Thesis Dissertation

TRANSFORMER BASED ARCHITECTURES FOR FINANCIAL PORTFOLIO OPTIMIZATION

Ioanna Hadjipolla

University of Cyprus

Department of Computer Science

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

Transformer Based Architectures for Financial Portfolio Optimization

Ioanna Hadjipolla

Supervisor
Chryssis Georgiou

The thesis was submitted in partial fulfillment of the requirements for the degree in Computer Science at the Department of Computer Science, University of Cyprus.

May 2025

Acknowledgments

I would like to express my deepest gratitude to my supervisor, Chryssis Georgiou, for their invaluable guidance, encouragement, and insightful feedback throughout the course of this research.

Special thanks go to my friends and colleagues for their motivation, collaboration, and thoughtful discussions, which have contributed to the evolution of this research. Above all, I am profoundly grateful to my family for their patience, and unwavering belief in me. Their emotional support has been my foundation throughout this journey. To everyone who contributed directly or indirectly to the completion of this thesis, your help and inspiration have meant more than words can express.

Abstract

This thesis investigates the application of Transformer-based architectures to financial portfolio optimization. Motivated by the unique challenges of high volatility, complex temporal dependencies, and non-stationary behavior in financial time series, the study explores how recent advances in deep learning, specifically Transformers, can improve decision-making in trading environments. Several Transformer architectures including encoder-decoder models, encoder-only variants, Generative Pretrained Transformers (GPT), and a Vision Transformer (ViT) adapted for time series, are implemented and evaluated for their ability to extract meaningful, high-level representations from historical market data. Each model is designed to capture complex temporal dependencies and latent structures inherent in financial time series. These extracted features are then fed into a Deep Q-Network (DQN) agent, which utilizes them to learn an effective trading policy aimed at maximizing long-term cumulative rewards. To rigorously evaluate the benefit of Transformer-enhanced feature extraction, a baseline DQN model is also trained directly on raw historical data without any pre-processing by Transformer models. Comparing the performance of both approaches highlights the added value of Transformer-based representations in guiding more strategic and profitable trading decisions. Additionally, the thesis considers advanced Transformer architectures such as the Temporal Fusion Transformer (TFT) and Informer, which demonstrate strong potential for time-series modeling. Although implementation of these models was limited by resource and reproducibility constraints, their capabilities and future applicability are discussed. Overall, the thesis contributes a systematic analysis of Transformer-based feature extraction for deep reinforcement learning in finance, offering insights into model design, training methodology, and the practical impact of architectural choices on portfolio performance.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

The motivation behind this study is to explore the untapped potential of *Transformer based architectures* [47] in financial *portfolio optimization* [6], with a particular focus on the rapidly evolving cryptocurrency markets. Transformers have revolutionized deep learning by overcoming key limitations of traditional sequential models such as RNNs and LSTM networks, which often suffer from vanishing gradients and restricted memory capabilities. Unlike these models, Transformers employ self-attention mechanisms that enable the parallel processing of entire sequences, allowing them to efficiently learn complex relationships between financial variables, detect emerging trends, and adapt to volatile market conditions. This architectural advantage has led to remarkable success in natural language processing and has shown *promising results in financial tasks* [49] such as stock price forecasting, risk assessment, and *asset allocation* [48]. However, their application in portfolio optimization remains largely unexplored. Given the growing need for adaptive, data driven strategies in navigating the non linear dynamics of modern financial systems, this study aims to investigate how Transformers can be leveraged to enhance portfolio performance and robustness, particularly in challenging market environments.

1.2 Goal of the Study

This thesis aims to bridge the gap between transformer based deep learning models and financial portfolio optimization. By integrating transformers into an existing portfolio optimization project, this study will evaluate their effectiveness in improving asset allocation strategies, enhancing risk-adjusted returns, and adapting to market fluctuations. The study will involve empirical testing and comparative analysis against other deep learning based approaches. Ultimately, this study seeks to demonstrate that transformer models, with their ability to process and analyze large volumes of financial data, can offer a novel and effective approach to portfolio optimization. By leveraging their advanced pattern recognition and predictive capabilities, investors and financial institutions may be able to construct more resilient and adaptive portfolios, particularly in volatile markets like cryptocurrencies.

1.3 Methodology

The methodology followed in this study began with an initial meeting involving my academic supervisor and a doctoral candidate whose ongoing PhD research serves as the foundation for this thesis. During this meeting, we clarified the scope and requirements of the dissertation and outlined a preliminary timeline to guide the research process. Given my limited prior exposure to the subject matter, I commenced by developing a solid understanding of general machine learning concepts, gradually narrowing my focus to Transformer architectures and their potential applications in financial portfolio optimization. My initial research concentrated on two advanced Transformer based models, *Temporal Fusion Transformer* [43] and *Informer* [45], which appeared particularly promising for time-series forecasting tasks. However, due to the scarcity of reliable documentation and open-source implementations, I was ultimately unable to successfully deploy these models and extract meaningful results. As a result, I selected four alternative Transformer models that embody different architectural paradigms: the *original Transformer (vanilla)* [19], an *encoder-only Transformer* [37][38], *Vision Transformer* [40], and *Generative Pre-Trained Transformer (GPT)* [38][39]. These models represent the three main Transformer configurations: encoder-decoder, encoder-only, and decoder-only. Given that Transformers are typically designed to process tokenized sequence data, a key part of the implementation involved adapting the input pipeline to handle numerical financial data. This required preprocessing the dataset to a format compatible with each model's architecture, including appropriate embedding and normalization techniques. After successfully executing all four models, I extracted output data from each and conducted a performance comparison to evaluate their effectiveness. Following this comparison, I selected the two highest-performing Transformer models to support the next phase of the thesis: integrating Deep Reinforcement Learning (DRL) for portfolio optimization. Specifically, these selected models were repurposed as feature extractors, generating informative representations of the financial data to be used as inputs to a DRL algorithm, in this case, a *Deep Q-Network (DQN)* [28]. The integrated approach was tested by comparing the portfolios generated under each Transformer-enhanced method, allowing for an assessment of the value added by Transformer-based feature extraction in a reinforcement learning framework.

1.4 Thesis Organization

The rest of this thesis is split into six chapters. Table 1.1 reports the content of each chapter.

Chapter	Description
2	This chapter describes the background knowledge for this thesis. Specifically, it provides an overview of the Transformer architecture, its evolution from earlier sequential models, and its growing application in financial modeling. It also explores the challenges of portfolio optimization in cryptocurrency markets and highlights the motivation behind using Transformer-based models for this purpose.
3	This chapter focuses on the use of transformer models for portfolio optimization. It covers the construction of simple transformer architectures for feature extraction, key hyperparameter adjustments, examples of their application in finance, and methods for adapting financial data sequences to enhance model performance.
4	This chapter presents the practical implementation of various transformer-based architectures tailored for financial forecasting. It begins by detailing the encoding of numerical data for transformer models and introduces the specific architectures employed in the study, including the vanilla encoder-decoder transformer, encoder-only transformer, GPT transformer, and a Vision Transformer adapted for time series. For each model, architectural design, training procedures, hyperparameters, and performance metrics are thoroughly discussed, accompanied by visual performance analysis.
5	Chapter 5 outlines key design decisions that facilitate the agent's ability to optimize its portfolio using deep reinforcement learning

	(DRL). Central to this approach is the use of transformer-based models, specifically the Generative Pretrained Transformer (GPT), for feature extraction from financial time series data. These model are employed to capture complex temporal dependencies and intricate patterns within the market. The extracted features are then fed into the Deep Q-Network (DQN), which leverages them for portfolio optimization, enabling the agent to make informed trading decisions. In addition to this integrated approach, the DQN model is also run independently without transformer-based feature extraction to establish a performance baseline. This allows for a direct comparison of results, highlighting the impact of transformer-enhanced features on trading performance. This combination of transformer architectures for feature extraction and DRL for decision-making forms the core of the agent's ability to maximize long-term returns in a dynamic financial environment.
6	This concluding chapter synthesizes the exploration of Transformer-based architectures for financial portfolio optimization, highlighting both practical experimentation and future potential. It reviews the successful application of GPT as a feature extractor in combination with a Deep Q-Network, emphasizing its performance advantages and suitability for modeling volatile financial time series. Simultaneously, the chapter examines the promise of advanced architectures like the Temporal Fusion Transformer (TFT) and Informer, which offer capabilities for capturing complex temporal dynamics and long-term dependencies. However, due to resource constraints and the lack of accessible, validated implementations, their integration remained beyond the scope of this work. The chapter reflects on key findings, outlines limitations, and suggests future research directions aimed at advancing the practical use of Transformers in financial decision-making.

Table 1.1: Chapter Descriptions

Chapter 2

Background Knowledge

2.1 Digital Asset Markets

2.1.1 Introduction to Digital Assets and Market Evolution

The *digital asset market* [1] has evolved rapidly since its inception, with the rise of cryptocurrencies like Bitcoin marking the beginning of a transformative shift in the global financial landscape. Initially perceived as speculative and niche, digital assets have gained substantial traction as both an alternative and complementary investment class. The market has expanded to encompass a variety of digital assets, including cryptocurrencies, stablecoins, security tokens, and tokenized real-world assets. Central to this transformation has been the underlying blockchain technology, which provides decentralization, transparency, and security. The increasing institutional adoption of digital assets, driven by products such as digital asset ETFs and growing regulatory clarity, has further cemented their position as a legitimate asset class. As the market matures, it presents both unique opportunities for diversification and challenges related to volatility and security risks.

2.1.2 Types of Digital Assets

Digital assets are broadly classified into several types, each offering distinct investment opportunities. The most known category is cryptocurrencies, which include assets like Bitcoin and Ether. These are *digital currencies* [2] designed to function as a medium of exchange, relying on blockchain technology for decentralization. Another prominent type is *stablecoins* [2], which are pegged to traditional currencies such as the U.S. Dollar, providing stability amidst the inherent volatility of other digital assets. *Security tokens* [2] represent a tokenized version of real-world assets, offering fractional ownership of equity or debt in real world entities. Additionally, indirect exposure to digital assets can be gained through products like exchange traded funds and exchange traded products, which aggregate a variety of digital assets into a single investment vehicle. Venture capital investments also play a significant role in the digital asset ecosystem, particularly in funding early stage blockchain startups and crypto native companies.

2.1.3 Risks and Challenges in Digital Asset Investments

While digital assets offer substantial investment potential, they also come with a unique set of *risks and challenges* [1][3]. The most significant risk is volatility, as digital asset prices can experience dramatic fluctuations, often influenced by market sentiment, regulatory developments, and technological advancements. Although volatility has decreased somewhat in recent months, digital assets remain prone to significant price swings, which can lead to substantial gains or losses. Liquidity is another concern, as while major cryptocurrencies like Bitcoin and Ether are relatively liquid, smaller or lesser known digital assets may be harder to trade, particularly during market downturns. Cybersecurity threats, including hacking, fraud, and market manipulation, pose additional risks, making it essential for investors to use reputable exchanges and secure their holdings. Finally, the complexities of digital asset technologies mean that investors must conduct thorough due diligence, understanding the underlying blockchain protocols, the teams behind digital assets, and the legal and regulatory landscape to avoid potential pitfalls.

2.2. Portfolio Optimization

2.2.1 Introduction to Portfolio Optimization

Portfolio optimization [6] is a fundamental process in finance aimed at determining the optimal combination of assets within an investment portfolio to achieve the highest return for a given level of risk. This process involves the use of mathematical models and quantitative techniques to balance risk and return by analyzing various asset combinations under different market scenarios. The goal is to construct an "efficient" portfolio that maximizes expected returns while minimizing risk, ensuring it aligns with an investor’s specific risk tolerance and investment objectives. A portfolio is considered efficient if it lies on the *"efficient frontier"* [7], which represents the set of portfolios that offer the best risk adjusted return, meaning no other portfolio provides a higher expected return for the same level of risk.

2.2.2 Key Learning Points in Portfolio Optimization

Portfolio optimization relies on quantitative methods to balance assets with the goal of maximizing risk adjusted returns. Key learning points include the understanding that portfolio optimization is not only about selecting the best assets but also about diversifying the portfolio to reduce risk. Popular *portfolio optimization methods* [8] include Modern Portfolio Theory, the Black-Litterman Model, and Monte Carlo Simulation. Each method has its own strengths and limitations depending on the characteristics of the investment strategy, such as the investor's risk tolerance, return expectations, and available data. Portfolio optimization requires a clear definition of investment objectives, risk profile, and time horizon, alongside access to reliable data sources. Although an optimized portfolio typically leads to superior risk adjusted returns, the complexity of the process often necessitates expert knowledge and advanced tools for accurate implementation.

2.2.3 Advantages and Disadvantages of Portfolio Optimization

Portfolio optimization offers several advantages that can significantly enhance the performance and risk management of an investment portfolio. However, it also comes with certain limitations and challenges.

- Advantages:
    - *Maximized Risk-Adjusted Returns* [13]: Optimization helps investors build portfolios that maximize expected returns for a given level of risk, which can lead to improved long-term financial performance.
    - *Risk Management* [14][15]: By diversifying assets across various classes and sectors, portfolio optimization reduces overall portfolio risk and helps investors stay within their risk tolerance levels.
    - *Enhanced Decision-Making* [15]: Portfolio optimization enables investors to systematically assess asset combinations, uncovering potentially lucrative opportunities that may not be apparent through simple analysis.
- Disadvantages:
    - *Dependence on Historical Data* [16]: Many portfolio optimization models, such as MVO, rely on historical data, which may not always reflect future market conditions, leading to estimation errors or suboptimal outcomes.
    - *Model Assumptions* [13]: Certain models assume that asset returns follow a normal distribution, which may ignore the impact of extreme market events, thus limiting their accuracy.
    - *Complexity and Expertise* [13]: Implementing optimization models can be resource-intensive and require advanced knowledge, especially when handling large datasets or using complex techniques like Monte Carlo Simulation.

2.2.4 Advancing Portfolio Optimization: The Role of Digital Assets

As digital assets like cryptocurrencies have become a more prominent part of investment portfolios, traditional portfolio optimization methods have faced challenges. Digital assets exhibit high volatility, low correlation with traditional assets, and often lack sufficient historical data, making portfolio optimization more complex. However, advanced techniques such as machine learning, deep learning, and reinforcement learning have emerged as potential solutions. These methods allow for the development of adaptive portfolio optimization strategies that can adjust in real-time based on evolving market conditions and investor preferences. Furthermore, the use of *on-chain data* [17], which refers to information that is publicly available and recorded directly on *blockchain networks* [50], such as transaction volumes, wallet activity, smart contract interactions, and token flows, provides more granular insights into market behavior, enhancing the accuracy and effectiveness of portfolio optimization models. As the digital asset market continues to grow, integrating these advanced techniques will be critical in refining portfolio optimization strategies for this new asset class.

2.3 Transformers

2.3.1 Transformers: Revolutionizing NLP Applications

Transformers are an architecture that has significantly transformed the field of *Natural Language Processing (NLP)* [51]. Many of the most powerful and effective NLP algorithms today are based on the transformer architecture. These models have been successfully applied in a wide range of applications, including machine translation, conversational chatbots, and other NLP tasks such as text summarization, sentiment analysis, and question answering.

2.3.2 The Need for Transformers: Overcoming the Limitations of Previous Architectures

Starting with *Recurrent Neural Networks (RNNs)* [52], there are issues with vanishing gradients that make it difficult to capture long-range dependencies in sequences. To address these *problems* [18], we introduced *Gated Recurrent Units* (GRUs) [53] and then *Long Short-Term Memory (LSTMs)* [54], which use gates to control the flow of information. While these units improved control, they also added computational complexity. All of these models are sequential, meaning they process the input sentence one word or token at a time. As a result, each unit creates a bottleneck in the flow of information, as the output of unit N can only be computed after processing all preceding units.

Figure 2.1 illustrates the basic architecture of a Recurrent Neural Network (RNN. In this unrolled form, the RNN is shown as a series of repeating modules (denoted as "A") that share the same parameters across each time step. Each module takes as input the current element of the input sequence xtxt and the hidden state from the previous time step h(t−1), and produces the current hidden state htht. This recurrent connection allows the RNN to maintain a form of memory across the sequence, enabling it to capture temporal dependencies. The diagram highlights how the hidden state evolves as new inputs are processed in sequence.

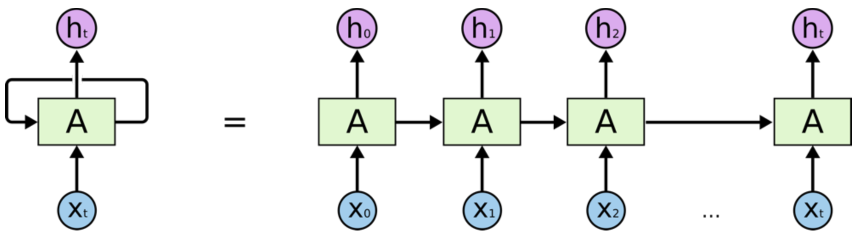


Figure 2.1 Unrolled RNN Depicting Its Sequential Form

Figure 2.2 depicts the internal architecture of a Long Short-Term Memory (LSTM) unit, a specialized type of Recurrent Neural Network (RNN) designed to address the limitations of standard RNNs in learning long-term dependencies. Each LSTM cell consists of three key gates: the forget gate, input gate, and output gate, which regulate the flow of information using the sigmoid (σ) and tanh activation functions. The sigmoid function outputs values between 0 and 1 and is used in all three gates to control how much information should be passed through—0 means "ignore" and 1 means "keep." The tanh function outputs values between -1 and 1 and is used to scale and shape the new candidate values being added to the cell state. The forget gate determines what portion of the previous cell state should be discarded, the input gate decides what new information to store, and the output gate controls what part of the cell state contributes to the hidden state output. This structure allows LSTMs to maintain and update information over long sequences.

