AUTOMATED TRADING IN LIMIT ORDER BOOK USING REINFORCEMENT LEARNING

Major Project Report

Submitted by:

Joshua Mani Vinod	**Suyash Subhash Ingle**	**Sunil Kumar Bhakhar**
B210616EC	B210643EC	B210680EC

In partial fulfillment for the award of the Degree of

BACHELOR OF TECHNOLOGY

IN

ELECTRONICS AND COMMUNICATION ENGINEERING

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY, CALICUT

NIT CAMPUS P.O., CALICUT

KERALA, INDIA 673601.

APRIL 2025

NATIONAL INSTITUTE OF TECHNOLOGY, CALICUT

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING

CERTIFICATE

This is to certify that the major project report entitled ***"AUTOMATED TRADING IN LIMIT ORDER BOOK USING REINFORCEMENT LEARNING"*** *is a bonafide record of the Project done by* ***Joshua Mani Vinod*** *(B210616EC),* ***Suyash Subhash Ingle*** *(B210643EC) and* ***Sunil Kumar Bhakar*** *(B210680EC) under our supervision, in partial fulfillment of the requirements for the award of the degree of* ***Bachelor of Technology in Electronics and Communication Engineering*** *from* ***National Institute of Technology Calicut****, and this work has not been submitted elsewhere for the award of a degree.*

Dr Anup Aprem
Assistant Professor (Grade I)
ECED, NIT Calicut

Dr. G. Sreelekha
Head of the Department
ECED, NIT Calicut

Place: NIT Calicut

Date: 07/04/2025

ACKNOWLEGEMENT

We express our deepest gratitude to all those who supported and guided us throughout the successful completion of our final year project.

First and foremost, we extend our sincere thanks to our project guide, **Dr. Anup Aprem**, Assistant Professor, Department of Electronics and Communication Engineering, NIT Calicut, for his invaluable guidance, continuous encouragement, and unwavering support. His insights and expertise have been instrumental at every stage of this project.

We would also like to express our heartfelt appreciation to Dr. G. Sreelekha, Head of the Department, Electronics and Communication Engineering, NIT Calicut, for providing us with the necessary facilities and a conducive environment to carry out our work effectively.

Finally, we are grateful to all the faculty members, peers, and well-wishers who directly or indirectly contributed to the successful completion of this project.

ABSTRACT

This work presents a novel trading strategy that leverages modern deep learning architectures by combining Deep Reinforcement Learning (DRL) with a Decision Transformer (DT) framework. Traditionally, methods such as Deep Q-Networks (DQN) have been employed to develop trading agents, but our approach integrates the Decision Transformer in an offline reinforcement learning setting, which has shown promising improvements in decision-making quality and cumulative returns. We preprocess historical stock price data from major companies(*Google, Microsoft, Amazon etc*) using technical indicators—including MACD, RSI, and various return measures—to construct rich state representations for the learning process. Experimental evaluations demonstrate that our hybrid model not only achieves competitive performance but also offers enhanced robustness over traditional methods, suggesting that the combination of DRL with transformer-based architectures can pave the way for more effective and profitable trading strategies.

Keywords— Reinforcement Learning, Deep Q-learning, Decision Transformer, Adaptive Trading Strategies, Offline RL

CONTENTS

List of Figures

List of Tables

List of Abbreviations

RL	Reinforcement Learning
DRL	Deep Reinforcement Learning
DT	Decision Transformer
DQN	Deep Q-Network
A2C	Advantage Actor-Critic
LOB	Limit Order Book
HFT	High-Frequency Trading
MACD	Moving Average Convergence Divergence
RSI	Relative Strength Index
EMA	Exponential Moving Average
EWMA	Exponentially Weighted Moving Average
RTG	Return-To-Go
MSE	Mean Squared Error
MA	Moving Average
BNN	Bayesian Neural Network
TD	Temporal Difference
SD	Standard Deviation

