interaction_values = shap_interaction_values

    def beeswarm_plot(
        self,
        index: int,
    ) -> None:
        """
        Create a beeswarm plot for the SHAP values.
        :param index: Index of the asset in the action space to plot.
        :return: None
        """
        ax = shap.plots.beeswarm(
            self.shap_values[..., index],
            show=False,
            max_display=10,
            group_remaining_features=False,
        )
        asset = self.action_space.columns[index]
        ax.set_title(f"SHAP Beeswarm Plot for {asset}")
        plt.savefig(
            f"{self.directory}/{self.model_name}_{self.filename}_shap_beeswarm_{asset}.png"
        )
        plt.show()

    def force_plot(
        self,
        index: int,
    ) -> None:
        """
        Create a force plot for the SHAP values.
        :param index: Index of the asset in the action space to plot.
        :return: None
        """
        force_plot = shap.plots.force(
            self.shap_values[..., index],
            feature_names=self.X_test.columns,
            link="logit",
        )

        shap.save_html(
            f"{self.directory}/{self.model_name}_{self.filename}_shap_force_{self.action_space.columns[index]}.html",
            force_plot,
        )
```

```
    def force_plot_single_obs(
        self,
        index: int,
        obs: int,
    ) -> None:
        """
        Create a force plot for a single observation.
        :param index: Index of the asset in the action space to plot.
        :param obs: Index of the observation to plot in time.
        :return: None
        """
        shap.plots.force(
            self.shap_values[obs, ..., index],
            feature_names=self.X_test.columns,
            link="logit",
            matplotlib=True,
            show=False,
        )
        asset = self.action_space.columns[index]

        plt.title(
            f"SHAP Force Plot for {asset} for observation {obs}",
            fontsize=16,
            y=1.5,
        )

        # Format value labels to two decimal places
        for text in plt.gca().texts:
            value = text.get_text()
            # If value contains = sign, format the number
            if "=" in value:
                parts = value.split("=")
                if len(parts) == 2:
                    try:
                        number = float(parts[1])
                        text.set_text(f"{parts[0]}= {number:.3f}")
                    except ValueError:
                        pass

        plt.savefig(
            f"{self.directory}/{self.model_name}_{self.filename}_shap_force_single_obs_
            ↪ {asset}_{obs}.png"
        )
        plt.show()

    def force_plot_assets(
        self,
        obs: int,
    ) -> None:
        """
        Create force plots for each asset in the portfolio.
        :param obs: Index of the observation to plot in time.
        """
        for index, _ in enumerate(self.action_space.columns):
```

```python
            self.force_plot_single_obs(index, obs)

    def waterfall_plot_single_obs(
        self,
        index: int,
        obs: int,
    ) -> None:
        """
        Create a waterfall plot for the SHAP values.
        :param index: Index of the asset in the action space to plot.
        :param obs: Index of the observation to plot in time.
        """
        shap.plots.waterfall(
            self.shap_values[obs, ..., index], show=False, max_display=10
        )

        asset = self.action_space.columns[index]

        plt.title(f"SHAP Waterfall Plot for {asset}")
        plt.savefig(
            f"{self.directory}/{self.model_name}_{self.filename}_shap_waterfall_{asset}.png"
        )
        plt.show()

    def heatmap(
        self,
        index: int,
    ) -> None:
        """
        Create a heatmap for the SHAP values.
        :param index: Index of the asset in the action space to plot.
        :return: None
        """
        shap.plots.heatmap(
            self.shap_values[..., index], max_display=10, show=False
        )
        asset = self.action_space.columns[index]
        plt.title(f"SHAP Heatmap for {asset}")
        plt.savefig(
            f"{self.directory}/{self.model_name}_{self.filename}_shap_heatmap_{asset}.png"
        )
        plt.show()

    def interaction_plot(
        self,
        index: int,
    ) -> None:
        """
        Create a summary plot for the SHAP interaction values.
        :param index: Index of the asset in the action space to plot.
        """
        if self.shap_interaction_values is None:
```

```python
            raise ValueError("SHAP interaction values are not provided.")

        shap.summary_plot(
            self.shap_interaction_values[..., index], self.X_test, show=False
        )

        asset = self.action_space.columns[index]
        plt.suptitle(
            f"SHAP Interaction Values for {asset}",
            y=1.1,
            fontsize=14,
        )
        plt.savefig(
            f"{self.directory}/{self.model_name}_{self.filename}_shap_interaction_{asse
            ↪  t}.png"
        )
        plt.show()
```

Listing 20: `style.mplstyle`

```
#### MATPLOTLIBRC FORMAT

## ***************************************************************************
## * LINES                                                                   *
## ***************************************************************************
## See https://matplotlib.org/stable/api/artist_api.html#module-matplotlib.lines
## for more information on line properties.
lines.linewidth: 1.0               # line width in points
lines.linestyle: -                 # solid line

## ***************************************************************************
## * FONT                                                                    *
## ***************************************************************************
## The font properties used by `text.Text`.


font.family:  sans-serif
font.style:   normal
font.variant: normal
font.weight:  normal
font.stretch: normal
font.size:    10.0


## ***************************************************************************
## * TEXT                                                                    *
## ***************************************************************************
## The text properties used by `text.Text`.

text.parse_math: True  # Use mathtext if there is an even number of unescaped
                        # dollar signs.

```

```
## ***************************************************************************
## * LaTeX                                                                   *
## ***************************************************************************
text.usetex: True   # use latex for all text handling.