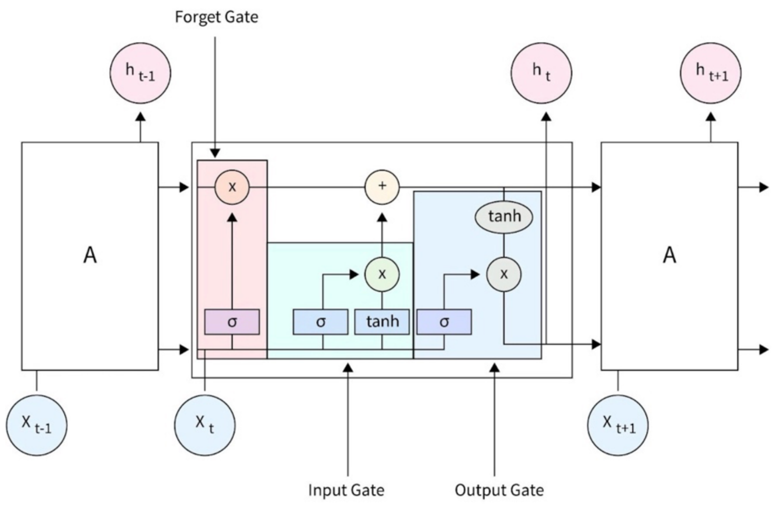
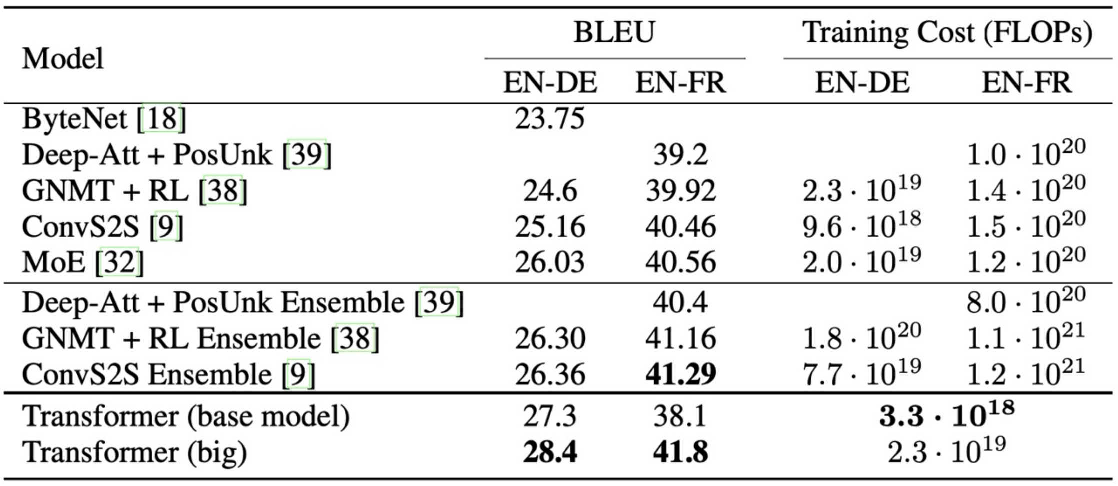


Figure 2.2 Unrolled LSTM Depicting Its Sequential Form

The Transformer architecture, however, allows for parallel processing of an entire sequence, enabling the model to ingest an entire sentence at once rather than processing it one word at a time from left to right. The Transformer network was introduced in the seminal paper *"Attention Is All You Need" (2017)* [19] by Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. This paper introduced the self-attention mechanism, which became the foundation for many modern deep learning models, including *Bidirectional Encoder Representations from Transformers (BERT)* [55] and GPT. The major innovation of the Transformer architecture lies in its combination of attention-based representations and a *Convolutional Neural Network (CNN)* [56] processing style. To understand Transformers, it is essential to understand how the attention mechanism works. As a model generates text word by word, it can reference and attend to words that are relevant to the current word being generated. This attention is learned during training through backpropagation. The key strength of the attention mechanism is that it does not suffer from the short-term memory limitations of RNNs, which are constrained by a limited window of reference. Although GRUs and LSTMs offer improved memory capabilities and larger reference windows, they still do not provide the infinite reference window that Transformers do.

Table 2.1 compares the performance and computational efficiency of various neural machine translation models using the *BiLingual Evaluation Understudy (BLEU)* [57] score metric across English-German (EN-DE) and English-French (EN-FR) translation tasks, along with their corresponding training costs measured in floating point operations (FLOPs). Traditional RNN-based and convolutional models such as GNMT and RL, ConvS2S, and Deep-Att and PosUnk are listed alongside more advanced models like the Transformer. The results highlight that the Transformer model, particularly in its 'big' configuration, achieves the highest BLEU scores (28.4 for EN-DE and 41.8 for EN-FR), indicating superior translation quality. Moreover, the base Transformer significantly outperforms other models in terms of training efficiency, achieving competitive accuracy with much lower computational cost (3.3×10^{18} FLOPs for EN-DE). This demonstrates the Transformer's architectural advantage over traditional RNN-based models by enabling parallelization and more efficient training, while also achieving state-of-the-art translation performance.



Model	BLEU		Training Cost (FLOPs)	
	EN-DE	EN-FR	EN-DE	EN-FR
ByteNet [18]	23.75			
Deep-Att + PosUnk [39]		39.2		$1.0 \cdot 10^{20}$
GNMT + RL [38]	24.6	39.92	$2.3 \cdot 10^{19}$	$1.4 \cdot 10^{20}$
ConvS2S [9]	25.16	40.46	$9.6 \cdot 10^{18}$	$1.5 \cdot 10^{20}$
MoE [32]	26.03	40.56	$2.0 \cdot 10^{19}$	$1.2 \cdot 10^{20}$
Deep-Att + PosUnk Ensemble [39]		40.4		$8.0 \cdot 10^{20}$
GNMT + RL Ensemble [38]	26.30	41.16	$1.8 \cdot 10^{20}$	$1.1 \cdot 10^{21}$
ConvS2S Ensemble [9]	26.36	**41.29**	$7.7 \cdot 10^{19}$	$1.2 \cdot 10^{21}$
Transformer (base model)	27.3	38.1	$\mathbf{3.3 \cdot 10^{18}}$	
Transformer (big)	**28.4**	**41.8**	$2.3 \cdot 10^{19}$	

Table 2.1 The Transformer achieves better BLEU scores than previous state-of-the-art models on the English-to-German and English-to-French newstest2014 tests at a fraction of the training cost [19].

2.3.3 Understanding Transformers: The Architecture Behind Modern NLP

Introduction to Transformers and Encoder-Decoder Overview

Transformers [20] are an attention-based encoder-decoder architecture that has revolutionized natural language processing. At a high level, the encoder maps an input sequence into a continuous representation, capturing all its learned information, while the decoder takes this representation and generates an output sequence step by step until an <end> token is produced. The process begins with the Input Embedding layer, which converts words into continuous-value vectors, allowing neural networks to process them effectively. Since transformers lack recurrence, a Positional Encoding step is necessary to retain word order information. This is achieved using sine and cosine functions, which provide unique positional information for each word in the sequence.

The Encoder Structure and Self-Attention Mechanism

The Encoder Layer is responsible for transforming input sequences into abstract representations. It consists of two key submodules: multi-headed self-attention and a feed-forward network, both of which are followed by residual connections and layer normalization to stabilize the network. The self-attention mechanism allows the model to relate each word to every other word in the input, ensuring that important contextual relationships are captured. To achieve this, the model generates query, key, and value vectors for each word, similar to a search system where a query is matched against keys to retrieve relevant values.

Attention Scores, Multi-Headed Attention, and Feed-Forward Network

The queries and keys undergo dot product multiplication, producing a score matrix that determines how much focus each word should place on other words in the sequence. These scores are then scaled down to prevent large gradient updates, passed through a Softmax function to normalize them into probability values, and multiplied by the corresponding value vectors to adjust their importance. The multi-headed attention mechanism enhances the model's ability to capture diverse relationships by splitting the attention process into multiple independent "heads," each focusing on different aspects of the input. The attention outputs are then concatenated and passed through a linear layer for further processing. This is followed by residual connections, which add the original input back to the processed attention output, allowing gradients to flow more effectively. The result is then normalized and sent through a point-wise feed-forward network, which applies additional transformations to refine the learned representation.

The Decoder and Autoregressive Generation

The encoder’s primary role is to encode the input into a continuous representation, enriched with attention-based contextual information that will be useful for the decoder. The decoder is responsible for generating the output sequence and follows a similar structure to the encoder but with key differences. It includes two multi-headed attention layers, a feed-forward network, and residual connections with layer normalization after each submodule. The first multi-headed attention layer processes the decoder’s input while ensuring autoregressive generation, meaning it does not have access to future words in the sequence. To achieve this, a look-ahead mask is applied, preventing the model from attending to future tokens by replacing those values with negative infinities before applying the Softmax function. This ensures that each word is generated sequentially without "cheating" by looking ahead. The second multi-headed attention layer plays a crucial role in aligning the encoder’s output with the decoder’s input. Instead of computing self-attention, it takes the encoder’s output as the query and key, while the decoder’s processed inputs act as the values. This mechanism helps the decoder determine which parts of the encoder's representation should be attended to for generating the next word in the sequence. The output is then passed through a feed-forward networkfor further transformation before being processed by a final linear layer, which acts as a classifier. The classifier's output is sent to a Softmax layer, which assigns probability values to each possible word. The word with the highest probability is selected as the next output word. The decoder repeats this process iteratively, taking the previously generated words as input and continuing until it predicts an <end> token, signaling the completion of the sequence. By stacking multiple layers of encoders and decoders, transformers can learn increasingly complex attention patterns, allowing them to capture intricate relationships within the data and significantly enhance predictive performance.

The concepts described above are depicted schematically in Figure 2.3

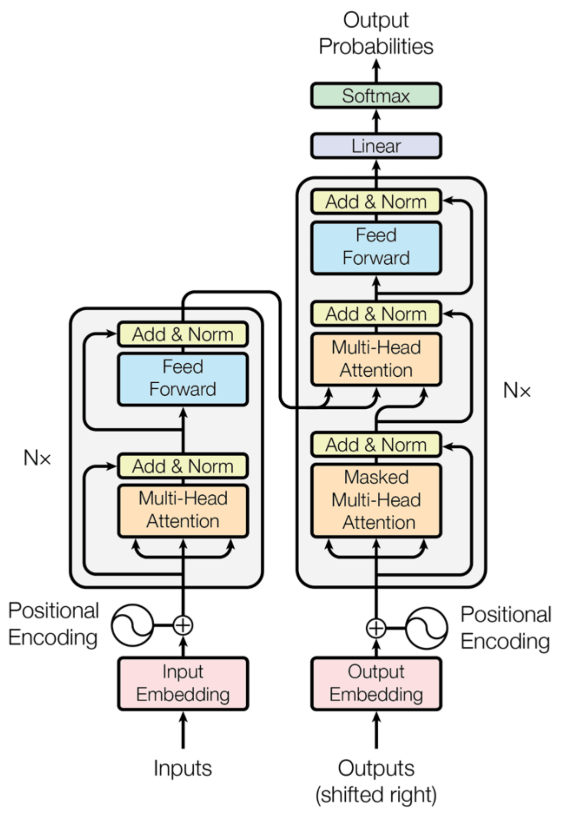


Figure 2.3 Transformer model architecture

2.3.4 Transformers for Portfolio Optimization in Financial Markets

The increasing complexity and dynamism of financial markets have fueled the need for more sophisticated portfolio optimization techniques. Traditional financial models, such as Mean-Variance Optimization and risk-based strategies, have long been used to balance risk and return. However, these approaches often fail to fully capture the non-linearity, high volatility, and rapidly evolving nature of modern financial assets, particularly cryptocurrencies. The emergence of machine learning and deep learning techniques has introduced a new paradigm in portfolio management, allowing for more adaptive and data-driven decision-making. Among these, transformer-based models initially developed for natural language processing have demonstrated remarkable success in various domains, including time-series forecasting and financial prediction [58]. Their ability to model long-range dependencies and capture intricate patterns in sequential data makes them particularly promising for financial applications, where understanding trends and market dynamics is crucial for effective investment strategies.

2.3.5 The Role of Transformers in Financial Modeling

Transformers have revolutionized deep learning by overcoming the limitations of recurrent neural networks and long short-term memory models. Unlike traditional sequential models that suffer from vanishing gradient problems and limited memory, transformers leverage self-attention mechanisms to process entire sequences of data simultaneously [59]. This ability allows them to efficiently learn complex relationships between financial variables, detect emerging trends, and adapt to volatile market conditions. While transformers have been widely applied in natural language processing, their recent adoption in financial modeling has shown promising results in stock price forecasting, risk assessment, and asset allocation. However, their potential in portfolio optimization remains largely unexplored, especially in the context of *cryptocurrency markets* [21], where traditional models struggle to cope with extreme fluctuations and unpredictable market behavior.

2.3.6 Challenges in Cryptocurrency Portfolio Optimization

Cryptocurrency markets present unique challenges for portfolio optimization. Unlike traditional financial assets, cryptocurrencies exhibit higher volatility, fragmented liquidity, and less established regulatory frameworks. Their prices are often driven by speculative trading, market sentiment, and macroeconomic factors that are difficult to quantify using conventional methods. Existing portfolio optimization techniques, such as Markowitz's mean-variance framework, struggle to adapt to these dynamic conditions, leading to suboptimal asset allocation and risk management. While machine learning and deep learning methods, including reinforcement learning and evolutionary algorithms, have been explored for cryptocurrency portfolio construction, there remains a gap in the literature regarding the application of transformer-based architectures in this domain. The ability of transformers to analyze large-scale financial data and uncover intricate patterns makes them a compelling candidate for optimizing cryptocurrency portfolios.

2.4 Reinforcement Learning

2.4.1 Introduction to Deep Reinforcement Learning

Deep reinforcement learning (DRL) [22] is a branch of machine learning that combines two major areas: reinforcement learning (RL) and deep learning. *Reinforcement learning* [23] focuses on training an agent to make decisions by interacting with an environment, usually through a process of trial and error. *Deep learning* [24], on the other hand, involves training deep neural networks to process and learn from complex, high-dimensional data such as images or sound.

When these two approaches are combined, the resulting DRL methods are capable of learning effective decision-making strategies directly from raw sensory input, without needing manually designed features. This has enabled significant progress in areas like robotics, game-playing, and autonomous control systems. Notably, DRL systems have achieved impressive results in tasks that were previously considered too complex.

Figure 2.4 illustrates the fundamental workflow of a Deep Reinforcement Learning (DRL) system, which involves the interaction between an RL-agent and an RL-environment. The RL-agent, typically implemented as a deep neural network, receives observations from the environment and uses them to make actions that affect the environment. Based on these actions, the environment responds with new observations and provides rewards that indicate the success or failure of the agent's behavior. These rewards are used by the agent to learn and adjust its decision-making policy over time. This closed feedback loop enables the agent to optimize its actions through trial and error, aiming to maximize cumulative rewards. The diagram captures the essential dynamics of reinforcement learning, where learning is driven by interaction rather than supervised data.

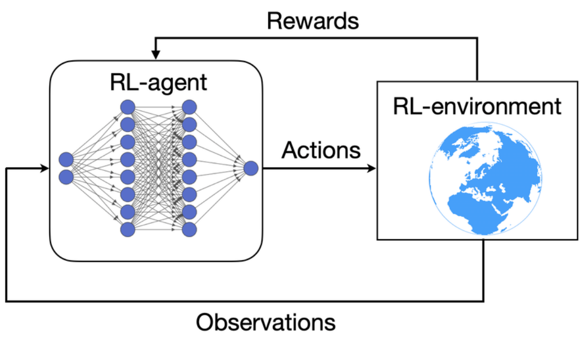


Figure 2.4 The Workflow of DRL

2.4.2 Fundamentals

2.4.2.1 Reinforcement Learning

In Reinforcement Learning, the core idea is that an agent learns to make decisions by receiving feedback from its environment. The agent is placed in a setting where it can observe a state, choose an action, and receive a reward based on the outcome. Over time, it learns to choose actions that maximize the total reward it receives. This learning setup is commonly modeled using a *Markov Decision Process (MDP)* [25], which defines:

- The possible states the agent can be in,
- The actions it can take,
- The probabilities of transitioning between states,
- And the rewards it gets for those transitions.

The agent's goal is to learn a *policy* [26], essentially a strategy for choosing actions, that maximizes the expected cumulative reward.

2.4.2.2 Deep Learning

Deep learning uses neural networks with multiple layers to learn patterns in data. These networks are particularly good at handling raw input like images, audio, or text because they can learn useful feature representations automatically, without needing to be explicitly told what to look for.

In the context of DRL, neural networks are used to approximate different functions, such as:

- The policy itself (mapping from state to action),
- Value functions (how good it is to be in a certain state),
- Or Q-functions (how good it is to take a certain action in a state).

This enables RL to scale to problems with very large or continuous state spaces that traditional methods would struggle with.

2.4.3 Key Algorithms

There are two broad categories of DRL algorithms: **model-free** and **model-based**.

2.4.3.1 Model-Free Methods

Model-free methods [27] do not attempt to learn a model of the environment. Instead, they focus on learning the policy or value functions directly through interactions with the environment. These approaches are often easier to implement and more flexible in complex, high-dimensional domains where modeling the environment explicitly can be challenging.

- ***DQN (Deep Q-Network)*** [28]

  One of the earliest and most influential deep RL algorithms, DQN was introduced by DeepMind to play Atari games using only raw pixels as input. It combines Q-learning with deep convolutional neural networks to approximate the Q-function, which estimates the expected return for taking an action in a given state.

- ***Policy Gradient Methods*** [29]

  These methods directly optimize the policy by estimating gradients of expected rewards with respect to the policy parameters. They are particularly useful in environments with continuous or high-dimensional action spaces, where value-based methods may struggle.

- ***Deep Policy Networks*** [30]

  A deep policy network is a neural network that directly maps input states to actions by parameterizing the policy itself.

  Unlike value-based methods, which estimate the expected future return of states or state-action pairs, policy-based methods learn the optimal behavior directly through the policy function $\pi(a|s;\theta)$, where a is the action, s is the state, and θ represents the parameters (weights and biases) of the neural network. This function outputs the probability of selecting action a given the state s.

  The policy is typically trained using gradient ascent on the expected cumulative reward, which adjusts the parameters θ to improve the network's performance over time. Deep policy networks are especially useful in environments with high-dimensional or continuous action spaces, where traditional value-function-based methods become impractical.

  These networks can also represent stochastic policies, making them particularly suited for tasks requiring complex, flexible decision-making strategies.

- ***Actor-Critic Methods*** [31]

  These combine the strengths of both policy gradient and value-based methods. The "actor" network is responsible for selecting actions (the policy), while the "critic" network evaluates how good those actions are (the value function). This structure helps reduce the high variance often encountered in pure policy gradient methods and leads to more stable and sample-efficient learning.

2.4.3.2 Model-Based Methods

Model-based methods [32] try to learn a model of how the environment works (i.e., how states transition based on actions). This model can then be used to simulate outcomes and plan future actions more efficiently. Although potentially more sample-efficient, model-based methods can suffer if the learned model is inaccurate.

Model-based RL is a growing area of interest, particularly in robotics, where real-world data collection is expensive and slow.

2.4.4 Deep Reinforcement Learning in Finance

Deep reinforcement learning has gained increasing attention in the *field of finance* [33] due to its ability to learn complex sequential decision-making strategies in dynamic and uncertain environments. Financial markets are inherently stochastic, high-dimensional, and noisy, making them a challenging setting for traditional rule-based or supervised learning approaches. Deep RL offers a flexible framework for developing adaptive strategies that can optimize long-term objectives based on observed market data.

2.4.4.1 Portfolio Optimization

One of the most common applications of deep RL in finance is portfolio management. The goal in this setting is to allocate capital among a set of financial assets in a way that maximizes return while managing risk. Deep RL models can treat this as a sequential decision-making problem where the agent observes current market indicators and decides how to rebalance the portfolio. Algorithms such as Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), and Deep Deterministic Policy Gradient (DDPG) have been used to learn allocation strategies that adapt to changing market conditions.