CHAPTER 1

INTRODUCTION

Financial trading has been a widely researched topic, and a variety of methods have been proposed to trade markets over the last few decades. These methods include *fundamental analysis*, *technical analysis*, and *algorithmic trading*. Many practitioners utilize a hybrid of these techniques to make trading decisions. Among these, algorithmic trading has gained significant interest in recent years, accounting for about 75% of the trading volume in the United States stock exchanges. The advantages of algorithmic trading are numerous, including strong computational foundations, faster execution, and risk diversification. A key component of such trading systems is a predictive signal that identifies profitable opportunities. Mathematical and statistical methods are widely applied to derive these signals. However, due to the low signal-to-noise ratio in financial data and the dynamic nature of the markets, the design of effective predictive signals is non-trivial, and their effectiveness can vary over time.

In recent years, *machine learning* algorithms have gained widespread popularity in many domains, with notable successes in applications such as *image classification* and *natural language processing*. Similar techniques have also been applied to financial markets in an effort to generate a higher alpha. Although most of the research has focused on regression and classification pipelines, where excess returns or market movements are predicted over fixed horizons, there is relatively little discussion about how to transform these predictive signals into actual trade positions. This mapping is non-trivial; predictive horizons are often short (e.g., one day or a few days ahead), yet large trends can persist for weeks or even months, with moderate consolidation periods. Thus, beyond having a good predictive signal, it is equally important to have a signal that consistently produces correct directional calls.

To address this complexity, adaptive trading strategies capable of real-time decision making are increasingly necessary. Reinforcement Learning (RL), a subset of machine

learning, presents a promising solution by enabling agents to learn optimal trading strategies through trial and error. This project explores the application of RL techniques in automating the development of strategies for High-Frequency Trading(HFT). Using state-of-the-art RL models, such as Deep Q-learning Networks (DQN), Policy Gradients, and Advantage Actor-Critic (A2C), our goal is to create robust trading strategies that improve decision-making accuracy. The study also integrates advanced features such as time series momentum and technical indicators to provide the RL agent with a deeper understanding of the market environment.

Building on these methods, the Decision Transformer (DT) offers a novel approach by reframing RL as a sequence modeling problem. The DT model leverages a transformer architecture, originally developed for language tasks, to learn from offline datasets, sidestepping the limitations of traditional RL methods that require active interaction with the environment. By utilizing past trajectories, DT combines historical observations, actions, and reward expectations to predict optimal future actions, demonstrating its potential for high-stakes applications like trading.

1.1 Motivation

The primary motivation behind this project stems from the limitations of traditional trading strategies in handling the volatility and complexity of high-frequency trading. Static, rule-based strategies often underperform in rapidly changing market conditions, as they cannot adapt to sudden fluctuations. Additionally, the sheer volume of data in HFT requires automated systems that can process information and execute trades autonomously. Reinforcement learning provides a framework for developing adaptive trading strategies that evolve based on market conditions, promising greater flexibility and effectiveness than conventional methods.

Another motivating factor is the potential for RL-based trading strategies to revolutionize the way trading algorithms are developed. By integrating time-series momentum and technical indicators, the RL model gains a richer set of inputs, which facilitates better-informed decision-making. Volatility scaling techniques are also applied to refine reward functions, improving the stability and efficiency of the learning process. Another key motivation for this project is the potential to revolutionize the way trading strategies are developed and deployed. RL allows us to create strategies that are not only more robust, but also capable of outperforming traditional methods. Early simulations in this project

have shown promising results, indicating that RL has the potential to significantly improve trading performance in HFT environments.

Incorporating the Decision Transformer model into our framework offers exciting new avenues. DT's ability to operate with offline data enables it to avoid costly or risky real-time interactions with the market, instead learning from historical data. This capability is especially advantageous in the financial domain, where each action carries potential financial risks. DT's sequence modeling approach enhances the agent's ability to recognize patterns and predict profitable actions, leading to trading strategies that are not only adaptive but also optimized for long-term profitability.