## ***************************************************************************
## * AXES                                                                    *
## ***************************************************************************
## Following are default face and edge colors, default tick sizes,
## default font sizes for tick labels, and so on.  See
## https://matplotlib.org/stable/api/axes_api.html#module-matplotlib.axes
axes.facecolor:     0.97, 0.97, 0.97   # axes background color
axes.edgecolor:     0.8, 0.8, 0.8      # axes edge color
axes.linewidth:     1.0       # edge line width
axes.grid:          True      # display grid or not
axes.grid.axis:     both      # which axis the grid should apply to
axes.titlelocation: center    # alignment of the title: {left, right, center}
axes.axisbelow:     True      # draw axis gridlines and ticks:
                              #     - below patches (True)
                              #     - above patches but below lines ('line')
                              #     - above all (False)


axes.spines.left:   True  # display axis spines
axes.spines.bottom: True
axes.spines.top:    True
axes.spines.right:  True

axes.prop_cycle: cycler('color', ['ff85b6','ff8a91','ffbe88','fcea66','aaff99','8fd388'
    ,'6de1c4','4aefff','84d1ff','bdb2ff'])



## ***************************************************************************
## * GRIDS                                                                   *
## ***************************************************************************
grid.color:     (0.76, 0.78, 0.83)  # grid color
grid.linestyle: --          # solid
grid.linewidth: 0.8         # in points
grid.alpha:     1.0         # transparency, between 0.0 and 1.0


## ***************************************************************************
## * LEGEND                                                                  *
## ***************************************************************************
legend.loc:           best
legend.frameon:       True     # if True, draw the legend on a background patch
legend.fancybox:      True     # if True, use a rounded box for the
                                # legend background, else a rectangle
legend.shadow:        False    # if True, give background a shadow effect

## ***************************************************************************
```

```
## * FIGURE                                                                  *
## ***************************************************************************
figure.titlesize:   x-large     # size of the figure title (``Figure.suptitle()``)
figure.titleweight: normal    # weight of the figure title
figure.labelsize:   large     # size of the figure label (``Figure.sup[x|y]label()``)
figure.labelweight: normal    # weight of the figure label
figure.dpi:         200       # figure dots per inch
figure.facecolor:   white     # figure face color
figure.edgecolor:   white     # figure edge color
figure.frameon:    True      # enable figure frame

## Figure layout
figure.autolayout: True  # When True, automatically adjust subplot
                           # parameters to make the plot fit the figure
                           # using `tight_layout`


## ***************************************************************************
## * HISTOGRAM PLOTS                                                         *
## ***************************************************************************
hist.bins: 10  # The default number of histogram bins or 'auto'.


## ***************************************************************************
## * SAVING FIGURES                                                          *
## ***************************************************************************
## The default savefig parameters can be different from the display parameters
savefig.facecolor: 'white'        # figure face color when saving
savefig.transparent: False        # whether figures are saved with a transparent
↪  background by default
```

I.8 Optimisation Module

Listing 21: wandb.py

```
from typing import Tuple

import pandas as pd
import wandb
from wandb.integration.sb3 import WandbCallback

from agents.drl_agent import DRLAgent
from config import config_models
from environments.env_portfolio_optimisation import (
    PortfolioOptimisationEnvWrapper,
)
from pbenchmark.portfolio_benchmark import PortfolioBenchmark

wandb.login()


```

```python
class WandbOptimisation:
    def __init__(
        self,
        entity: str,
        project: str,
        train_data: pd.DataFrame,
        val_data: pd.DataFrame,
        test_data: pd.DataFrame,
        state_columns: list,
    ):
        """
        Initialize the WandbOptimisation class.
        :param entity: The entity name for Weights & Biases.
        :param project: The project name for Weights & Biases.
        :param train_data: The training data.
        :param val_data: The validation data.
        :param test_data: The test data.
        :param state_columns: The columns that represent the environment.
        """
        self.entity = entity
        self.project = project
        self.train_data = train_data
        self.val_data = val_data
        self.test_data = test_data
        self.state_columns = state_columns