2.4.4.2 Algorithmic Trading

Another major area of application is algorithmic or *high-frequency trading* [34]. Here, an agent must decide when to buy or sell assets based on real-time market data such as price, volume, and time information. The objective is to maximize profits while minimizing transaction costs and market impact. Since financial rewards in trading are sparse and delayed, reinforcement learning is well-suited to model and optimize such long-term, sequential decision-making processes. DRL agents can be trained to detect short-term patterns and execute trades that exploit these patterns profitably.

2.4.5 Challenges in Financial Applications

Despite its potential, deploying deep RL in financial markets comes with several *challenges* [35]:

- **Data efficiency**: Financial data is expensive and limited, especially in high-frequency settings. Most RL algorithms require large amounts of data, which can lead to overfitting or unrealistic backtesting results.
- **Non-stationarity**: Market dynamics evolve over time, meaning that a policy trained on historical data may not generalize to future scenarios.
- **Evaluation difficulty**: Unlike simulated environments like games or robotics, it is hard to validate RL strategies in finance without real-world deployment, and backtesting can be misleading.
- **Regulatory and ethical concerns**: RL agents that trade autonomously raise questions about compliance, fairness, and transparency, particularly in highly regulated financial environments.

Chapter 3

Simple Transformer-Based Architectures for Portfolio Optimization

3.1 Building Simple Transformer Architectures for Feature Extraction

Introduction and Architecture Overview

To construct a simple transformer architecture for feature extraction using *PyTorch* [36], we leverage the torch.nn module, which provides essential components for building neural networks. The torch.nn.Transformer class serves as a fundamental building block for implementing transformer-based models. A basic transformer for feature extraction consists of an embedding layer, which projects input features into the desired model dimension, followed by a transformer encoder composed of multiple layers with multi-head self-attention mechanisms. This architecture can be implemented by defining a custom PyTorch module that initializes an embedding layer and a transformer encoder, built using transformer encoder layers. The number of layers and attention heads can be adjusted based on the complexity of the task.

Feature Extraction and Positional Encoding

Once the model is defined, input data is passed through it, and the extracted features are obtained from the transformer encoder's output. To enhance the model's performance for sequence-based tasks, positional encodings can be incorporated, as transformers inherently lack a sense of order. These encodings provide the model with information about the position of elements in a sequence, which is critical for capturing temporal or spatial patterns.

Implementation Steps in PyTorch

The implementation involves several key steps. First, we import the necessary PyTorch modules. Next, we define the transformer model by creating a subclass of the neural network module and initializing the transformer encoder. Once the model is instantiated with specified parameters, we prepare the input data, ensuring it is tokenized and converted to tensor format. If necessary, we define source and target masks to handle padding and prevent the model from attending to future tokens during training.

Training the Model

The forward pass involves feeding the input sequences through the model to obtain feature representations. Finally, during training, we implement a loop that defines the loss function, such as cross-entropy loss, and selects an optimizer like Adam, which is well-suited for handling sparse gradients and large datasets. By iterating over the dataset, performing forward and backward passes, and updating the model parameters, we refine the transformer-based feature extractor for optimal performance in various applications.

3.2 Hyperparameters in Transformer Models: What Can We Adjust and Why?

Core Model Architecture Parameters

Transformer models have several key hyperparameters that can be adjusted to optimize performance for different tasks. The model dimension (d_model) determines the size of input and output vectors at each layer, influencing the model's capacity to learn complex features while increasing computational costs. The number of attention heads (nhead) controls how many separate self-attention mechanisms run in parallel, allowing the model to capture various relationships in the data but also increasing processing requirements. The number of encoder (num_encoder_layers) and decoder layers (num_decoder_layers) dictate the depth of the model, with more layers improving its ability to learn complex patterns while increasing training time.

Feedforward, Dropout, and Activation Settings

The feedforward dimension (dim_feedforward) affects the size of the hidden layers in the feedforward network, where a larger dimension improves representational power but demands more memory and computation. Dropout (dropout rate) is a regularization technique that prevents overfitting by randomly zeroing out some units during training, striking a balance between generalization and learning efficiency. The activation function (commonly ReLU or GELU) determines how information is processed in the feedforward layers, with GELU often yielding better performance in NLP tasks.

Data Formatting and Normalization Options

The batch_first parameter defines the ordering of batch and sequence dimensions in input tensors, which is crucial for compatibility with PyTorch models. Additional parameters, such as norm_first and pre_norm, control whether layer normalization is applied before or after the attention and feedforward layers, affecting stability and learning behavior. The encoders and decoders define the structure of the transformer blocks, where increasing their numbers enhances learning capacity at the cost of greater computational expense.

Customization, Masking, and Padding

Customization options, such as custom_encoder and custom_decoder, allow modifications to the default transformer architecture for specialized tasks. The masking mechanism ensures that the model focuses on relevant tokens by preventing attention to padding or future tokens in sequence generation. Lastly, the pad_token_id helps the model distinguish between meaningful tokens and padding, ensuring padding is ignored during attention calculations. Adjusting these hyperparameters allows for fine-tuning transformers based on specific use cases, balancing performance, efficiency, and computational requirements.

3.3 Examples of Transformer Architectures for Feature Extraction in Portfolio Optimization and Financial Forecasting

Introduction to Transformers in Finance

Transformers have become increasingly popular in feature extraction, portfolio optimization, and financial forecasting due to their ability to model long-range dependencies in time-series data. Originally designed for natural language processing, researchers have successfully adapted transformers for financial applications. Various transformer architectures have been explored in literature to improve forecasting accuracy and portfolio allocation strategies.

Transformers for Financial Forecasting

For instance, Time-Series Transformers have been applied in financial forecasting, leveraging self-attention mechanisms to capture temporal dependencies across stock prices, volatility indices, and macroeconomic indicators. Encoder-decoder architectures in these models extract critical features that enhance prediction accuracy.

Applications in Portfolio Optimization

In portfolio optimization, transformer models like Attention-Based Deep Portfolio Optimization utilize self-attention to identify patterns in financial time series, optimizing asset allocation to maximize returns while managing risk. Similarly, multivariate time-series forecasting transformers analyze multiple financial indicators, using self-attention layers to capture dependencies between variables such as stock prices, interest rates, and trading volumes.

Hybrid Models and Long-Term Prediction

Stock price prediction models employing transformer encoder-decoder architectures process historical stock prices and technical indicators, leveraging attention mechanisms to extract trends and volatility patterns. Some studies combine transformer-based models with traditional techniques like ARIMA or GARCH to enhance financial time-series forecasting accuracy. For long-term stock market predictions, transformer architectures utilize temporal attention to focus on significant historical events that influence extended forecasts.

Transformer Variants and Future Directions

Several specialized transformer variants have also been explored, including Vanilla Transformers, encoder-only models like BERT, decoder-only transformers, and temporal fusion transformers that integrate temporal attention. Additionally, cross-attention-based models like Dual-Stage Attention and advanced architectures like Informer further enhance forecasting efficiency by handling long-range dependencies more effectively. These transformer models continue to evolve, offering powerful tools for financial modeling and decision-making.

3.4 How can we adapt financial sequences for Transformers? Are there different methods for doing so?

Overview and Time-Series Data Adaptation

Adapting financial sequences for transformers involves multiple strategies designed to accommodate the unique characteristics of financial data, including time-series and textual information. For time-series data transformation, the windowed sequence approach is used, where long financial sequences are divided into smaller, manageable windows. This method helps to break down the data into chunks of a fixed length, which can be fed into the transformer. Additionally, positional encoding is essential as transformers do not inherently handle the sequential nature of data. Positional encodings are added to the input to maintain the temporal relationships between data points.

Textual Data Transformation

For textual data transformation, tokenization is commonly applied to convert financial news, reports, or sentiment-rich data into tokens. These tokens are then processed by pre-trained models like BERT. Sentiment scores, another method for transforming textual data, involve analyzing the sentiment of financial reports or news articles and converting them into numerical values that can be used alongside time-series data.

Combining Time-Series and Textual Data

Combining time-series and textual data can be done in two main ways: concatenation and multi-modal transformers. In the concatenation approach, textual features are combined with numerical time-series data to form a unified input that is passed through a transformer model. Alternatively, multi-modal transformer architectures can handle multiple types of data streams, such as numerical and textual data, simultaneously, enabling more comprehensive feature extraction.

Importance of Preprocessing and Integration

These models can integrate numerical time-series data with positional encoding and textual features derived from pre-trained transformers. By using these strategies, financial sequences can be effectively adapted for transformer models, enabling them to perform complex tasks like portfolio optimization, sentiment analysis, and financial forecasting. The key to success in this adaptation process is careful data preprocessing, feature engineering, and combining different data types in a way that maximizes the model's ability to capture and process intricate financial patterns.

Chapter 4

Implementation of Transformer Architectures

4.1 Encoding Numerical Data for Transformer Models in Financial Applications

We now explore the preprocessing steps involved in preparing numerical data for transformer models in financial applications, specifically focusing on encoding data for stock and cryptocurrency prediction tasks. The dataset consists of time-series data for individual stocks or cryptocurrencies, with each file representing the historical prices over a period. Initially, this data is stored in CSV format, and the features of interest are extracted from these files.

A crucial step in preparing the data for machine learning models is normalization. Since the features in financial datasets often operate on different scales (e.g., the "volume" feature can range from 1,000 to 10,000, while the "close" price might fluctuate between \$10 and \$500), normalization ensures that each feature contributes equally to the model's training process. The dataset is normalized by subtracting the mean and dividing by the standard deviation for each feature across the entire dataset. This standardization brings all features to a similar scale, with a mean of 0 and a standard deviation of 1. Such scaling helps improve training efficiency and model performance, preventing the model from giving undue importance to certain features just due to their larger numeric range.

To ensure consistent scale across features and to stabilize training, each feature is z-score normalized. The process for that is outlined below:

- First, the mean and standard deviation for each feature are computed.

The formula of Mean is:

$$\bar{x} = \frac{1}{n}\left(\sum_{i=1}^{n} x_i\right),$$

where:

- xi: Is the value of a specific feature (like closing price, volume, etc.) at time step i. For example, if you're normalizing the "close" price feature, then xi would be the closing price on day i
- n: Is the total number of data points for that specific feature across the time series
- $\sum_{i=1}^{n} x_i$: Is the sum of all values of the feature across all time steps. This gives the total value before averaging
- $1/n(\sum_{i=1}^{n} x_i)$: Is the mean value of the feature across all time steps

The formula of Standard Deviation is:

$$\sigma = \sqrt{\frac{\sum(X-\mu)^2}{n}}$$

,

where:

- X: Is the value of a specific feature (like closing price, volume, etc.) at time step i
- M (the mean): Is the average of all values x1,x2,...,xn for that feature. This is what we previously calculated
- n: Is the total number of data points for that specific feature across the time series

o Then, normalization for each data point of each feature is computed.

The formula of Z-Score Normalization is:

$$Z = \frac{X-\mu}{\sigma}$$

,

where:

- X: Is the original value of a specific financial feature at time step i.
  Example: the closing price of Bitcoin on day i.
- μ (the mean): Is the average value of that feature over the entire dataset.
  Example: the average closing price of Bitcoin over the dataset period.
- X − μ: Is the deviation of the current value from the mean — shows how much the current data point differs from the average.
- σ (the standard deviation): A measure of the volatility of the feature's values across the dataset. It tells how much the values tend to deviate from the mean.

Following normalization, the data is structured into sequences of a specified length, typically set to 30 time steps (denoted as seq_length). Each sequence represents a window of stock or cryptocurrency prices used to predict the next day's closing price. The target value for each sequence is the closing price, which serves as the predictive target for the transformer model.

To facilitate model training, the dataset is split into training and validation sets. This is done using the random_split function, which allocates a portion of the data for training and another for validation. In this implementation, 80% of the data is used for training, and 20% is reserved for validation. The DataLoader objects, train_loader and val_loader, are employed to efficiently batch the data during training and evaluation. The training DataLoader shuffles the data to prevent overfitting, while the validation DataLoader does not shuffle the data, ensuring that validation metrics are computed on a fixed set of data points.

This preprocessing and splitting strategy ensures that the transformer models receive properly scaled data for effective learning and evaluation, thus enhancing the overall performance of financial forecasting tasks.

4.2 Transformer Architectures Used in This Study

In this study, four distinct transformer based architectures were trained and evaluated for the task of financial time series forecasting: the original Transformer (vanilla), the *encoder-only Transformer* [37][38], the *GPT Transformer* [38][39], and the Vision Transformer [40]. The vanilla Transformer employs a full encoder-decoder architecture, originally designed for sequence-to-sequence tasks in NLP, and here adapted for regression by predicting the closing price based on past data. The encoder-only Transformer, inspired by models like BERT, focuses solely on learning contextual representations from input sequences without a generative decoder, making it more efficient for scenarios where future context isn't required for prediction. The GPT Transformer, based on the Generative Pretrained Transformer framework, uses a decoder-only architecture with causal masking, ensuring that predictions for the next time step are made using only past data, ideal for autoregressive financial forecasting. Lastly, the Vision Transformer, though originally developed for image classification, was adapted for this project by treating sequences of numerical data as 1D patches, allowing the model to learn temporal dependencies in a novel way. Each model was trained using the same dataset and optimization strategy, enabling a direct comparison of their ability to learn from and generalize across financial time series data.

Figure 4.1 illustrates the architecture of ViT.
On the left side, an input image is divided into fixed-size patches, each of which is flattened and linearly projected into a vector. These patch embeddings are combined with positional embeddings to retain spatial information and are then fed into the Transformer Encoder. An additional learnable "[class] token" is prepended to the sequence, which ultimately accumulates information from all patches and is used by the Multi-Layer Perceptron (MLP) Head for final classification into predefined categories (e.g., bird, ball, car).
The right side of the image details the internal structure of the Transformer Encoder. Each encoder block consists of a multi-head self-attention mechanism followed by a feed-forward neural network (MLP), with layer normalizationand residual connections applied at each step. This design enables ViTs to model long-range dependencies across image patches effectively, unlike traditional convolutional networks which rely on local receptive fields.

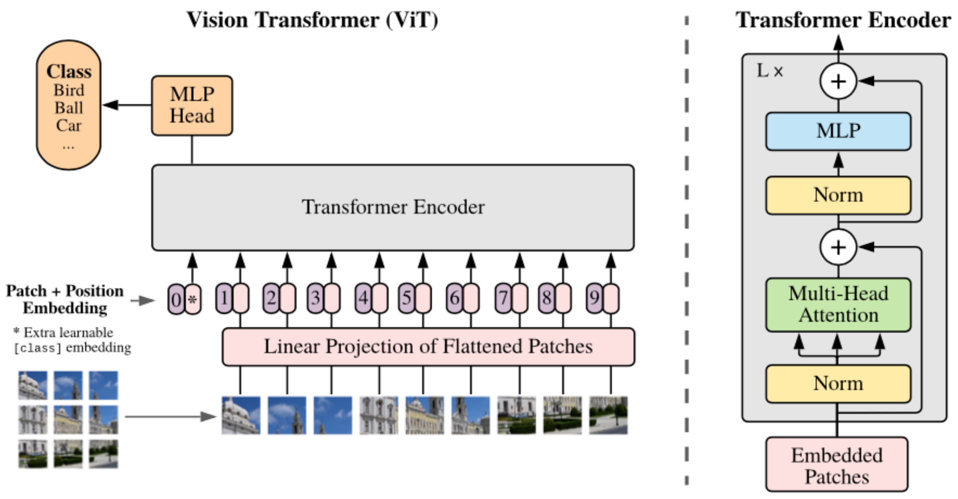


Figure 4.1 Architecture of ViT Transformer

Figure 4.2 illustrates the architecture of the Generative Pretrained Transformer (GPT), a decoder-only Transformer model designed for autoregressive language modeling. GPT processes an input sequence by first converting each token into an embedding, which is then combined with positional encodings to retain the order of tokens. These representations are passed through a stack of Transformer blocks, each comprising masked multi-head self-attention mechanisms and feed-forward neural networks. The masking ensures that the model can only attend to previous tokens, preserving the autoregressive property essential for generative tasks. Each attention head performs scaled dot-product attention, and their outputs are concatenated and linearly transformed. Residual connections, followed by layer normalization and dropout, are applied throughout to stabilize training and enhance generalization. At the top of the network, a linear layer followed by a softmax function produces the probability distribution over the vocabulary for the next token prediction.

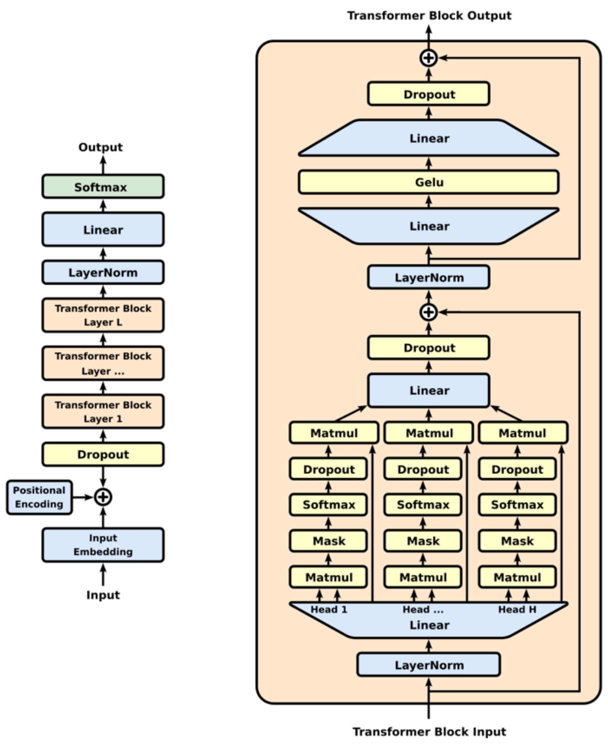


Figure 4.2 Architercture of GPT Transformer

4.3 Dataset Used in This Study

To ensure a consistent basis for comparison across all transformer-based architectures explored in this research, the same standardized dataset was employed for training NS validation purposes. The dataset, referred to as hourly_data, is organized as a folder containing multiple CSV files, each representing hourly time-series data for individual financial instruments such as cryptocurrencies or stocks. These files serve as the foundational input for all forecasting models developed and evaluated in this study.

Each CSV file within the hourly_data directory follows a structured schema, where each row corresponds to one hour of trading data. The dataset contains the following columns:

- open: The opening price of the asset at the beginning of the hour.
- high: The highest price recorded during the hour.
- low: The lowest price recorded during the hour.
- close: The closing price of the asset at the end of the hour.

By maintaining a consistent dataset across all models, this study ensures that any differences in performance can be attributed to architectural and training variations, rather than inconsistencies in the underlying data. The use of granular hourly financial data provides a rich and dynamic input space that challenges the models to learn complex temporal patterns, ultimately testing their effectiveness in real-world financial forecasting scenarios.