1.2 Organization of the report

This report is structured into several sections to provide a comprehensive understanding of the research and implementation of reinforcement learning-based trading strategies for high-frequency trading (HFT). The organization is as follows:

Background This section provides an introduction to the research area, including a literature survey on existing trading strategies and reinforcement learning applications. It defines the problem statement and outlines the objectives of the project.

Methodology The methodology section details the approach taken to develop and evaluate the RL-based trading strategies. It covers:

- **Data Representation and Preprocessing:** The structure and processing of historical market data for model training.
- **State Representation:** How market states are defined to inform decision-making.
- **Action Representation and Epsilon-Greedy Selection:** The set of possible actions for the agent and the strategy for balancing exploration and exploitation.
- **Reward Function:** The design of reward functions, incorporating volatility scaling, transaction costs, and profit modeling (additive vs. multiplicative).
- **Experience Replay:** The mechanism for improving learning efficiency by reusing past experiences.
- **Agents:** The implementation of Deep Q-Network (DQN) and Decision Transformer models.
- **Hyperparameters:** Key parameters influencing model performance.

- **Simulating Market Behavior:** Techniques used to create a realistic trading environment for model evaluation.

Experiments: In the Experiments section, findings from various tests and simulations are discussed. This section is organized by examining the baseline models, comparative analyses with previous algorithms, and insights into the decision transformer's performance. Key observations are highlighted to discuss the effectiveness and limitations of the algorithm.

Conclusion: The final section, Conclusion, draws together the main results and offers suggestions for further research or practical applications. This section reviews the technological, financial, and operational factors that can impact future work and outlines the next steps for development.

CHAPTER 2

BACKGROUND

2.1 Literature survey

Financial Time Series Forecasting [5]: Financial markets have long been the subject of predictability studies, with statistical (econometric) parametric models and machine learning approaches being the two primary methods for forecasting. Traditional econometric models, such as AR, MA, and ARMA [5], have been widely used but often fall short due to the nonlinear and non-stationary nature of financial time series. These models also struggle with high-frequency data, where price movements are influenced more by short-term factors than fundamental economics. In contrast, machine learning techniques, particularly deep learning models like CNNs and LSTMs, have gained traction due to their ability to capture complex, nonlinear patterns and process large volumes of data, making them better suited for financial forecasting.

Financial Time Series Forecasting with Uncertainty [6]: Uncertainty in financial predictions can be expressed through Bayesian inference, using both parametric and nonparametric methods. Gaussian Processes (GPs) and Bayesian Neural Networks (BNNs) are notable for modeling uncertainty, although GPs lack the feature extraction capabilities of deep learning models. BNNs, despite being underutilized in finance due to training difficulties, offer a way to quantify risk and predict price movements. Combining deep learning with Quantile Regression (QR)[6] further enhances uncertainty modeling by estimating multiple return quantiles simultaneously, providing a more comprehensive view of risk exposure in high-frequency trading environments.

Deep Reinforcement Learning for Trading [7, 8]: Reinforcement learning (RL) in trading is divided into model-based and model-free approaches. While model-based methods attempt to model market dynamics, they are less explored due to the complexities

of financial time series. Model-free methods, including Deep Q-Networks (DQN) and actor-critic approaches, are more commonly used. These methods allow for continuous action spaces, enabling scaling of positions based on market conditions. However, training stability is a challenge, requiring techniques like Double DQN and volatility scaling.[**?**] Despite these challenges, RL has shown promise in optimizing trading strategies by learning from market data through trial and error.

Advantage Actor-Critic (A2C)

Advantage Actor-Critic (A2C) [9] is a policy-based reinforcement learning algorithm that learns both a policy and a value function simultaneously:

- **Actor**: Updates the policy by selecting actions that maximize the expected return.
- **Critic**: Evaluates the actions by estimating the value of the current state.
- **Advantage Function**:

$$A(s, a) = Q(s, a) - V(s) \tag{2.1}$$

  is used to reduce variance in gradient estimation.