        # Initialise wandb API
        self.api = wandb.Api()

        # Initialise portfolio benchmark
        self.portfolio_benchmark = PortfolioBenchmark()

    def wandb_train(
        self,
        model_name: str,
    ):
        """
        Train a model using Weights & Biases.
        :param model_name: The name of the model to train.
        :param train_data: The training data.
        :param val_data: The validation data.
        :param state_columns: The columns that represent the environment.
        """
        with wandb.init(settings={"quiet": "True"}) as run:
            environment = PortfolioOptimisationEnvWrapper(
                train_data=self.train_data,
                trade_data=self.val_data,
                state_columns=self.state_columns,
                verbose=0,
            )

            env_train = environment.get_train_env()
            gym_env, _ = environment.get_trade_env()

            agent = DRLAgent(run)
```

```python

            configuration = wandb.config.as_dict()

            model = agent.get_model(
                model_name,
                model_kwargs=configuration,
                environment=env_train,
                directory=None,
                use_case="portfolio-optimisation",
                verbose=0,
            )

            trained_model = agent.train(
                model,
                tb_log_name=model_name,
                callback=WandbCallback(),
            )

            df_account, _ = agent.predict(trained_model, gym_env)

            metrics = {
                "sharpe_ratio": self.portfolio_benchmark.compute_sharpe_ratio(
                    df_account
                ),
                "cumulative_return": self.portfolio_benchmark.compute_cum_returns(
                    df_account
                ),
            }

            wandb.log(metrics)

    def sweep(
        self,
        sweep_config: dict,
        model_name: str,
        number_trials: int = 5,
    ) -> str:
        """
        Start a sweep for hyperparameter optimization.
        :param sweep_config: The sweep configuration.
        :param model_name: The name of the model to optimize.
        :param number_trials: The number of trials to run.
        :return: The sweep ID.
        """
        configuration = sweep_config
        configuration["parameters"] = config_models.MODEL_SWEEP_CONFIG[
            model_name
        ]

        sweep_id = wandb.sweep(configuration, project=self.project)
        wandb.agent(
            sweep_id,
            lambda model_name=model_name: self.wandb_train(model_name),
            count=number_trials,
        )
```

```python

        return sweep_id

    def get_best_sweep_run(
        self, sweep_id: str, model_name: str
    ) -> Tuple[str, dict]:
        """
        Get the best sweep run for a given model.
        :param sweep_id: The ID of the sweep.
        :param model_name: The name of the model.
        :return: The best run ID and configuration.
        """
        sweep = self.api.sweep(f"{self.entity}/{self.project}/{sweep_id}")
        runs = sweep.runs

        best_run = max(
            runs,
            key=lambda run: run.summary.get("sharpe_ratio", 0),
        )
        best_run_config = best_run.config

        best_config = dict()
        print("Best run configuration:")
        for key in config_models.MODEL_SWEEP_CONFIG[model_name]:
            best_config[key] = best_run_config.get(key, 0)
            if type(best_config[key]) is float:
                print(f"\t{key}: {best_config[key]:.6f}")
            else:
                print(f"\t{key}: {best_config[key]}")

        print("Best run metrics:")
        for key in [
            "sharpe_ratio",
            "cumulative_return",
        ]:
            print(f"\t{key}: {best_run.summary.get(key, 0):.4f}")

        return best_run.id, best_config

    def test_best_run(
        self,
        model_name: str,
        configuration: dict,
        train_val_data: pd.DataFrame,
        logs_directory: str,
        models_directory: str,
    ) -> Tuple[pd.DataFrame, pd.DataFrame]:
        """
        Test the best run configuration for a given model.
        :param model_name: The name of the model.
        :param configuration: The configuration of the model.
        :param train_val_data: The training and validation data.
        :param logs_directory: The directory to save logs.
        :param models_directory: The directory to save models.
        :return: The account and actions DataFrames.
```

```python
        """
        environment = PortfolioOptimisationEnvWrapper(
            train_data=train_val_data,
            trade_data=self.test_data,
            state_columns=self.state_columns,
        )

        env_train = environment.get_train_env()
        gym_env, _ = environment.get_trade_env()

        agent = DRLAgent()

        model = agent.get_model(
            model_name,
            model_kwargs=configuration,
            environment=env_train,
            directory=logs_directory,
            use_case="portfolio-optimisation",
        )

        print(f"Training model: {model_name.upper()}")
        trained_model = agent.train(
            model,
            tb_log_name=model_name,
        )

        print(f"Saving model: {model_name.upper()}")
        agent.save_model(
            model=model,
            model_name=model_name,
            directory=models_directory,
        )

        print(f"Evaluating model: {model_name.upper()}")
        df_account, df_actions = agent.predict(
            model=trained_model,
            environment=gym_env,
        )

        return df_account, df_actions
```