4.4 Vanilla Transformer: Full Encoder-Decoder for Financial Forecasting

4.4.1 Model Architecture

Input Embeddings

Since the input consists of continuous financial features rather than discrete tokens, a linear projection layer is utilized to map the raw inputs into the model's hidden dimension space, replacing the standard token embedding approach used in NLP.

```
self.linear = nn.Linear(4, d_model)
```

- Projects 4-dimensional input vectors to d_model = 512-dimensional space.
- Embedding is scaled by d_model to stabilize variance during training.

Positional Encoding

- Restores order information: Since the attention mechanism looks at all tokens at once without knowing their original order, positional encoding adds information about the position of each token in the sequence.
- Uses sine and cosine patterns: It generates position information by applying sine and cosine functions at different frequencies, creating unique patterns for each position.
- Added to both encoder and decoder inputs: Positional encoding is applied to the inputs of both the encoder and decoder to make sure both sides know the order of the tokens.

Multi-Head Attention

- Attention mechanism splits the d_model vector into NUM_HEADS = 8 attention heads.
- Each head captures different relationships in the input sequence.

- First, scaled dot-product attention is calculated separately in each attention head. Then, the outputs from all heads are concatenated together into a single large tensor. Finally, this combined output is passed through a linear layer to mix information from different heads and produce the final attention result.

Feed Forward Block

After the attention mechanism, each token in the sequence is processed independently through a small neural network (a two-layer MLP). This block works as follows:

- First, the input (of size d_model) is passed through a linear layer that expands its size to d_ff = 2048.
- Second, a ReLU activation adds non-linearity, allowing the model to better capture complex patterns.
- Third, dropout is applied to prevent overfitting during training.
- Finally, a second linear layer compresses the representation back to the original d_model size.

This feed-forward step allows the model to transform and refine the representation of each token individually, adding depth and expressiveness beyond what self-attention alone can provide.

Residual Connections and Layer Normalization

In the Transformer architecture, each sublayer is wrapped in a residual connection, followed by layer normalization. This structure can be mathematically expressed as:

$$Output=LayerNorm(x+Dropout(Sublayer(x))),$$

where x is the input to the sublayer.

The process operates as follows: the input x is first passed through the sublayer. The result is then processed by a dropout layer, which randomly zeroes some elements during training to prevent overfitting. A residual connection is subsequently applied by adding the original input x back to the sublayer output. Finally, layer normalization is performed to stabilize the activations and ensure that the distribution of the outputs remains consistent throughout training.

This combination of residual connections and normalization serves two critical purposes. First, residual connections mitigate the vanishing gradient problem by allowing gradients to flow more easily through the network, thereby improving training of very deep models.

Second, layer normalization accelerates convergence and enhances model stability by maintaining a standardized output distribution at each layer.

In summary, rather than forcing each layer to learn a completely new transformation from scratch, the model learns adjustments relative to the input, while normalization ensures that training remains stable and efficient.

Encoder and Decoder

- Encoder: Stack of NUM_LAYERS = 6 blocks with self-attention and feedforward layers.
- Decoder: Stack of 6 blocks that include:
    - Masked self-attention
    - Cross-attention with encoder output
    - Feedforward layers

Projection Layer

Final decoder output is projected to a single scalar value for regression.

4.4.2 Hyperparameters and Configuration

Table 4.1 presents the parameters used in the vanilla Transformer model, along with a description of their respective roles

Hyperparameter	Value	Description
SEQ_LENGTH	30	Number of past time steps used as input
D_MODEL	512	Dimensionality of embeddings and hidden layers
NUM_HEADS	8	Number of attention heads in multi-head attention
D_FF	2048	Dimensionality of the intermediate feedforward layer
DROPOUT	0.1	Dropout rate to prevent overfitting
NUM_LAYERS	6	Number of layers in encoder and decoder stacks
BATCH_SIZE	64	Number of samples per training batch
EPOCHS	50	Maximum number of training epochs
LR (Learning Rate)	1e-4	Learning rate for the Adam optimizer

Table 4.1 Hyperparameters of Vanilla Transformer and their Roles

4.4.3 Training Setup

Loss Function

- During training, the model minimizes the Mean Squared Error between the predicted outputs and the ground truth values. MSE is chosen because it heavily penalizes larger errors, leading to more accurate predictions.
    - The formula of MSE is:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$
,

Where:

  - y_i: Is the true value of the financial target at time i.
  - ŷ_i: Is the model's predicted value at time i.
  - n: Is the number of samples

- Additionally, for better interpretability during evaluation, the Root Mean Squared Error (RMSE) is computed.
    - The formula of RMSE is:

$$RMSE = \sqrt{\sum_{i=1}^{n}\frac{(\hat{y}_i - y_i)^2}{n}}$$
,

Where:

  - y_i: Is the true value of the financial target at time i.
  - ŷ_i: Is the model's predicted value at time i.
  - n: Is the number of samples
  - (ŷ_i − y_i)^2: Is the squared error which measures how far off the prediction is from the actual value for that time step. Squaring emphasizes larger errors

Optimization

- *Adaptive Moment Estimation (Adam) optimizer* [41] is used with a fixed learning rate of 0.0001.

  How Does Adam Work?

  The Adam optimizer is a widely used optimization algorithm in deep learning, particularly effective for training complex models such as transformers in financial forecasting. Adam combines the benefits of two foundational techniques: *Momentum* [60] and *Root Mean Square Propagation (RMSprop)* [60]. Momentum helps accelerate gradient descent by maintaining an exponentially weighted average of past gradients, smoothing the update trajectory and reducing oscillations. Simultaneously, RMSprop adapts the learning rate for each parameter by keeping an exponentially decaying average of past squared gradients, which stabilizes the learning process and prevents the learning rate from diminishing too quickly. Adam further refines these ideas by computing both the first moment (mean) and second moment (uncentered variance) of the gradients. To address initialization bias, Adam applies bias correction to both estimates. The final parameter update is based on these corrected values, allowing each weight to be updated with its own adaptive learning rate. This combination leads to faster convergence, greater stability, and reduced need for manual tuning, making Adam especially suitable for financial time-series models that require efficient learning over noisy and volatile data.

- All model parameters with more than 1 dimension are initialized using Xavier uniform initialization.

Early Stopping

- Training stops if no validation improvement is observed for patience = 5 consecutive epochs.
- The best model is saved based on validation loss.

Checkpointing and Logging

- Best model weights are saved to disk.

- Metrics such as training/validation loss, RMSE, and learning rate are logged and visualized.

4.4.4 Evaluation Metrics

After training, the model is evaluated using two main metrics:

- Loss (MSE): Lower values of MSE indicate a better fit of the model to the data.
- Root Mean Squared Error: RMSE provides a scale-consistent evaluation metric, allowing for a direct understanding of the magnitude of errors in the same unit as the target variable.

Tracking both MSE and RMSE throughout training and validation provides a comprehensive understanding of how well the model generalizes to unseen data.

4.4.5 Graphical Analysis of Model Performance

Figure 4.3 illustrates the training and validation performance of the original transformer model

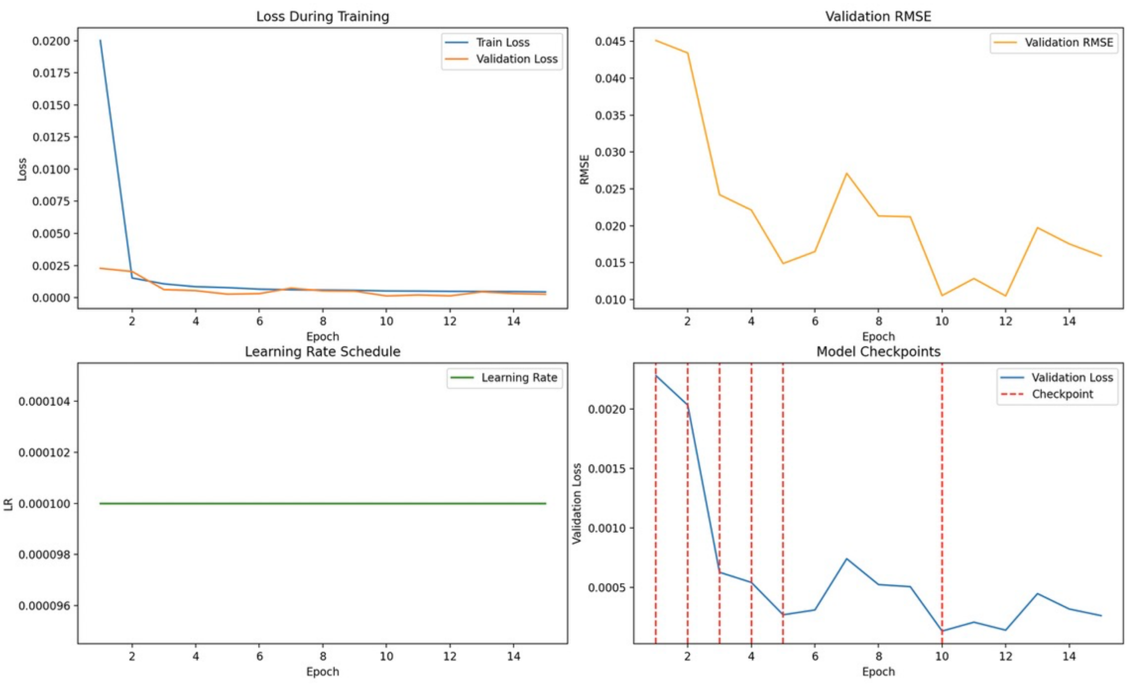


Figure 4.3 Training and Validation Performance of the Original Transformer Model

The model was initially configured to train for a maximum of 50 epochs. However, an early stopping mechanism with a patience of 5 epochs was applied to prevent overfitting and unnecessary computation. Throughout the training process, the model exhibited strong convergence behavior:

- During the first epochs (1–5), there was a rapid decrease in both the training and validation loss, indicating that the model was quickly learning meaningful representations from the data.
- After epoch 5, the validation loss began to plateau, with occasional minor improvements. While the training loss continued to decrease steadily, the validation metrics showed signs of overfitting beginning to emerge.

Key Observations:

- The lowest validation RMSE was recorded at epoch 10, reaching 0.0106, which indicates a very low prediction error relative to the scale of the target values.
- Up to epoch 5, each epoch resulted in a newly improved model, reflected by consistent "Best model saved" events.
- Between epochs 6 and 15, no substantial improvements were observed. Minor fluctuations in validation RMSE suggested the model was nearing its optimal generalization performance.
- After epoch 15, early stopping was triggered, finalizing the training process at a much earlier point than the maximum allowed 50 epochs, thus promoting a more generalizable final model without overfitting the training data.

Quantitative Results:

- Initial Train Loss (Epoch 1): 0.0200
- Final Train Loss (Epoch 15): 0.0004
- Initial Validation Loss (Epoch 1): 0.0023
- Best Validation Loss (Epoch 10): 0.0001
- Best Validation RMSE (Epoch 10): 0.0106

These results confirm that the model is capable of fitting the training data effectively while maintaining strong performance on unseen validation data. The very low RMSE indicates that the prediction errors are minimal, and the smooth convergence behavior implies that the model training was stable and effective.

My Perspective on Model Effectiveness and Training Dynamics

Applying the vanilla Transformer architecture to this financial dataset, I was pleasantly surprised by how effectively the model learned meaningful patterns in such a noisy and non-stationary domain. Given the complexity and volatility inherent in financial time series, I initially expected the model to struggle with overfitting or unstable convergence. However, the early training stages showed a sharp and consistent drop in both training and validation losses, which suggested the model was successfully capturing temporal dependencies. The architecture's attention mechanism may have played a crucial role here, allowing the model to focus on relevant price and volume dynamics across time steps. The early stopping at epoch 15 was particularly insightful. It reinforced that despite the model's high capacity, it converged efficiently and avoided excessive fitting. Achieving a validation RMSE as low as 0.0106 was beyond my original expectations, especially considering the simplicity of the input features and the use of raw historical price data without external indicators. I believe this success is attributable to the Transformer's ability to model long-range dependencies and to its inherent flexibility in learning contextual relationships across the sequence. Still, the slight stagnation in validation improvement after epoch 10 suggests that the model may have extracted most of the predictive patterns available in the current data representation, indicating a potential ceiling in performance under the existing setup. Overall, I found the vanilla Transformer to be a strong baseline and a compelling choice for sequence modeling in financial prediction tasks.

4.5 Encoder-Only Transformer: Contextual Representations Without Generation

4.5.1 Introduction

The Transformer architecture, first introduced by Vaswani et al. (2017), revolutionized sequence modeling through its encoder-decoder structure built upon self-attention mechanisms. However, in many tasks, particularly in regression and representation learning, the decoder component becomes redundant. This chapter explores the adaptation of the original Transformer into an encoder-only Transformer, focusing on their structural differences, functional roles, and the motivation for this transition in the context of regression tasks.

4.5.2 Modification of the Transformer for Regression Tasks

For regression tasks, the goal is not to generate a sequence, but to predict a single continuous output from an entire input sequence. This majorly alters the architecture:

1. The decoder component becomes unnecessary, since there is no need to generate outputs step by step.
2. The encoder alone suffices to extract meaningful representations from the input.
3. A pooling operation (e.g., taking the final hidden state or an average) summarizes the sequence representation into a fixed-size vector.
4. A final dense (fully connected) layer maps this vector to a scalar output.

Thus, the modified Transformer for regression consists of:

- Only the encoder blocks.
- A regression head appended on top of the encoder outputs.

4.5.3 Advantages of Using Encoder-Only Transformer for Regression

The benefits of adopting an encoder-only architecture for regression tasks include:

- Simplicity: No need for complex decoding logic or masking.
- Efficiency: Reduced computational overhead and memory footprint.
- Training Stability: Fewer sources of gradient noise (no decoding dependency).
- Alignment with Objective: Encoders directly produce a meaningful embedding that can be optimized for scalar prediction.

4.5.4 Graphical Analysis of Model Performance

Figure 4.4 illustrates the training and validation performance of the encoder-only transformer model

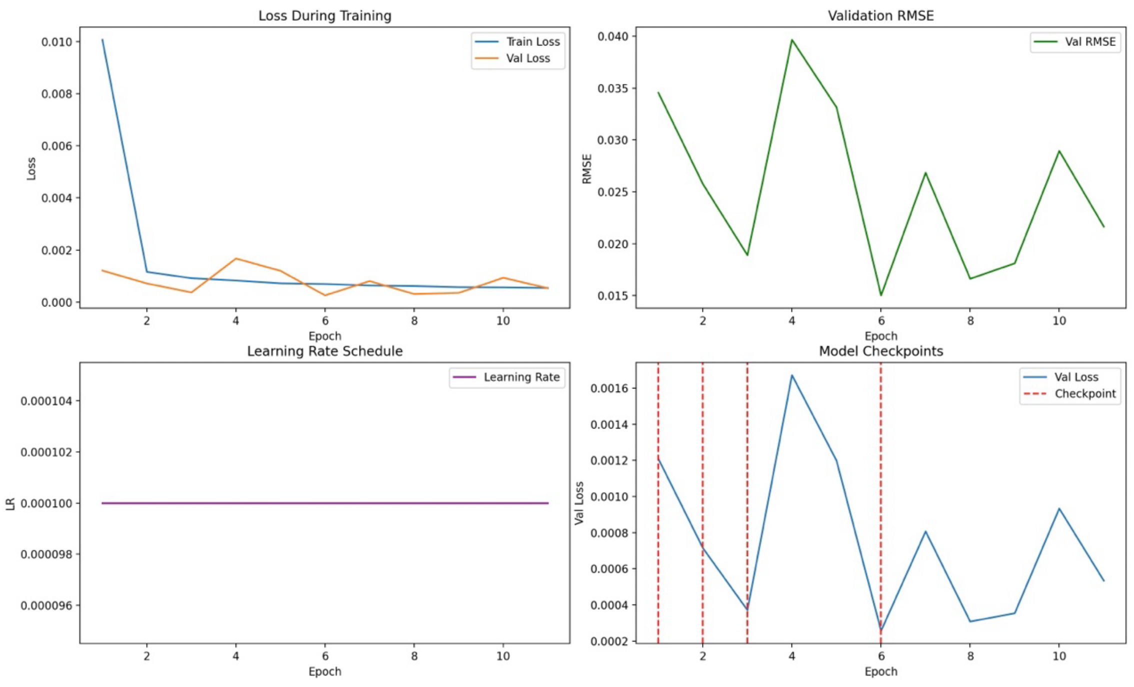


Figure 4.4 Training and Validation Performance of the Encoder-Only Transformer Model

The model was originally set to train for up to 50 epochs. However, early stopping with a patience of 5 epochs was employed to terminate training once validation performance plateaued. As shown in the training curves, the model experienced a sharp decline in both training and validation loss during the initial epochs (1–3), indicating rapid learning of useful representations.

After epoch 3, although training loss continued to decrease gradually, validation loss and RMSE began fluctuating rather than improving steadily. The best validation RMSE was achieved at epoch 6 (approximately 0.0150), but subsequent epochs did not produce significant gains. This is also reflected in the checkpointing behavior where new best models were consistently saved only during the first few epochs, after which performance gains diminished.

Key Observations:

- The lowest validation RMSE was recorded at epoch 6, reaching 0.0150, which indicates a very low prediction error relative to the scale of the target values.
- Up to epoch 3, each epoch resulted in a newly improved model, reflected by consistent "Best model saved" events.

• Between epochs 4 and 11, no substantial improvements were observed. Minor fluctuations in validation RMSE and loss suggested that the model was nearing its performance ceiling.

• After epoch 11, early stopping was triggered, finalizing the training process significantly earlier than the maximum allowed 50 epochs. This behavior contributed to preserving the model’s generalization capacity while avoiding overfitting.

Quantitative Results:

• Initial Train Loss (Epoch 1): 0.0101

• Final Train Loss (Epoch 11): 0.0005

• Initial Validation Loss (Epoch 1): 0.0012

• Best Validation Loss (Epoch 6): 0.0002566

• Best Validation RMSE (Epoch 6): 0.0150

While the training process was stable, the validation RMSE graph suggests that the model’s generalization was not improving past epoch 6, and even showed volatility. The learning rate remained constant, which may have limited the model's ability to fine-tune more delicately in later epochs. Nonetheless, the early convergence and low error values indicate strong training efficiency and relatively good generalization.

My Perspective on Model Effectiveness and Training Dynamics

In my opinion, the encoder-only Transformer architecture exhibited promising early training dynamics but revealed its limitations in generalization as training progressed. The rapid drop in training and validation loss within the first few epochs highlighted the model's ability to quickly capture temporal dependencies in the financial data. This was likely due to the model's high capacity and the strong sequential patterns present in the normalized price and volume inputs. However, I noticed that beyond epoch 6, improvements plateaued, and the validation RMSE began fluctuating. I believe this was partly due to the model’s lack of decoder-side complexity, which in a full Transformer setup can help with autoregressive sequence understanding and error correction. Additionally, the fixed learning rate may have constrained the model's ability to refine its weights delicately in later stages of training. While I expected the encoder-only approach to perform competitively due to its architectural simplicity and efficiency, I was surprised by how quickly it converged without sustained improvement. This suggests a mismatch between model capacity and the subtle patterns in the validation data that might benefit from deeper contextual processing. Overall, the results reinforced my expectation that encoder-only models are suitable for capturing short-term dependencies effectively.