A2C directly learns the policy, making it more sample-efficient in continuous action spaces. The critic provides feedback to the actor by assessing the quality of actions, which helps stabilize the training process.

Proximal Policy Optimization (PPO)

Proximal Policy Optimization (PPO) [10] is a policy-gradient method that improves upon standard policy optimization techniques by using a surrogate loss function:

$$L_{\text{PPO}} = \min\left(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right) \tag{2.2}$$

Key features of PPO:

- Uses a clipped surrogate objective to prevent large updates, ensuring that the new policy does not deviate too far from the old policy.
- Balances exploration and exploitation using the advantage function $\hat{A}_t$.
- Works well in continuous action spaces and is computationally efficient.

PPO is widely used due to its stability, ease of implementation, and strong performance across a variety of tasks.

Deep Q-Networks (DQN): [7]: DQN is a value-based reinforcement learning algorithm that uses a neural network to approximate the Q-value function

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \right) \tag{2.3}$$

Key features of DQN:

- Uses experience replay to store past transitions and updates from a mini-batch of experiences.
- Utilizes a target network to stabilize learning by fixing the target Q-values during updates.
- Learns the optimal Q-values by minimizing the temporal difference (TD) error, updating the policy indirectly from the learned Q-values.

DQN is effective in scenarios with discrete actions and works well in environments with significant state spaces.

Decision Transformer [4]: Decision Transformer is a sequence modeling-based reinforcement learning algorithm that utilizes a transformer architecture to predict actions based on past states, actions, and rewards.

Key features of Decision Transformer:

- Uses a causal transformer to model trajectories as sequences, learning the dependencies between states, actions, and rewards.
- Conditions action selection on desired return-to-go (RTG), enabling flexible goal-directed behavior.
- Learns optimal policies by predicting future actions directly from past trajectories, rather than estimating value functions explicitly.

Decision Transformer is effective in offline reinforcement learning settings and works well in environments where long-horizon dependencies are important.

2.2 Problem Statement

Traditional Deep Reinforcement Learning (DRL) approaches, such as Deep Q-Networks (DQN), have been widely used for developing stock trading agents. However, these methods often struggle with stability, data efficiency, and generalization in highly dynamic and noisy financial markets. There is a need for a more robust and effective framework that can leverage past trading experiences to make better sequential decisions without the limitations of online interaction. This work addresses the challenge by proposing the integration of a Decision Transformer (DT) within an offline reinforcement learning setting, aiming to enhance decision-making quality, improve cumulative returns, and increase robustness in stock trading strategies.

2.3 Objectives

The primary objective of this project is to develop and evaluate Reinforcement Learning-based trading strategies for continuous futures contracts in a high-frequency trading environment. Specifically, the project aims to:

- Implement RL algorithms such as Deep Q-learning Networks, Decision Transformer to automate the strategy development process.
- Explore both discrete and continuous action spaces to determine the most effective approach for decision-making in real-time trading.
- Enhance state representations by incorporating features from time series momentum and technical indicators, with the application of volatility scaling techniques to improve reward functions.
- Conduct simulations to evaluate the performance of the developed strategies and identify areas for further improvement.

CHAPTER 3

METHODOLOGY

This methodology is centered around the idea of using historical stock data, including price and volume, to make decisions such as buying, holding, or closing positions. The environment is built to track price movements across a given number of bars (e.g., daily, weekly, or other time intervals), calculate rewards based on trading actions, and manage the commission fees incurred during trading.