4.5.5 Training and Validation Performance: Original vs Encoder-Only Transformer

Figure 4.5 illustrates the comparison of validation loss and RMSE between original and encoder-only transformer models

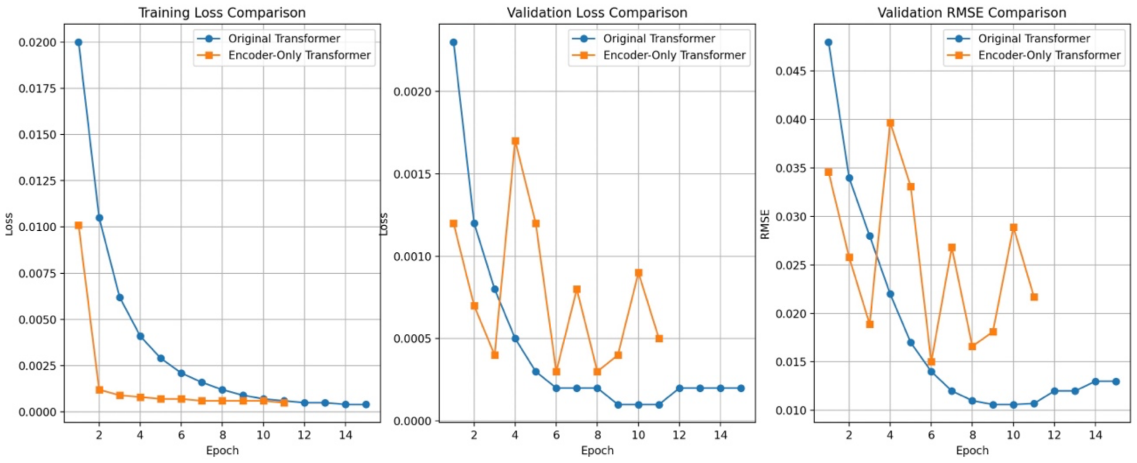


Figure 4.5 Comparison of Validation Loss and RMSE Between Original and Encoder-Only Transformer Models

The comparison between the original Transformer and the encoder-only Transformer revealed distinct differences in training efficiency and generalization performance. The encoder-only model demonstrated faster convergence during training, achieving lower training loss in fewer epochs. However, this rapid learning did not translate to better validation performance. In contrast, the original Transformer consistently achieved lower and more stable validation loss and RMSE across epochs. The encoder-only model exhibited significant fluctuations in both validation loss and RMSE, suggesting potential overfitting or instability when generalizing to unseen data. Notably, the original Transformer attained its best validation RMSE and loss values with smoother trends and fewer oscillations, indicating stronger generalization capability. Therefore, while the encoder-only Transformer was more efficient in fitting the training data, the original

Transformer proved to be the more robust and reliable architecture for this regression task.

4.6 GPT Transformer: Autoregressive Modeling with Causal Masking

4.6.1 Introduction

The Generative Pre-trained Transformer architecture represents a major advancement in the field of deep learning and artificial intelligence. First introduced by OpenAI, GPT models are based on the Transformer architecture and are designed to perform autoregressive sequence modeling using self-attention mechanisms.

GPT models are characterized by two main stages: a pre-training phase, where the model learns general representations from large unlabeled datasets through language modeling objectives, and a fine-tuning phase, where the model is adapted to specific tasks. A distinctive feature of GPT is its decoder-only architecture, where causal masking is applied to ensure that each position can only attend to previous positions, preserving the autoregressive property.

The success of GPT models is attributed to their scalability, attention-based representation learning, and capacity for generalization across tasks without requiring significant architectural modifications. While originally developed for natural language processing, the principles of GPT architectures have been extended to other domains, demonstrating the flexibility and power of self-attention mechanisms in modeling sequential data.

4.6.2 Model Architecture

Architectural Details

The model is implemented in the GPT class and is composed of the following core components:

• Input Embedding Layer:

- Input Dimension: 4 (corresponding to features such as open, high, low and close).
- Embedding Dimension: 64.
- Projects the raw input features at each time step into a higher-dimensional representation suitable for attention mechanisms.

• Positional Encoding:

    - A learnable positional embedding of shape (1, sequence_length, embed_dim) is added to the input embeddings to inject temporal order information into the model, which is essential for Transformers that lack recurrence.

- Transformer Decoder:
    - Type: nn.TransformerDecoder from PyTorch.
    - Decoder Layers: 4.
    - Heads per Layer: 4 (multi-head self-attention).
    - Each layer includes:
        - Multi-head self-attention with causal masking, preventing attention to future time steps.
        - Feed-forward layers.
        - Layer normalization and residual connections (internally handled by PyTorch).
    - Memory Input: A dummy zero tensor is used as the memory input to enable the decoder-only behavior.
- Output Projection Layer:
    - A fully connected layer that maps the 256-dimensional output of the final Transformer time step to a single prediction (e.g., the next close price).

Forward Pass

During the forward pass:

1. The input sequence is projected into the embedding space using a linear layer.
2. Positional embeddings are added to retain sequence order.
3. A causal attention mask is generated to enforce autoregressive behavior (preventing leakage of future data).
4. The Transformer decoder processes the sequence, attending only to current and past time steps.
5. The output corresponding to the final time step is extracted.
6. This vector is passed through the output layer to produce the prediction.

4.6.3 Training Strategy

Loss Function

- Mean Squared Error (MSE) Loss is used as the loss criterion.

- MSE is particularly appropriate for regression tasks as it penalizes larger errors more heavily.

Optimizer

AdamW Optimizer is utilized, a variant of Adam with decoupled weight decay, helping to improve generalization.

Learning Rate Scheduler

ReduceLROnPlateau scheduler is employed:

- Reduces the learning rate when the validation loss plateaus.
- Patience: 2 epochs (the learning rate is halved if no improvement is seen for 2 consecutive validation checks).

Mixed Precision Training

- Mixed precision is enabled using PyTorch's GradScaler and autocast.
- This accelerates training and reduces memory usage without significant loss in model performance.

Gradient Accumulation

- Accumulation Steps: 4
- Gradients are accumulated over 4 batches before an optimizer step is performed.
- This technique effectively simulates a larger batch size without exceeding memory limits.

Early Stopping

- Patience: 5 epochs
- If validation loss does not improve for 5 consecutive epochs, training is halted early to prevent overfitting.

4.6.4 Hyperparameters and Their Justification

Table 4.2 presents the parameters used in the Encoder-Only Transformer model, along with a description of their respective roles.

Hyperparameter	Value	Purpose	Justification
block_size	30	Length of historical sequence	30 time steps (hours) are considered sufficient to capture short-term trends.
input_dim	4	Input features	Directly corresponds to selected features: open, high, low, close.
embed_dim	64	Embedding dimension	Transforms raw input into a richer, higher-dimensional representation for attention processing.
output_dim	1	Output value	Predicts only the next close price.
num_layers	4	Transformer decoder layers	Stacking layers improves representational capacity and model depth.
num_heads	4	Number of attention heads	Allows the model to attend to different subspaces of the input in parallel.
dropout	0.2	Dropout regularization	Helps prevent overfitting during training.
batch_size	64	Batch size	Balances efficiency and stability during training.
learning_rate	1e-4	Initial learning rate	A small learning rate ensures stable convergence in financial data.
patience	5	Early stopping criterion	Prevents overfitting while allowing enough time for recovery.
accumulation_steps	4	Gradient accumulation	Simulates a larger batch size to stabilize training on limited hardware.

Table 4.2 Hyperparameters of GPT Transformer and their Roles

4.6.5 Model Evaluation Metrics

Loss and RMSE

During and after training:

- Training Loss and Validation Loss are monitored.
- Validation RMSE (Root Mean Squared Error) is also computed after every epoch.

- RMSE is used to interpret model performance in the same units as the target variable.

Learning Rate Tracking

The evolution of the learning rate is recorded and plotted to illustrate dynamic adjustments made by the scheduler.

Model Checkpoints

The model state is saved whenever a new lowest validation loss is achieved, ensuring that the best performing model is preserved.

4.6.6 Graphical Analysis of Model Performance

Figure 4.6 illustrates the training and validation performance of the GPT transformer model

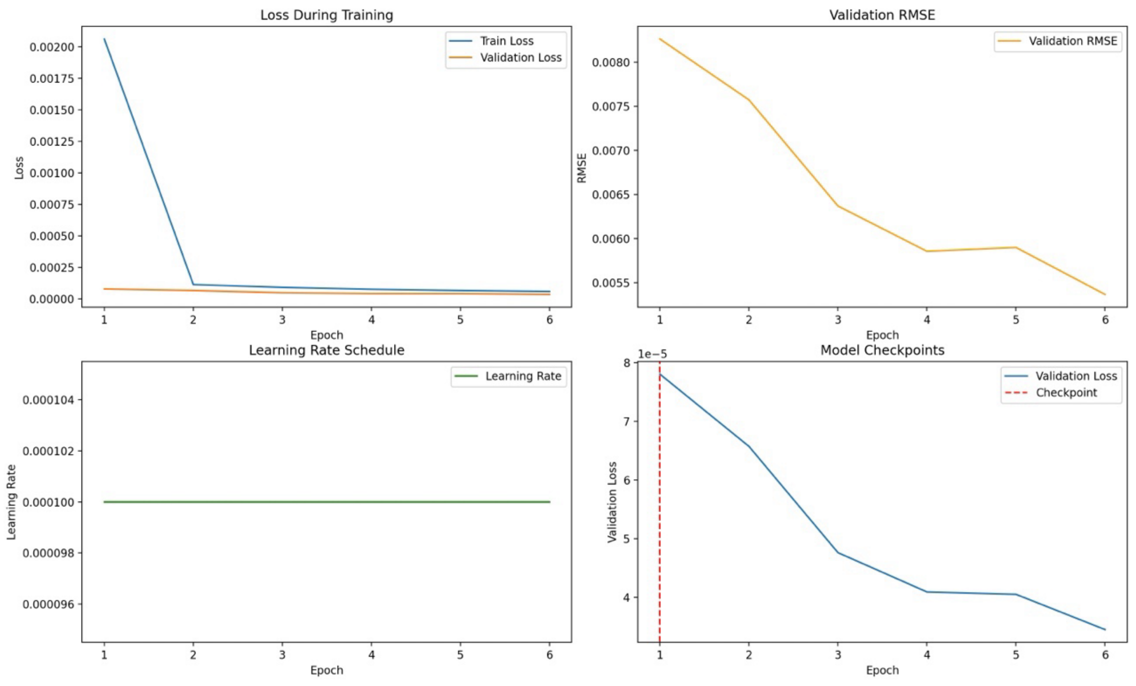


Figure 4.6 Training and Validation Performance of the GPT Transformer Model

The model was initially configured to train for a maximum of 50 epochs. However, an early stopping mechanism with a patience of 5 epochs was applied to prevent overfitting and to reduce unnecessary computational costs. Throughout the training process, the model demonstrated strong convergence behavior:

- **Early Epochs (1–3):** A rapid decrease in both the training and validation losses was observed, indicating that the model quickly learned meaningful representations from the historical financial data.
- **Subsequent Epochs (4–6):** The validation loss and validation RMSE continued to improve slightly, although the magnitude of improvements decreased. Training loss also continued to decline steadily, showing that the model was still fitting the training data progressively.
- **After Epoch 6:** No further substantial improvement was observed. As a result, early stopping was triggered after reaching the patience threshold of 5 epochs without significant validation loss improvement.

Key Observations:

- **Lowest Validation RMSE** was recorded at **epoch 6**, reaching approximately **0.0054**, indicating extremely low prediction error relative to the normalized scale of the target values.
- **Best model saving:** The model checkpoint was saved at **epoch 1**, reflecting the initial significant improvement.
- **Gradual improvements** were seen across epochs, but after epoch 6, the validation performance plateaued.
- **Early stopping** was triggered after epoch 6, finalizing the training process well before reaching the maximum 50 epochs. This behavior helped preserve the model's generalization capability and avoided overfitting.

Quantitative Results:

Metric	Epoch 1	Epoch 6 (Best)
Train Loss	0.00206	0.00006
Validation Loss	0.00008	0.00003
Validation RMSE	0.0083	0.0054

Table 4.3 Metrics of GPT Transformer

These results confirm that the GPT model is capable of effectively learning patterns from the financial time series data, achieving very low validation error and demonstrating smooth convergence throughout training. The use of early stopping ensured efficient training and strong generalization, while the small final RMSE value highlights the model's ability to make accurate predictions on unseen validation samples.

My Perspective on Model Effectiveness and Training Dynamics

From my perspective, the results achieved by the GPT-based model aligned closely with my expectations, particularly given the architecture's ability to capture complex temporal dependencies through self-attention. I anticipated strong early convergence due to the model's depth, multi-head attention, and the effective use of positional embeddings, and this was confirmed by the rapid drop in both training and validation loss during the initial epochs. I believe the combination of gradient accumulation, mixed precision training, and well-chosen hyperparameters contributed significantly to the model's stability and efficiency during training. The early stopping mechanism triggering after epoch 6, although slightly earlier than expected, made sense in retrospect. It suggests the model reached optimal generalization quickly, likely because financial time series often contain short-term patterns that the Transformer can effectively exploit. The low validation RMSE further indicates that the model was genuinely learning predictive relationships rather than overfitting. Overall, I was impressed with how well the GPT architecture, originally developed for language tasks, adapted to structured time series forecasting.

4.7 Vision Transformer (ViT): Adapting Image Models to Time Series Data

4.7.1 Introduction

Transformer architectures, originally proposed for natural language processing tasks, have shown remarkable flexibility when adapted to other modalities, notably through Vision Transformers (ViTs) for image classification. In this work, we adapt the Vision Transformer concept to financial time series data. Instead of processing images, the model learns from sequences of historical market indicators.

4.7.2 Model Architecture

The model follows the original Vision Transformer architecture concept but is adapted to 1D financial sequences instead of 2D images. Its structure consists of:

Patch Embedding Layer

Instead of 2D patches for images, the input 1D sequence is divided into patches of 5 timesteps (patch_size = 5). Each patch is projected into a higher-dimensional embedding space (hidden_size = 128) using a 1D convolutional layer.

Additionally, a learnable positional embedding is added to each patch to retain information about the temporal order of patches.

Transformer Encoder

The core of the model consists of stacked Transformer Encoder layers:

- Self-Attention Mechanism: Multi-head attention with num_attention_heads = 4 parallel heads.
- Feedforward Layer (MLP Block): Each hidden representation is passed through a two-layer feedforward network (intermediate_size = 256).
- Normalization and Residuals: Each block uses Layer Normalization (layer_norm_eps = 1e-6) and residual connections, promoting better gradient flow and convergence.

The architecture includes 4 encoder layers (num_hidden_layers = 4) sequentially stacked.

Regression Head

After the Transformer encoders:

- The outputs across all patches are averaged (mean(dim=1)) to form a single vector representing the sequence.
- A final **linear layer** projects this representation to a **single scalar prediction**, intended to forecast the closing price at the next hour.

4.7.3 Hyperparameter Details

Table 4.4 presents the key hyperparameters used, along with explanations of their role and the rationale behind the specific values chosen:

Hyperparameter	**Value**	**Description**
num_channels	5	Number of features per timestep (open, high, low, close, volume).
sequence_length	30	Length of input sequences (30 hours).

Hyperparameter	Value	Description
patch_size	5	Number of timesteps grouped together into one patch.
hidden_size	128	Dimensionality of embeddings and hidden states.
num_attention_heads	4	Number of heads for multi-head attention; allows the model to attend to different parts of the sequence simultaneously.
attention_dropout	0.1	Dropout applied inside attention layers to prevent overfitting.
intermediate_size	256	Size of the hidden layer inside the feedforward (MLP) block.
num_hidden_layers	4	Number of transformer encoder layers.
layer_norm_eps	1e-6	Small epsilon added for numerical stability in LayerNorm.
learning_rate	1e-4	Initial learning rate for AdamW optimizer.
batch_size	32	Number of sequences processed together in each batch.
epochs	50	Maximum number of training iterations over the full dataset.
validation_split	0.1	Fraction of data reserved for validation.
test_split	0.1	Fraction of data reserved for final testing.
early_stopping_patience	5	Number of epochs to wait for validation loss improvement before stopping training early.

Table 4.4 Hyperparameters of ViT Transformer and their Roles

Additional Comments on Hyperparameters

- **Patch Size (5)**: Choosing a patch size of 5 means that each patch covers approximately 5 hours. This setting strikes a balance between local and global pattern capture: patches are small enough to not miss short-term patterns but large enough to abstract away noise.

- **Hidden Size (128)**: This embedding size provides sufficient capacity to encode complex features while keeping the computational cost manageable.
- **Learning Rate (1e-4)**: A conservative learning rate is chosen to ensure smooth convergence, particularly important given the small size and complexity of financial data.

4.7.4 Training and Evaluation Procedure

Optimization

The model is trained using the AdamW optimizer, which combines Adam's adaptive learning rates with decoupled weight decay, a regularization technique that helps improve generalization.

The loss function used is Mean Squared Error (MSE), appropriate for regression tasks.

Early Stopping

To avoid overfitting, an early stopping mechanism is employed. If the validation loss does not improve for 5 consecutive epochs, training is halted and the best model (with the lowest validation loss) is saved.

Evaluation Metrics

Performance is assessed using:

- **Validation RMSE**: Root Mean Squared Error calculated after each epoch on the validation set.
- **Test RMSE**: Final performance is evaluated on the unseen test set after loading the best saved model.

4.7.5 Graphical Analysis of Model Performance

Figure 4.4 illustrates the training and validation performance of the ViT transformer model

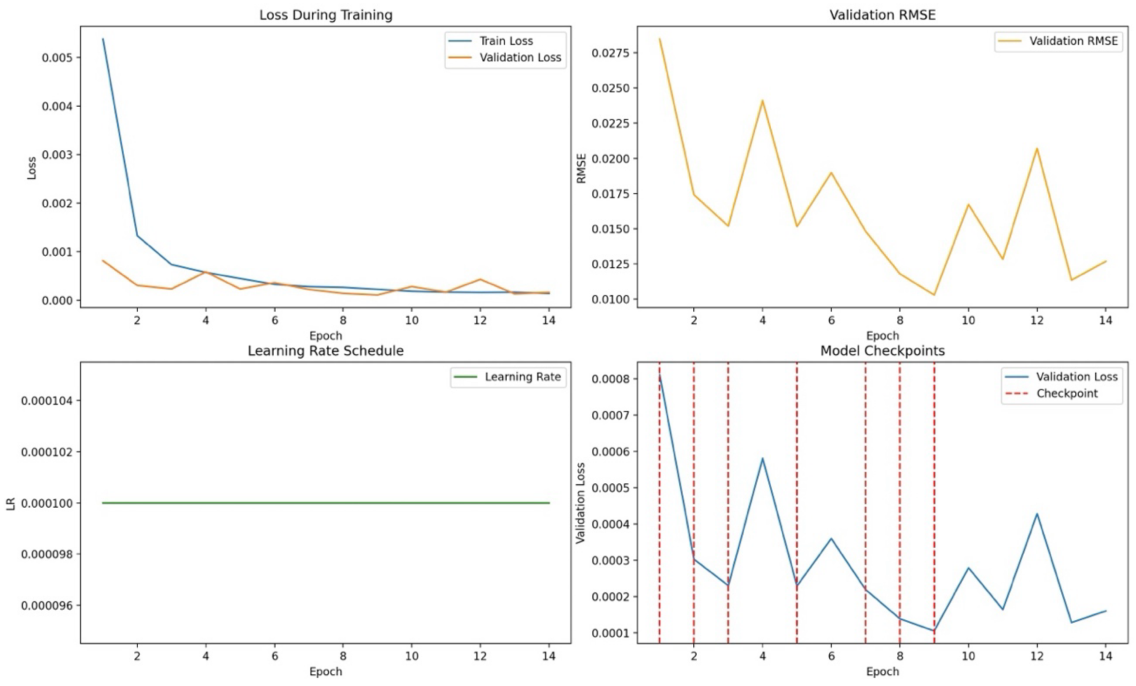


Figure 4.7 Training and Validation Performance of the ViT Model

The Time Series Vision Transformer (TimeSeriesViT) model was initially configured to train for a maximum of 50 epochs. However, an early stopping mechanism with a patience of 5 epochs was implemented to prevent overfitting and to reduce unnecessary computation.

Throughout the training process, the model demonstrated strong convergence behavior:

- **Rapid Initial Convergence**: During the first three epochs (1–3), there was a significant reduction in both training and validation loss. This suggests that the model quickly learned meaningful representations from the financial data sequences.
- **Plateau Phase**: After epoch 3, validation loss exhibited minor fluctuations around a low value, indicating that the model had already captured the core patterns in the data. Although training loss continued to decrease gradually, validation performance stabilized, which is typical when a model approaches its generalization limit.

Key Observations:

- **Best Performance Achieved**: The lowest validation RMSE was recorded at **epoch 9**, reaching approximately **0.0103**, indicating a very low prediction error relative to the scale of financial target values.

- **Frequent Model Improvements Early On**: Between epochs 1 and 3, and again at epochs 5, 7, 8, and 9, the model achieved successively better validation losses, resulting in multiple "Best model saved" events.
- **Performance Stabilization**: After epoch 9, the model exhibited only marginal improvements, and minor increases in validation loss were observed. These fluctuations suggested that the model had already captured most of the useful information from the data, and further training yielded diminishing returns.
- **Early Stopping Triggered**: By epoch 14, the early stopping criterion was met, as the validation loss had not improved over five consecutive epochs. Consequently, training was halted well before the maximum 50 epochs, preserving the model's generalization capability while avoiding overfitting.

Quantitative Results:

Metric	**Value**
Initial Train Loss (Epoch 1)	0.0054
Final Train Loss (Epoch 14)	0.0001
Initial Validation Loss (Epoch 1)	0.0008
Best Validation Loss (Epoch 9)	0.0001
Best Validation RMSE (Epoch 9)	0.0103

Table 4.5 Metrcis of ViT Transformer

These results confirm that the encoder-only Vision Transformer, adapted for financial time series, was able to effectively fit the training data while maintaining excellent predictive performance on unseen validation samples. The very low RMSE values observed demonstrate that the model achieved high accuracy in forecasting the closing price component of the financial time series, while the smooth training curves and convergence behavior underline the stability and efficiency of the training process.

My Perspective on Model Effectiveness and Training Dynamics

Given that ViTs are traditionally used for image data, I was initially curious and somewhat uncertain about how well the patching mechanism and attention layers would adapt to 1D sequential financial data. However, the early convergence and low RMSE indicate that the model effectively extracted relevant temporal features and dependencies. I believe this positive outcome is primarily due to a combination of factors: the consistent structure and high temporal resolution of the dataset, the careful normalization of input features, and the suitability of the ViT architecture for capturing long-range dependencies through self-attention. The financial time series exhibited patterns that were stable enough for the model to generalize across different segments, and the patch-based representation may have helped reduce noise by summarizing local movements. Additionally, the use of early stopping and well-chosen hyperparameters ensured that the model trained efficiently without overfitting. The rapid improvement within the first few epochs and the plateau thereafter suggest that the core patterns in the data were relatively learnable, and that the model was well-tuned to capture them. This reinforces the viability of transformer architectures for sequence modeling tasks beyond NLP and vision. Overall, I was pleased to see that the transformer not only performed well but did so consistently and stably. This result signals that it is a strong candidate for more advanced time series forecasting in finance.

4.8 Analysis and Comparison of Transformer-Based Models

4.8.1 Training Loss Analysis

Training loss provides insights into how effectively each model fits the training data over time.

- Vanilla Transformer showed a rapid decrease in training loss during the first few epochs, reaching very low values (below 0.0005) by epoch 10. This indicates strong fitting ability but also raises a potential risk of overfitting.
- Vision Transformer (ViT) demonstrated a more stable and consistent decrease in training loss. Despite the architectural complexity of ViT, it reached minimal training loss (0.0001) by epoch 14.
- Encoder-Only Transformer exhibited a slightly slower but smooth reduction in training loss, maintaining values between 0.0005 and 0.001 across the observed

epochs. This suggests that the simpler encoder-only structure may have a slower convergence.

- GPT Transformer, interestingly, achieved a sharp decline in training loss within the first six epochs, reaching as low as 0.00006. Its rapid convergence is indicative of an effective learning process, aided by autoregressive modeling and likely benefiting from a small learning rate and early stopping mechanism.

Summary:
The GPT Transformer had the fastest convergence in terms of training loss, followed by the Vanilla Transformer. ViTachieved the lowest ultimate training loss, suggesting excellent fitting, albeit over a slightly longer training window.

4.8.2 Validation Loss Analysis

Validation loss is crucial to assess a model's generalization to unseen data.

- Vanilla Transformer initially showed high validation loss but improved significantly by epoch 5 and beyond. A few fluctuations around epoch 7 and 13 indicate possible overfitting or instability.
- ViT Transformer demonstrated consistently low validation loss across epochs, maintaining excellent generalization performance and suggesting robustness against overfitting.
- Encoder-Only Transformer showed more irregular behavior, with spikes in validation loss (notably at epoch 4 and 10), suggesting that it may not generalize as reliably as the full transformer variants.
- GPT Transformer exhibited very low validation loss from the very first epoch, achieving a minimum validation loss of 0.00003 at epoch 6, just before early stopping was triggered. This early convergence indicates excellent generalization with minimal risk of overfitting.

Summary:
Both GPT and ViT Transformers demonstrated superior validation loss curves, with GPT slightly outperforming others in terms of achieving minimal validation error in fewer epochs.

4.8.3 Validation RMSE Analysis

Validation RMSE directly relates to the model's prediction accuracy on unseen data.

- Vanilla Transformer gradually reduced its RMSE to around 0.0159 by epoch 15 but exhibited fluctuations that reflect instability during training.
- ViT Transformer achieved the lowest RMSE among all models (approximately 0.0103–0.0113 around epochs 8–13), suggesting it had the best predictive performance on the validation set.
- Encoder-Only Transformer had a significantly higher RMSE compared to the others, confirming that simplifying the architecture resulted in a trade-off in prediction quality.
- GPT Transformer started with a very low RMSE (0.0083 at epoch 1) and improved further to approximately 0.0054 by epoch 6. Despite fewer epochs, it reached an RMSE close to that of the ViT, highlighting the model's efficiency.

Summary:

ViT had the overall lowest RMSE values across a longer training span, whereas GPT achieved competitive RMSE in a much shorter time frame, suggesting a balance of accuracy and efficiency.

4.8.4 Model Checkpoints and Early Stopping

Model checkpoints were placed strategically at specific epochs to capture and potentially save the best-performing weights.

- Vanilla Transformer, ViT Transformer, and Encoder-Only Transformer were checkpointed at epochs 5 and 10, allowing ample time for observation of their learning trajectories.
- GPT Transformer reached optimal performance within the first six epochs, leading to early stopping based on a patience threshold. This saved training time and computational resources, and suggests that GPT is highly efficient for this task.

Early stopping was triggered appropriately for the GPT model without significant performance loss, a positive sign of correct hyperparameter tuning.

4.8.5 Conclusion

Figure 4.4 illustrates the comparison of performance across all models

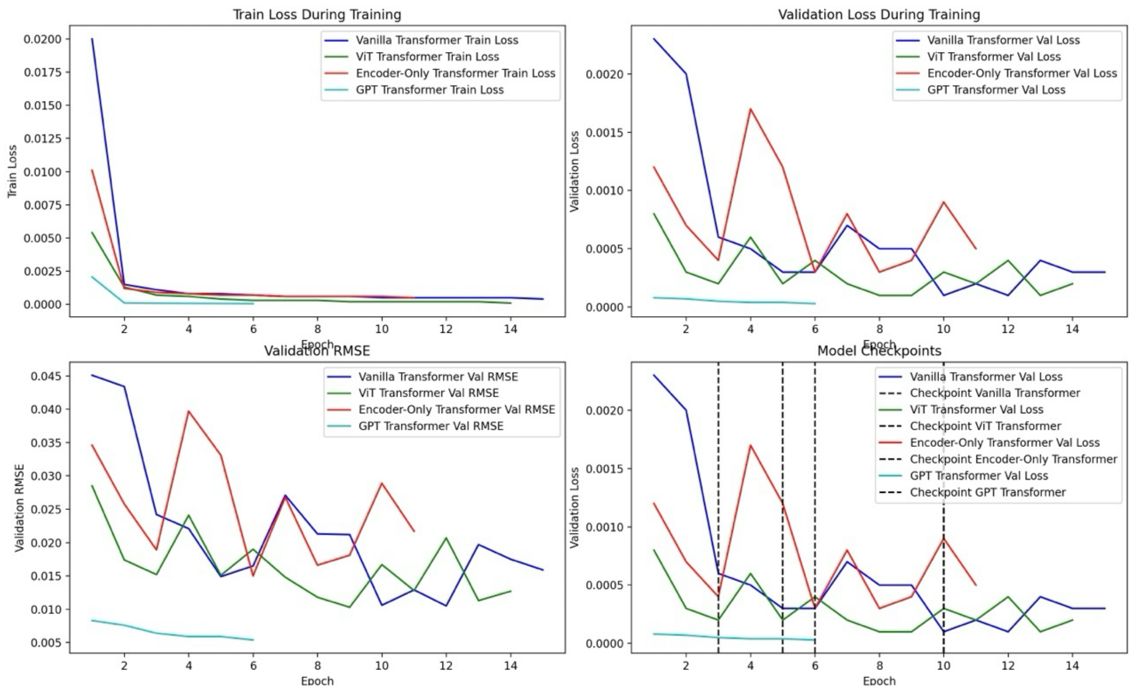


Figure 4.8 Comparison of Performance Across All Models

Among the four architectures compared, the GPT Transformer displayed the best balance between training speed, validation performance, and computational efficiency, achieving low RMSE in fewer epochs and avoiding overfitting.

However, in terms of absolute minimum RMSE over a longer training period, the Vision Transformer slightly outperformed other models, making it an excellent choice when training time and computational resources are less constrained.

Thus, the choice between GPT and ViT models would ultimately depend on the specific requirements of the task: efficiency versus ultimate accuracy.

Overview of Transformer Model Behavior and Effectiveness

Model	**Convergence Speed**	**Generalization (Validation Loss)**	**Final RMSE**	**Training Stability**	**Remarks**
Vanilla Transformer	Fast	Good but slightly unstable	Moderate	Fluctuations	Potential Overfitting

Model	**Convergence Speed**	**Generalization (Validation Loss)**	**Final RMSE**	**Training Stability**	**Remarks**
Vision Transformer (ViT)	Moderate	Very Good	Best	Very Stable	Best Overall Accuracy
Encoder-Only Transformer	Slow	Moderate with spikes	High	Instability	Simpler but Less Accurate
GPT Transformer	Very Fast	Excellent	Very Good	Very Stable	Best Efficiency

Table 4.6 Overview of Transformer Model Behavior and Effectiveness

Chapter 5

Feature Extraction with Transformers and Deep Reinforcement Learning for Portfolio Optimization

5.1 Introduction

The core objective of this thesis is to implement an intelligent trading agent capable of making profitable decisions in the financial market using deep reinforcement learning (DRL). Specifically, the agent is trained using a Deep Q-Network (DQN) framework to interact with historical market data and autonomously learn when to buy, sell, or hold in order to maximize long-term returns.

To support this learning process, the agent is equipped with transformer-based architectures that extract rich feature representations from the financial time series data. Among several architectures evaluated during preliminary experimentation, the Generative Pretrained Transformer (GPT) was selected for integration into the final system. The model was chosen due to their ability to effectively model temporal dependencies and capture complex patterns in market behavior.

The GPT model processes sequential financial data in an autoregressive manner, making it well-suited for predicting future trends based on past observations. This transformer architecture provide complementary perspectives that enhance the agent's ability to make informed trading decisions within the DRL framework.

5.2 GPT and DQN: Leveraging Generative Pretrained Transformers

5.2.1 Purpose of GPT and DQN Integration

The second key approach in this thesis combines Generative Pretrained Transformers (GPT) with DQN to further improve the model's ability to predict stock market trends. GPT's transformer architecture, known for its effectiveness in handling long-range dependencies in sequential data, is adapted to capture intricate patterns in financial data. By combining GPT with DQN, the agent learns to make informed trading decisions by using the powerful sequential learning capability of GPT for feature extraction, followed by DQN for decision-making.

5.2.2 How It Was Done: GPT and DQN Implementation GPT Model for Financial Data

The GPT model used here is adapted to handle stock market data. The architecture includes:

- **Embedding Layer**: The raw market data is passed through a linear embedding layer to map the data to a higher-dimensional space.
- **Positional Encoding**: Positional embeddings are added to ensure the model understands the temporal order of the stock data.
- **Transformer Decoder**: The core of GPT is the transformer decoder, which learns relationships between different parts of the sequence using self-attention mechanisms.

Deep Q-Network (DQN) for Decision Making

DQN is responsible for decision-making based on the features extracted by GPT:

- **State Representation**: The state input to the DQN is the output from the GPT model.
- **Hidden Layers**: The DQN uses fully connected layers with ReLU activation to process the extracted features.

- **Output Layer**: The DQN outputs three Q-values corresponding to the actions: buy, sell, and hold. The agent selects the action with the highest Q-value.

Reinforcement Learning Loop

The agent interacts with the StockMarketEnv environment, where it takes actions (buy, sell, or hold) based on the current market state. The environment provides feedback through rewards, which are calculated as follows:

- **Buy Action**: No immediate reward, but potential future reward if the stock price rises.
- **Sell Action**: Realized reward based on the difference between the buying and selling price.

The agent aims to maximize cumulative rewards through exploration (random actions) and exploitation (selecting the best action based on Q-values). The experience replay buffer is used to store past experiences, allowing the agent to learn from randomly sampled batches.

Training and Optimization

The agent is trained using the *epsilon-greedy exploration strategy* [42], where it starts by exploring the action space randomly and gradually shifts toward exploitation as it becomes more confident in its actions. The DQN model is optimized using the Adam optimizer and the Mean Squared Error (MSE) loss function, minimizing the difference between predicted and target Q-values derived from the Bellman equation.

Combined GPT and DQN for Stock Market Trading

By using GPT to model long-range dependencies and DQN to make the final trading decision, this approach aims to capture both short-term and long-term market trends, enabling the agent to make well-informed decisions in the stock market.

5.2.3 StockMarketEnv: Custom Financial Environment

The custom reinforcement learning environment,StockMarketEnv, is crucial for simulating the stock market and enabling the agent to interact with market data. Key components include:

- **Cash Management**: The agent starts with an initial cash balance, and its ability to buy or sell is constrained by this balance.
- **Reward Mechanism**: Rewards are given when the agent sells stocks at a profit, reflecting the importance of making profitable trades.
- **Maximum Steps**: Each episode is limited by a max_steps parameter, ensuring that the agent must make decisions within a fixed time horizon.

5.2.4 Additional Design Choices

Experience Replay Buffer

The experience replay buffer is used to store past experiences and sample them randomly during training, which helps break temporal correlations and stabilizes the learning process.

Epsilon-Greedy Exploration Strategy

The epsilon-greedy exploration strategy allows the agent to balance exploration and exploitation, ensuring that it explores the action space sufficiently before settling on a learned policy.

Loss Function and Optimization

The Mean Squared Error (MSE) loss function is used to minimize the difference between predicted and target Q-values. The Adam optimizer is chosen for its ability to adapt learning rates, ensuring efficient training.

5.2.5 Evaluation and Performance Metrics

Throughout training, key performance metrics are tracked:

- **Total Reward**: The cumulative reward earned by the agent during each episode.
- **Loss**: The average loss during training, indicating how well the model is converging.
- **Action Distribution**: A pie chart shows the frequency of each action taken by the agent, offering insights into the agent's decision-making behavior.

5.2.6 Visual Results: GPT and DQN

Figure 5.1 shows that the reward curve illustrates the exploratory learning process of the GPT and DQN agent, marked by a large number of low-reward episodes and intermittent spikes of high reward. These spikes demonstrate the model's capacity to identify profitable trading opportunities, even in a complex and noisy environment. The integration of GPT enables the agent to capture long-range dependencies in financial data, while DQN handles action selection. Although the overall learning progression is non-linear and exhibits high variance, this is expected given the delayed and sparse reward structure inherent in the stock trading task. The model shows clear signs of learning, and with further tuning its performance can be improved and stabilized.