3.1 Data Representation

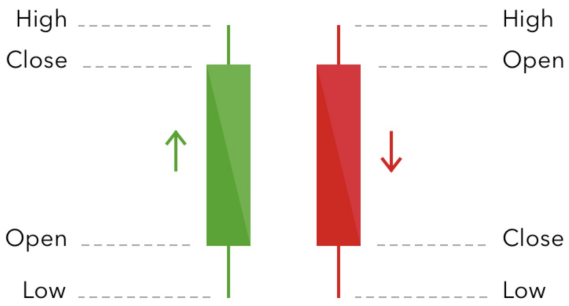


Figure 3.1: Open, High, Low, and Close (OHLC) prices for each trading interval

The stock data [8, 11] utilized in the simulation consists of open, high, low, close prices, and trading volume (`<OPEN>`, `<HIGH>`, `<LOW>`, `<CLOSE>`, `<VOL>`). These attributes are organized within a DataFrame (`data`), serving as the historical price dataset that informs the agent's trading decisions. The dataset contains over 10,000 data points for each of the leading companies such as Google, Microsoft, Apple, IBM, and others.

3.2 Data Preprocessing

In addition to the fundamental stock features mentioned earlier, we incorporate various technical indicators to enhance the state representation for the trading agent. These include the Moving Average Convergence Divergence (MACD) [12] and the Relative Strength Index (RSI) [13], which help identify market trends and momentum. Furthermore, we normalize closing prices and compute historical returns over different time frames, such as "MonthlyReturn", "BiMonthlyReturn", "QuarterlyReturn", and "AnnualReturn", to capture price fluctuations effectively.

The preprocessing steps involve calculating MACD using exponential moving averages (EMAs) of the closing prices and deriving the RSI based on average gains and losses over a rolling window. Additionally, we normalize closing prices relative to their maximum value and compute returns adjusted for volatility over different periods. This enriched dataset provides a more comprehensive input for the trading strategy.

3.3 State Representation

The state space is designed to capture relevant market information by incorporating historical prices, returns, and technical indicators. At each time step, the state consists of the past 60 observations for a set of selected features, providing a comprehensive view of market trends and momentum.

At each timestep t, the state s_t is represented as:

$$\begin{aligned} s_t = [&\texttt{Close\_Norm}_{t-60:t}, \texttt{MonthlyReturn}_{t-60:t}, \\ &\texttt{BiMonthlyReturn}_{t-60:t}, \texttt{QuarterlyReturn}_{t-60:t}, \\ &\texttt{AnnualReturn}_{t-60:t}, \texttt{MACD}_{t-60:t}, \texttt{RSI}_{t-60:t}], \end{aligned} \tag{3.1}$$

resulting in a state dimension of $60 \times 7 = 420$.

3.3.1 Normalized Closing Price

The normalized closing price is computed to scale raw prices into a comparable range, which helps the model learn more effectively across different assets. We define:

$$\texttt{Close\_Norm}_t = \frac{p_t - \mu_{t-60:t}}{\sigma_{t-60:t}}, \tag{3.2}$$

where p_t is the raw closing price at time t, and $\mu_{t-60:t}$ and $\sigma_{t-60:t}$ are the mean and standard deviation of closing prices over the past 60 days, respectively.

3.3.2 Returns over 1M, 2M, 6M, 1Y

We include returns over multiple horizons to capture both short- and long-term price movements:

- **Monthly return:**

$$\texttt{MonthlyReturn}_t = \frac{p_t - p_{t-21}}{p_{t-21}}, \tag{3.3}$$

  where 21 trading days = 1 month.

- **Bi-monthly return:**

$$\texttt{BiMonthlyReturn}_t = \frac{p_t - p_{t-42}}{p_{t-42}}, \tag{3.4}$$

  where 42 trading days = 2 months.

- **Quarterly return:**

$$\texttt{QuarterlyReturn}_t = \frac{p_t - p_{t-63}}{p_{t-63}}, \tag{3.5}$$

  where 63 trading days = 3 months.

- **Annual return:**

$$\texttt{AnnualReturn}_t = \frac{p_t - p_{t-252}}{p_{t-252}}, \tag{3.6}$$

  where 252 trading days = 1 year.

3.3.3 Technical Parameters (MACD and RSI)

MACD (Moving Average Convergence Divergence) MACD helps identify trend direction and momentum by comparing short- and long-term exponentially weighted moving averages (EMAs) of price. Following [12], we compute:

$$q_t = \frac{m(S)_t - m(L)_t}{\mathrm{std}\big(p_{t-63:t}\big)}, \quad \mathrm{MACD}_t = \frac{q_t}{\mathrm{std}\big(q_{t-252:t}\big)}, \tag{3.7}$$

where $m(S)_t$ and $m(L)_t$ are the EMAs of the price series with time scales S (short) and L (long), respectively. The EMA is defined recursively as

$$m(k)_t = \alpha_k p_t + (1 - \alpha_k) m(k)_{t-1}, \quad \text{with } \alpha_k = \frac{2}{k+1}.$$

Typically, we set $S = 12$ and $L = 26$.

RSI (Relative Strength Index) The RSI indicator, proposed in [13], is an oscillating indicator that fluctuates between 0 and 100. It helps identify whether an asset is in oversold conditions (with values below 20) or overbought conditions (with values above 80) by evaluating the intensity of recent price movements. In our state representations, we incorporate the RSI with a 30-day look-back window.

$$\text{RSI}_t = 100 - \frac{100}{1 + \frac{\text{AvgGain}_{t-30:t}}{\text{AvgLoss}_{t-30:t}}}, \tag{3.8}$$

where $\text{AvgGain}_{t-30:t}$ and $\text{AvgLoss}_{t-30:t}$ are defined as:

$$\text{AvgGain}_{t-30:t} = \frac{1}{30} \sum_{i=t-29}^{t} \max(P_i - P_{i-1}, 0), \tag{3.9}$$

$$\text{AvgLoss}_{t-30:t} = \frac{1}{30} \sum_{i=t-29}^{t} \max(P_{i-1} - P_i, 0), \tag{3.10}$$

with P_i denoting the closing price at time i. The max function ensures that only positive gains contribute to AvgGain, and only positive losses contribute to AvgLoss.

3.4 Action Representation

The environment is designed around three possible actions an agent can take:

- **Buy (+1):** Taking a fully long position in the stock.
- **Sell (-1):** Taking a fully short position in the stock.
- **Hold (0):** Maintaining the current position without executing a trade.

This type of action space representation is often referred to as **target orders**, where instead of making direct trading decisions (e.g., placing market orders), the agent decides the position it wants to hold. If the current action remains the same as the previous action, no transaction cost is incurred, and the agent simply maintains its position. However, if the

agent moves directly from a long position to a short position (or vice versa), the transaction cost is doubled.

3.4.1 Epsilon-Greedy Action Selection

To balance *exploration* (trying new actions) and *exploitation* (choosing the best-known action), we use an Epsilon-greedy action selection method. The agent selects an action as follows:

- With probability ϵ, the agent randomly selects an action (exploration).
- With probability $1 - \epsilon$, the agent chooses the action with the highest Q-value (exploitation).

Initially, ϵ is set to a high value (e.g., 1.0) to encourage more exploration. Over time, ϵ gradually decays using an exponential decay function:

$$\epsilon = \max(\epsilon_{\text{end}}, \epsilon \times \epsilon_{\text{decay}}) \tag{3.11}$$

where:

- ϵ_{end} is the minimum epsilon value (e.g., 0.01).
- ϵ_{decay} is the decay factor (e.g., 0.995 per epoch).

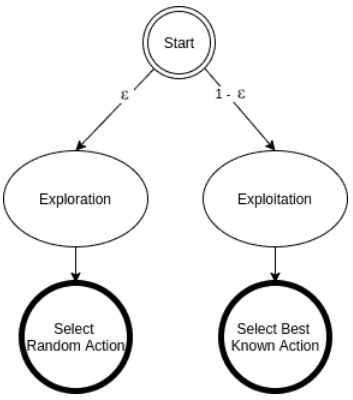


Figure 3.2: Epsilon-Greedy Action Selection [1]

This ensures that the agent starts with more random exploration and gradually shifts toward choosing the best-known actions as training progresses. The Epsilon-greedy strategy is crucial in reinforcement learning, as it helps the agent discover better policies while avoiding getting stuck in suboptimal strategies.