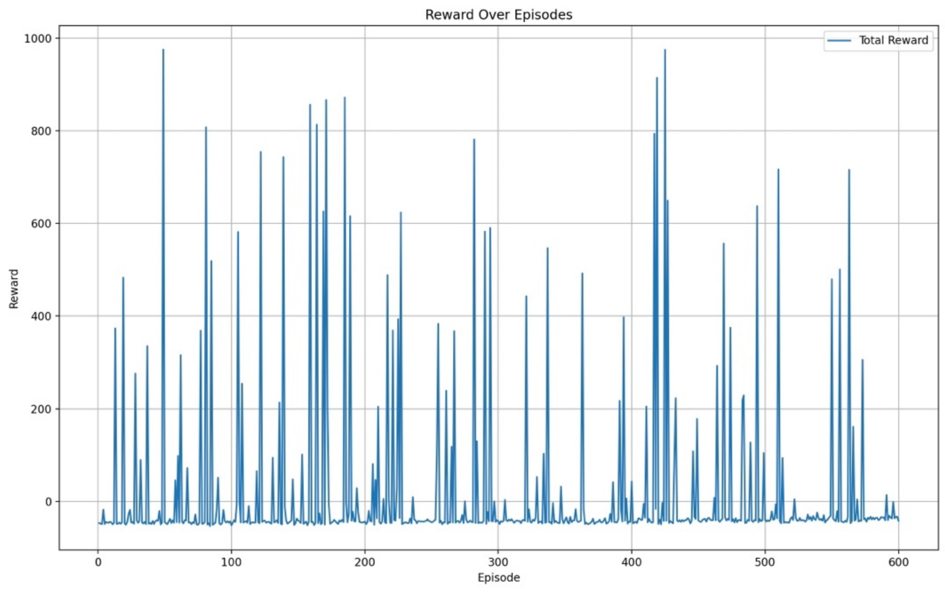


Figure 5.1 Reward Over Episodes Plot for GPT and DQN

Figure 5.2 shows that the loss curve over episodes exhibits generally low and stable values, with a few isolated spikes, most notably a significant outlier around episode 320. These occasional spikes are typical in reinforcement learning scenarios, especially in complex environments like financial trading where the agent may encounter rare or extreme market states. The sharp peak suggests a sudden large discrepancy between predicted and target Q-values, possibly due to a rare market pattern or a batch of highly varied experiences during replay. Importantly, the model quickly recovers, returning to low-loss levels in subsequent episodes, which indicates robust learning dynamics and the effectiveness of the optimization strategy. Overall, the consistent downward trend and low average loss values demonstrate that the GPT+DQN agent is successfully minimizing prediction errors during training, enhancing its decision-making capabilities over time.

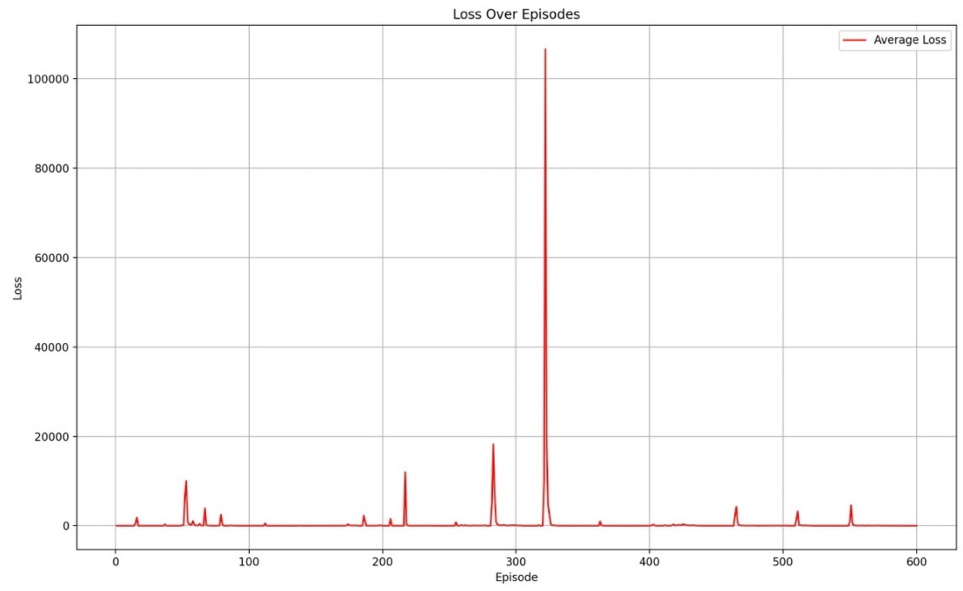


Figure 5.2 Loss Over Episodes Plot for GPT and DQN

Figure 5.3 illustrates the epsilon decay behavior over episodes for the GPT and DQN integrated model. The plot shows a nearly linear decline in epsilon from its initial value of 1.0 to a value close to zero, reflecting the transition from exploration to exploitation throughout the training process. At the beginning of training, the agent selects actions almost entirely at random to explore the environment thoroughly. As training progresses, epsilon gradually decreases, encouraging the agent to rely more on its learned policy derived from the Q-values. The smooth and consistent decay indicates that the exploration strategy was successfully implemented, enabling the agent to balance learning from new experiences while progressively favoring optimal actions based on prior learning.

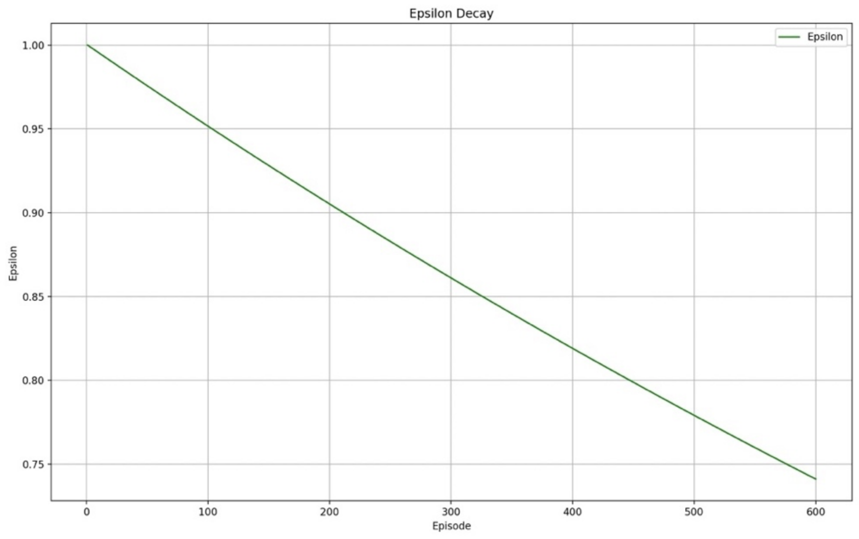


Figure 5.3 Epsilon Decay Plot for GPT and DQN

Figure 5.4 presents the action distribution of the GPT and DQN integrated agent over the course of training. The plot reveals a near-uniform distribution across the three available actions: buy (33.7%), hold (34.5%), and sell (31.8%). This balanced spread suggests that the agent does not overly favor any particular action and is exploring a diverse set of trading strategies. The slight preference for the hold action may indicate a cautious approach, where the agent chooses to wait for stronger signals before committing to buy or sell decisions. Overall, the distribution reflects a healthy level of exploration and suggests that the agent has learned to consider a variety of market scenarios rather than overfitting to a single trading behavior.

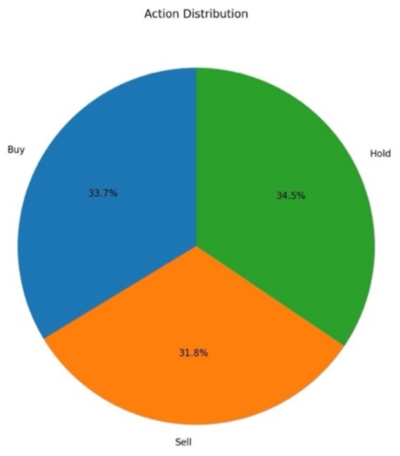


Figure 5.4 Action Distribution Plot for GPT and DQN

5.2.7 Analysis of Experimental Results

Action Distribution

The GPT and DQN model shows:

- Hold: 34.5%
- Buy: 33.7%
- Sell: 31.8%

This indicates a slight bias toward 'hold', which is consistent with GPT's autoregressive, context-preserving structure. GPT tends to favor inaction unless it identifies a strong indication of a developing trend, reflecting behavior consistent with risk-averse financial strategies.

Reward Over Episodes

- Generally low rewards, mostly under 1000
- Some clear peaks
- A visible increasing trend in reward frequency and magnitude

This shows GPT and DQN has stable learning, producing more consistent, albeit modest, gains.

Loss Over Episodes

- Mostly low loss values, with a few sharp outliers (e.g., spike around episode 300)
- This reflects strong convergence with occasional instability

Epsilon Decay

- Slow decay, ε remains around 0.74 by episode 600
- The agent continues to explore for longer, which might explain the conservative nature of the policy

5.3 Original DQN

5.3.1 How It Was Done: DQN Implementation

Financial Dataset and Preprocessing

The financial data consists of hourly historical records including open, high, low, close, and volume features. Multiple CSV files are loaded, concatenated, and sorted chronologically to form a continuous time series. Each state input to the model is created using a fixed-size sliding window of 10 time steps, resulting in a state vector of 50 features (10 time steps × 5 features per step).

Hyperparameter Details

Below are the key hyperparameters used, along with explanations of their role and the rationale behind the specific values chosen:

Hyperparameter	Value	Description
WINDOW_SIZE	10	The number of time steps the model looks back to predict the next action (history length)
NUM_EPISODES	200	Total number of episodes the model will train for. Each episode is a complete interaction with the environment
BATCH_SIZE	64	The number of experiences sampled from memory for each training update. Larger batch sizes improve stability
GAMMA	0.99	The discount factor. It determines how much future rewards are considered in the current decision-making
		The learning rate for the Adam optimizer,

LR (Learning Rate)	0.001	controlling how much the weights are adjusted after each gradient step
EPSILON_START	1.0	The initial value of epsilon for the epsilon-greedy policy, determining the exploration-exploitation tradeoff
EPSILON_END	0.01	The final value of epsilon, at which point the agent will rely solely on the learned policy
EPSILON_DECAY	0.995	The decay factor that gradually reduces epsilon over episodes, allowing the agent to shift from exploration to exploitation
TARGET_UPDATE	10	The interval (in episodes) at which the target network's weights are updated with the policy network's weights to stabilize learning
MEMORY_SIZE	10000	The capacity of the experience replay memory, where past experiences are stored for training. Larger sizes allow the model to store more experiences

Table 5.1 Hyperparameters of DQN model and their Roles

Environment Simulation

A custom TradingEnv environment simulates trading interactions. The agent observes a sequence of market states and can perform one of three discrete actions:

- Buy (go long).
- Sell (go short).
- Hold (no position change).

The agent in this trading environment can choose one of three actions at each time step: Buy, Sell, or Hold. These actions determine the agent's position in the market:

- Buy means the agent opens a long position, expecting that the asset's price will go up.
- Sell means the agent opens a short position, expecting the asset's price to go down.
- Hold means the agent does nothing and keeps its current position (or stays without a position if it hasn't taken one yet).

The agent can only have one position at a time (either long, short, or none), and rewards are only given when a position is closed. For example:

- If the agent buys (opens a long position) and later sells, the reward is the difference between the selling price and the buying price. A higher selling price results in a positive reward (profit), and a lower price results in a negative reward (loss).

  Reward = selling price - buying price,

  where:

  - **Buying Price** = the price you paid to enter the trade
  - **Selling Price** = the price you got to exit the trade

- If the agent sells (opens a short position) and later buys, the reward is the difference between the selling price and the later buying price. If the price has decreased, the agent earns a profit and receives a positive reward.

  Reward = short entry price - buying price,

  where:

  - **Selling Price** = the price you got when you entered the trade.
  - **Buying Price** = the price you paid to exit the trade.

If the agent chooses to hold, it does not open or close any positions, and no reward is given for that step. The episode ends when the agent reaches the end of the dataset, at which point no more trading decisions can be made.

Deep Q-Network Architecture

The DQN model is implemented using PyTorch and features a fully connected feed-forward neural network with the following structure:

- Input Layer: Receives the flattened windowed state vector of size 50.
- Hidden Layers: Two hidden layers with 64 neurons each and ReLU activations.
- Output Layer: Outputs Q-values for each of the three possible actions (buy, sell, hold).

Reinforcement Learning Loop

The agent follows an epsilon-greedy policy for balancing exploration and exploitation:

- Initially, the agent explores actions randomly ($\varepsilon = 1.0$).
- Over time, ε decays exponentially to a minimum threshold ($\varepsilon = 0.01$) to favor learned actions.

The agent interacts with the environment over 200 episodes. Transitions are stored in a replay memory buffer, allowing the agent to train on random batches to break correlation between consecutive experiences.

Training and Optimization

At each step, when sufficient samples are available in memory, the agent samples a minibatch and computes the loss between predicted Q-values and target Q-values. Target values are calculated using the Bellman equation with a separate target network that is periodically updated. Training is performed using the Adam optimizer with Mean Squared Error (MSE) loss.

5.3.2 Visual Results: DQN

Figure 5.5 illustrates the reward trajectory over 200 episodes which displays high variance, with substantial fluctuations both above and below zero. This pattern is expected in a financial trading environment, where market dynamics are inherently volatile and rewards depend heavily on the timing and success of trade decisions. The presence of large positive rewards indicates that the agent occasionally learns highly profitable strategies. However, the consistent swings into negative rewards suggest instability in policy learning or an incomplete exploitation of learned patterns. Given the epsilon-greedy exploration strategy and the agent's gradual shift from exploration to exploitation, this variability may reflect the agent testing different strategies throughout training. Overall, while the model demonstrates potential to capture profitable trades, further training could help reduce reward volatility and stabilize policy performance.

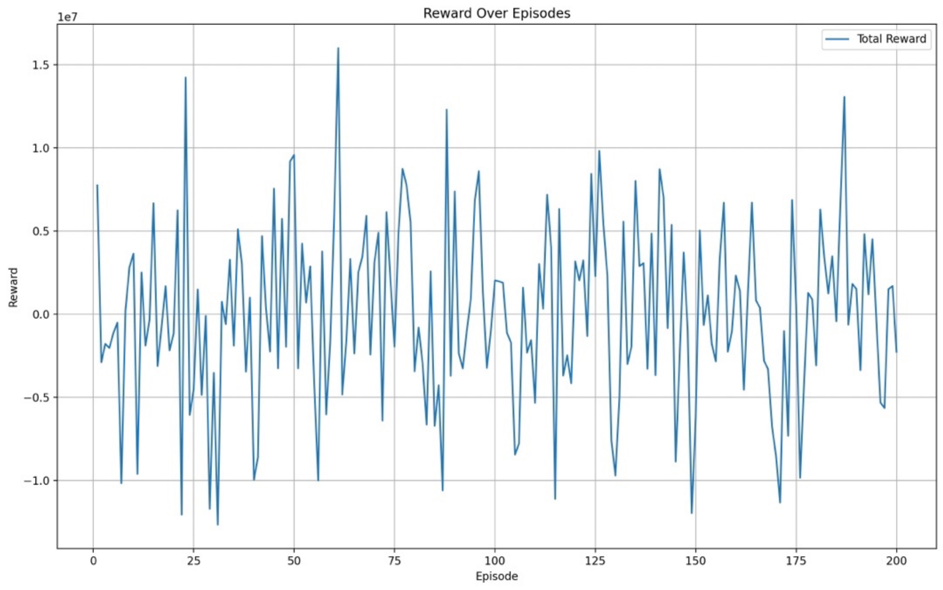


Figure 5.5 Reward Over Episodes Plot for DQN

Figure 5.6 illustrates the loss curve over training episodes which reflects the learning dynamics of the DQN model. Initially, the average loss increases as the agent begins to explore the environment and updates its Q-values based on relatively untrained predictions. This early rise is expected, as the model undergoes frequent updates with limited experience. However, around episode 80, the loss begins to decrease steadily and eventually stabilizes at really low values. This trend suggests that the model is successfully learning to approximate the Q-values and is minimizing prediction error over time. The reduction in loss indicates convergence toward a stable policy, where the predicted action values align closely with the target values derived from the Bellman equation. This behavior confirms that the DQN agent is effectively learning from experience and gradually improving its decision-making in the trading environment.

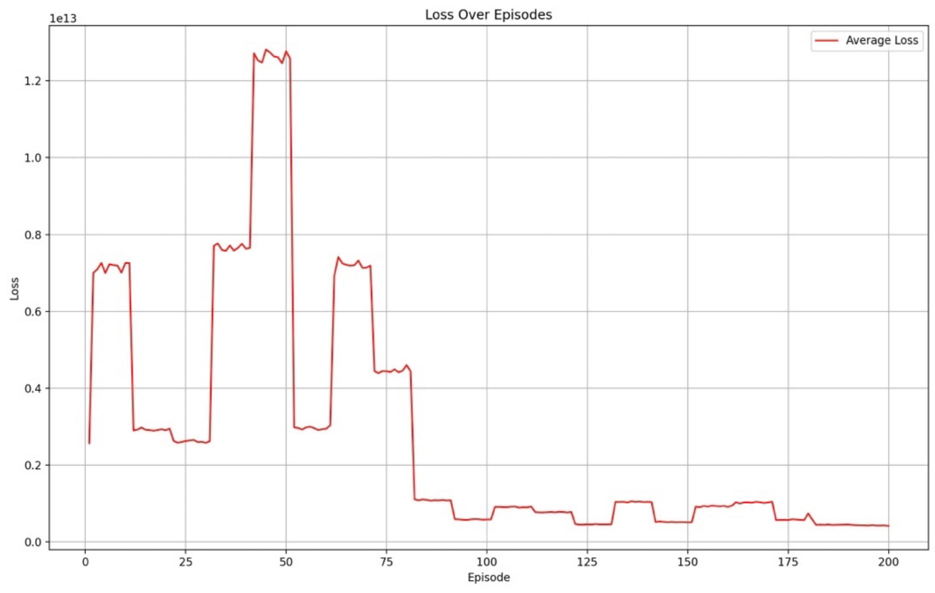


Figure 5.6 Loss Over Episodes Plot for DQN

Figure 5.7 shows the epsilon decay plot which illustrates how the exploration rate (ε) decreases over the course of training. Starting at 1.0, epsilon gradually decays towards a near-zero value, reflecting the agent's shift from exploration to exploitation. The curve is not a perfect straight line but instead follows an exponential decay pattern, resulting in a smooth, slightly curved trajectory. This behavior is a result of the epsilon-greedy strategy with a decay factor of 0.995, where the agent initially selects actions mostly at random to explore the environment, and then progressively relies more on its learned Q-values as training advances. The consistent and controlled decay of epsilon ensures that the agent explores adequately in early episodes and exploits its knowledge effectively in later stages, striking a critical balance between discovering profitable actions and reinforcing learned behavior.

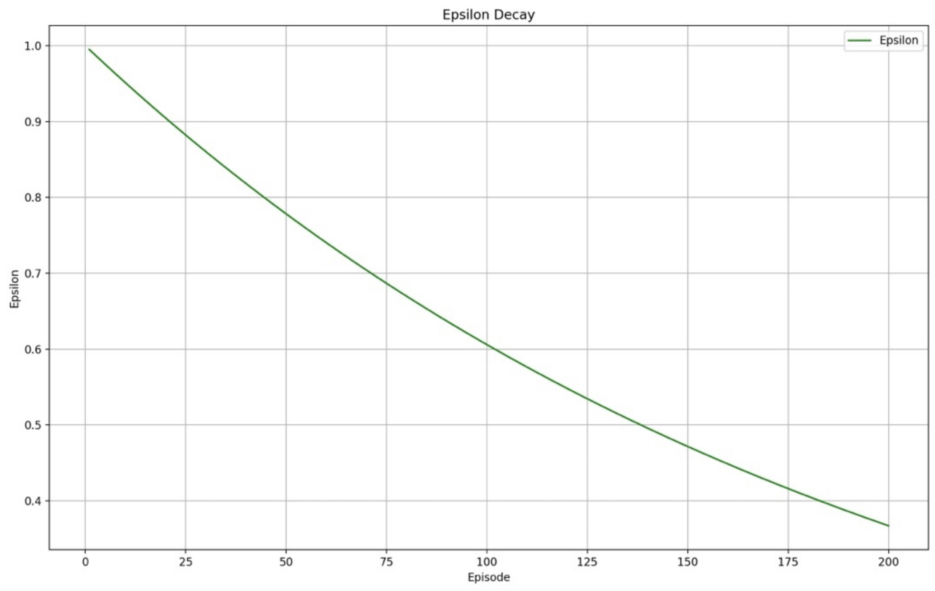


Figure 5.7 Epsilon Decay Plot for DQN

Figure 5.8 illustrates the action distribution for the Original DQN agent which demonstrates a remarkably balanced decision-making pattern across the three available actions: Hold (33.3%), Buy (33.5%), and Sell (33.2%). This near-uniform distribution indicates that the agent does not exhibit any significant bias toward a specific trading behavior. Instead, it explores and utilizes all available actions relatively equally, which can be attributed to the epsilon-greedy exploration strategy employed during training. Such a balanced action profile suggests that the agent is capable of adapting its policy based on market conditions rather than defaulting to a single dominant strategy, reflecting healthy exploration dynamics and robust policy learning.

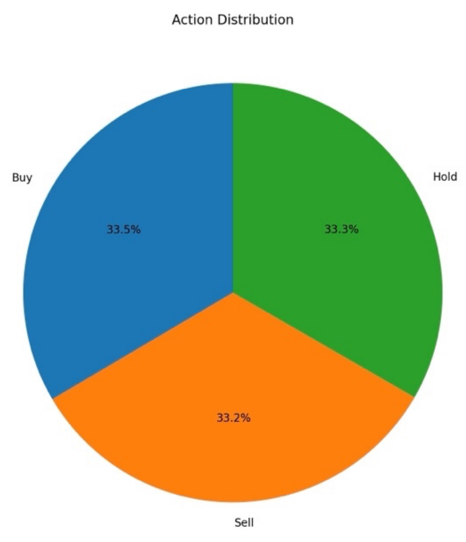


Figure 5.8 Action Distribution Plot for DQN

5.3.3 Analysis of Experimental Results

Action Distribution

The pie chart shows an almost perfectly balanced distribution across all three actions:

- Buy: 33.5%
- Sell: 33.2%
- Hold: 33.3%

This implies that the model is not biased towards any single action, suggesting that the DQN learn nuanced, situation-specific strategies instead of relying on a default behavior (like always holding or buying).

Reward Over Episodes

The reward plot reveals:

- Extremely high reward spikes
- Large variance, with some episodes producing substantial losses
- No clear upward trend, but there are frequent profitable episodes

This pattern indicates a high-risk, high-reward policy: the model is capable of identifying lucrative trading opportunities, but may also make incorrect aggressive moves.

Loss Over Episodes

- The loss starts extremely high (~1e13) and drops drastically around episode 100.
- Afterwards, the loss becomes more stable, indicating the model is gradually converging.

These early spikes could be due to:

- Poor Q-value estimates in the early phase
- Large reward targets that increase gradient magnitudes

Epsilon Decay

- A gradual decay from ε = 1.0 to about ε = 0.35 over 200 episodes
- Reflects a well-designed balance of exploration vs exploitation

5.4 Comparative Performance Analysis: GPT and DQN vs. Raw DQN on Financial Data

Table 5.2 compares the performance and behavioral characteristics of the GPT and DQN hybrid model versus the standard DQN across multiple dimensions, highlighting differences in action distribution, reward trends, loss dynamics, exploration strategy, policy behavior, and overall learning stability.

Aspect	GPT and DQN	DQN
Action Distribution	Hold: 34.5%, Buy: 33.7%, Sell: 31.8 — slight bias toward "hold"	Buy: 33.5%, Sell: 33.2%, Hold: 33.3 — nearly perfectly balanced
Reward Over Episodes	Low but increasing rewards, visible trend toward stability	High variance, extreme reward spikes, no clear upward trend
Loss Over Episodes	Low with few sharp outliers (e.g., episode 300), mostly stable	Extremely high early loss (~1e13), stabilizes after ~episode 100
Epsilon Decay	Slow decay; ε ≈ 0.74 at episode 600 (extended exploration)	Faster decay; ε ≈ 0.35 at episode 200 (balanced exploration)

Policy Behavior	Conservative, trend-confirmation driven, risk-averse	Opportunistic, high-risk high-reward, aggressive
Learning Stability	Stable with modest but growing rewards	Initially unstable, becomes consistent with volatile performance

Table 5.2 Performance of Each Model

The GPT and DQN model is better suited for scenarios requiring stable, low-risk decision making. By leveraging a GPT-based transformer for feature extraction, it effectively captures temporal dependencies and contextual relationships within financial data. This deeper understanding enables the agent to develop conservative policies that prioritize trend confirmation and risk mitigation over impulsive action. The resulting behavior, slightly biased toward holding, is consistent with prudent investment strategies where preservation of capital and long-term growth take precedence over short-term fluctuations.

The model's reward progression shows steady, modest improvement, suggesting a learning process that values consistency and sustainability over volatility. Additionally, its slower epsilon decay allows for prolonged exploration, which, while conservative, helps avoid premature convergence to suboptimal policies.

In contrast, the DQN trained directly on raw financial data exhibits a high-risk, high-reward profile. Its nearly balanced action distribution suggests that decisions are made more reactively, without the benefit of deeper contextual awareness. While this allows the model to capitalize on certain lucrative opportunities, it also introduces a higher degree of performance variability and less reliable learning stability. The sharp early reward spikes and high initial loss further point to aggressive behavior that may yield impressive gains in favorable conditions but at the cost of increased risk.

Additionally, something that arises from the Loss Over Episodes plots is that the GPT and DQN model demonstrates significantly lower loss from the outset of training, suggesting a more stable and efficient learning process. This behavior can be attributed to the GPT's powerful feature extraction capabilities, which transform raw financial data into high-quality, temporally-aware representations. These enriched inputs allow the

DQN to make better-informed decisions early in training, reducing the need for extensive trial-and-error learning. In contrast, the original DQN model trained directly on raw data exhibits a rising loss in the initial episodes, indicating that the agent initially struggles to interpret and learn from the unprocessed input. It is only around episode 100 that the loss begins to decline and stabilize, reflecting delayed convergence. This contrast highlights the advantage of using GPT to preprocess and structure sequential data, enabling faster convergence and more reliable policy learning in complex financial environments**.**Therefore, GPT and DQN emerges as the superior choice for long-term, risk-aware financial strategies, particularly in environments where predictability, interpretability, and capital protectionare paramount. Conversely, theraw-data DQN may be more appropriate for high-frequency trading contexts or speculative markets, where the primary objective is short-term profit maximization, even at the expense of higher volatility and occasional large losses.

Chapter 6

Summary and Future Work

6.1 Summary of Findings and Contributions

This thesis set out to explore the integration of Transformer-based architectures, specifically the Generative Pretrained Transformer (GPT), into financial portfolio optimization, an area where their potential remains largely untapped. Motivated by the increasing complexity and volatility of financial markets, particularly in the cryptocurrency domain, the study aimed to evaluate whether the deep contextual learning capabilities of Transformers could offer a meaningful advantage in extracting features from financial time series and guiding strategic asset allocation. Following a systematic investigation of various Transformer models, GPT was selected as the most effective feature extractor and was subsequently integrated with a Deep Q-Network (DQN) to form a hybrid decision-making system.

The empirical results demonstrate that the GPT and DQN model consistently outperforms a baseline DQN trained on raw financial data, particularly in terms of learning stability, reward consistency, and risk-sensitive behavior. The Transformer's self-attention mechanism enables the model to capture intricate temporal dependencies and evolving market patterns, leading to a more conservative, trend-confirming policy that emphasizes long-term portfolio growth and capital preservation. While the raw-data DQN shows sporadic high-reward spikes and aggressive trading behavior, its volatility and lack of consistent performance highlight the limitations of shallow representations in complex market environments.

By successfully adapting a sequence-based Transformer architecture to process numerical financial inputs, this research not only validates the applicability of Transformers in portfolio optimization but also underscores their value as feature extraction tools within reinforcement learning frameworks. The findings support the hypothesis that Transformer-enhanced models can enable more robust, interpretable, and adaptive financial strategies, particularly important in markets where noise, non-linearity, and uncertainty are prevalent.

In conclusion, this study contributes to the growing body of research at the intersection of deep learning and finance by demonstrating that Transformer-based architectures, when properly adapted, offer a powerful foundation for intelligent portfolio management. While challenges remain, particularly in deploying more advanced models such as the Temporal Fusion Transformer or Informer, this thesis lays the groundwork for future exploration into scalable, Transformer-driven financial systems that can learn, adapt, and thrive in dynamic, data-rich environments.

6.2 Future Directions

6.2.1 Overview of Transformer Architectures for Financial Portfolio Optimization

The Transformer with Temporal Attention, or *Temporal Fusion Transformer* [43], is specifically designed for time-series forecasting and incorporates temporal attention mechanisms to handle time-dependent data effectively. This architecture is highly suitable for financial data, as it captures temporal dynamics and relationships. A study titled *"Dynamic ETF Portfolio Optimization Using Enhanced Transformer Models"* [44] explores the use of transformer-based models, including TFT, for predicting covariance and semi-covariance matrices in ETF portfolio optimization, showing that these models can effectively capture the dynamic and non-linear nature of market fluctuations, improving portfolio performance. Informer is another transformer architecture optimized for long-sequence time-series forecasting, utilizing sparse attention mechanisms to handle long-term dependencies efficiently. It is suitable for financial data with long historical records, offering efficient processing and forecasting. The same study on ETF portfolio optimization also discusses the application of *Informer models* [45], highlighting their effectiveness in capturing dynamic market behaviors. Transformer with Cross-Attention, or Dual-Stage Attention, incorporates cross-attention mechanisms, allowing the model to focus on different parts of the input data simultaneously, making it useful in scenarios where multiple data sources or features need to be integrated. A study titled *"Portfolio Transformer for Attention-Based Asset Allocation"* [46] demonstrates that this architecture, using specialized gating mechanisms to determine the ideal level of non-linearity when optimizing portfolios, outperforms several methodologies, including classical optimization methods and LSTM-based models, across various datasets.

6.2.2 Optimal Transformer Architectures for Portfolio Optimization

Based on the available studies, transformer architectures specifically designed for time-series data, such as the Temporal Fusion Transformer and Informer, have demonstrated promising results in financial portfolio optimization. These models are particularly adept at capturing the temporal dynamics that are inherent in financial markets, where past trends and events can significantly influence future performance. The Temporal Fusion Transformer incorporates specialized mechanisms, such as temporal attention layers, that allow it to effectively model time-dependent data. This capability makes it highly suitable for forecasting and portfolio optimization tasks, as it can learn complex patterns and relationships within financial time-series data. Similarly, the Informer model is optimized for handling long-sequence time-series forecasting by utilizing sparse attention mechanisms. This feature helps Informer manage long-term dependencies in financial data, which is essential when working with datasets that span many years or contain high-frequency data points. Both models, by incorporating these advanced features, are capable of handling the inherent complexity and volatility of financial markets. Studies have shown that these transformer models, when applied to financial portfolio optimization, are able to outperform traditional models, including classical statistical approaches and machine learning methods like LSTMs, in terms of prediction accuracy and portfolio performance. As financial markets continue to evolve, transformer models like TFT and Informer are poised to play an increasingly important role in optimizing investment strategies and improving decision-making in finance.

6.2.3 Challenges in Implementing Transformer Architectures for Financial Applications

Despite the promising potential of transformer-based architectures like Temporal Fusion Transformer and Informer in financial portfolio optimization, there remain significant challenges in their practical implementation. One major obstacle is the limited availability of validated implementations and comprehensive resources beyond the original research papers. Given that these architectures are relatively new in the field, there is a scarcity of open-source codes and fully documented frameworks that could facilitate their adoption. This poses a challenge for researchers and practitioners attempting to integrate them into real-world financial applications. Future research should focus on developing more accessible and optimized implementations of these architectures, along with standardized benchmarks to evaluate their performance against existing financial models. Furthermore, industry collaborations and open-source contributions could help bridge the gap between theoretical advancements and practical applications, ensuring that these advanced transformer models can be effectively utilized for financial decision-making and portfolio optimization.

Bibliography

[1] “What is Digital Asset Market.” [Online]. Available: https://jcl.law.uiowa.edu/sites/jcl.law.uiowa.edu/files/2021-08/Kaal_Final_Web_0.pdf

[2] “Types of Digital Assets” [Online]. Available: https://www.investax.io/blog/what-are-digital-assets

[3] “Risks and Challenges of Digital Asset Market” [Online]. Available: https://www.chainbits.com/blockchain-101/understanding-the-risks-of-digital-asset-trading/

[4] [Online]. Available: https://www.axios.com/2025/05/01/bitcoin-etf-investing-stablecoin

[5] [Online]. Available: https://www.mckinsey.com/industries/financial-services/our-insights/from-ripples-to-waves-the-transformational-power-of-tokenizing-assets

[6] [Online]. Available: https://en.wikipedia.org/wiki/Portfolio_optimization

[7] [Online]. Available: https://www.investopedia.com/terms/e/efficientfrontier.asp

[8] [Online]. Available: https://www.fe.training/free-resources/portfolio-management/portfolio-optimization/

[9] [Online]. Available: https://www.fe.training/free-resources/portfolio-management/portfolio-optimization/

[10] [Online]. Available: https://cfastudyguide.com/mean-variance-optimization/

[11] [Online]. Available: https://www.investopedia.com/terms/b/black-litterman_model.asp

[12] [Online]. Available: https://corporatefinanceinstitute.com/resources/financial-modeling/monte-carlo-simulation/

[13] [Online]. Available: https://www.wallstreetmojo.com/portfolio-optimization/

[14] [Online]. Available: https://www.realized1031.com/blog/what-are-the-benefits-cons-and-limitations-of-modern-portfolio-theory

[15] [Online]. Available: https://www.linkedin.com/advice/0/what-advantages-disadvantages-using-quantitative-hitac

[16] [Online]. Available: https://www.investopedia.com/articles/company-insights/083016/example-applying-modern-portfolio-theory-mps.asp

[17] [Online]. Available: https://arxiv.org/pdf/2307.01599

[18] [Online]. Available: https://www.dataairevolution.com/2024/10/why-transformers-are-the-future-limitations-of-lstms-and-how-theyre-solved/
[19] [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf
[20] [Online]. Available: https://www.youtube.com/watch?v=bCz4OMemCcA
[21] [Online]. Available: https://research-api.cbs.dk/ws/portalfiles/portal/76453232/1427693_MScThesis_Final_Kristin_Arnor.pdf
[22] [Online]. Available: https://en.wikipedia.org/wiki/Deep_reinforcement_learning
[23] [Online]. Available: https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
[24] [Online]. Available: https://en.wikipedia.org/wiki/Deep_learning
[25] [Online]. Available: https://www.spiceworks.com/tech/artificial-intelligence/articles/what-is-markov-decision-process/
[26] [Online]. Available: https://www.baeldung.com/cs/ml-policy-reinforcement-learning
[27] [Online]. Available: https://www.geeksforgeeks.org/model-free-reinforcement-learning-an-overview/
[28] [Online]. Available: https://medium.com/@samina.amin/deep-q-learning-dqn-71c109586bae
[29] [Online]. Available: https://en.wikipedia.org/wiki/Policy_gradient_method
[30] [Online]. Available: https://arxiv.org/pdf/2310.19432
[31] [Online]. Available: https://medium.com/intro-to-artificial-intelligence/the-actor-critic-reinforcement-learning-algorithm-c8095a655c14
[32] [Online]. Available: https://www.geeksforgeeks.org/model-based-reinforcement-learning-mbrl-in-ai/
[33] [Online]. Available: https://www.researchgate.net/publication/387522515_Deep_Reinforcement_Learning_for_Financial_Portfolio_Optimization
[34] [Online]. Available: https://www.investopedia.com/terms/h/high-frequency-trading.asp

[35] [Online]. Available:
https://www.sciencedirect.com/science/article/pii/S2590005625000177#sec6
[36] [Online]. Available:
https://docs.pytorch.org/docs/stable/generated/torch.nn.Transformer.html
[37] [Online]. Available: https://medium.com/@pickleprat/encoder-only-architecture-bert-4b27f9c76860
[38] [Online]. Available: https://medium.com/@RobuRishabh/types-of-transformer-model-1b52381fa719
[39] [Online]. Available: https://www.geeksforgeeks.org/introduction-to-generative-pre-trained-transformer-gpt/
[40] [Online]. Available: https://huggingface.co/docs/transformers/en/model_doc/vit
[41] [Online]. Available: https://www.geeksforgeeks.org/adam-optimizer/
[42] [Online]. Available: https://www.geeksforgeeks.org/epsilon-greedy-algorithm-in-reinforcement-learning/
[43] [Online]. Available: https://medium.com/dataness-ai/understanding-temporal-fusion-transformer-9a7a4fcde74b
[44] [Online]. Available: https://arxiv.org/pdf/2411.19649
[45] [Online]. Available: https://medium.com/@bijit211987/transformers-like-informer-arrevolutionizing-time-series-forecasting-f4e4ebd7db1b
[46] [Online]. Available: https://arxiv.org/pdf/2206.03246
[47] [Online]. Available: https://www.datacamp.com/tutorial/how-transformers-work
[48] [Online]. Available: https://www.investopedia.com/terms/a/assetallocation.asp
[49] [Online]. Available:
https://www.researchgate.net/publication/385860012_Natural_language_processing_nlp_for_financial_text_analysis
[50] [Online]. Available: https://aws.amazon.com/what-is/blockchain/?aws-products-all.sort-by=item.additionalFields.productNameLowercase&aws-products-all.sort-order=asc
[51] [Online]. Available: https://www.ibm.com/think/topics/natural-language-processing
[52] [Online]. Available: https://www.ibm.com/think/topics/recurrent-neural-networks
[53] [Online]. Available: https://www.geeksforgeeks.org/gated-recurrent-unit-networks/
[54] [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[55] [Online]. Available: https://en.wikipedia.org/wiki/BERT_(language_model)

[56] [Online]. Available: https://www.datacamp.com/tutorial/introduction-to-convolutional-neural-networks-cnns

[57] [Online]. Available: https://en.wikipedia.org/wiki/BLEU

[58] [Online]. Available: https://www.researchgate.net/publication/387524930_Financial_Time_Series_Analysis_with_Transformer_Models

[59] [Online]. Available: https://medium.com/%40marvelous_catawba_otter_200/attention-is-all-you-need-f9fe38d6e2fc

[60] [Online]. Available: https://www.geeksforgeeks.org/adam-optimizer/