3.5 Reward Function

The design of the reward function [7] is crucial in reinforcement learning, as it determines the agent's behavior. In this work, we define the reward function based on additive profits while incorporating market volatility considerations. The reward at time t is given by:

$$R_t = \mu \left[\left(\frac{\sigma_{\text{tgt}}}{\sigma_{t-1}} A_{t-1} \right) r_t + (-C \cdot p_{t-1}) \cdot \left| \left(\frac{\sigma_{\text{tgt}}}{\sigma_{t-1}} A_{t-1} - \frac{\sigma_{\text{tgt}}}{\sigma_{t-2}} A_{t-2} \right) \right| \right] \tag{3.12}$$

where:

- σ_{tgt} is the volatility target.
- σ_{t-1} is an ex-ante volatility estimate, calculated using an exponentially weighted moving standard deviation with a 60-day window on r_t:

$$\sigma_{t-1} = \sqrt{(1-\lambda) \sum_{i=1}^{60} \lambda^{i-1} (r_{t-i} - \bar{r})^2}$$

  where λ is the decay factor (e.g., 0.94) and $\bar{r}$ is the weighted average return.

$$\bar{r} = \frac{\sum_{i=1}^{60} \lambda^{i-1} r_{t-i}}{\sum_{i=1}^{60} \lambda^{i-1}}$$

- A_{t-1} and A_{t-2} are the agent's positions at time $t-1$ and $t-2$, respectively.
- C is the transaction cost rate (e.g., 0.0001 per unit of traded value).
- p_{t-1} is the previous price of the asset.
- $r_t = p_t - p_{t-1}$ represents additive profits.

3.5.1 Interpretation of Reward Function

1. Profit Contribution (First Term)

$$\left(\frac{\sigma_{\text{tgt}}}{\sigma_{t-1}} A_{t-1}\right) r_t$$

- Volatility scaling: Positions are rescaled by the ratio $\sigma_{\text{tgt}}/\sigma_{t-1}$ to normalize risk. High past volatility reduces position impact, while low volatility amplifies it.
- Correct action reward: If the agent buys ($A_{t-1} = 1$) and price increases ($r_t > 0$), reward is positive. If it sells ($A_{t-1} = -1$) and price decreases ($r_t < 0$), reward is also positive.
- Incorrect action penalty: Opposite scenarios yield negative contributions.

2. Transaction Cost Penalty (Second Term)

$$(-C \cdot p_{t-1}) \cdot \left| \left(\frac{\sigma_{\text{tgt}}}{\sigma_{t-1}} A_{t-1} - \frac{\sigma_{\text{tgt}}}{\sigma_{t-2}} A_{t-2}\right) \right|$$

- Cost factor C: A small per-unit cost discourages unnecessary trading.
- Position change: Measures change in normalized position between $t-2$ and $t-1$. Large switches incur higher costs.
- Stability incentive: Encourages smoother trading behavior by penalizing frequent or extreme reversals.

3.5.2 Example Scenarios

- Profitable trade: Agent buys at $p_{t-1} = 100$ ($A_{t-1} = 1$), price rises to $p_t = 105$, so $r_t = 5$. The first term yields a positive reward; the cost term is zero if no position change occurred.
- Wrong trade: Agent sells at $p_{t-1} = 100$ ($A_{t-1} = -1$), price rises to $p_t = 105$. The first term is negative, reflecting a loss, and the cost term remains zero if the position was unchanged.
- Frequent trading penalty: Agent buys at $t-2$, sells at $t-1$, buys again at t. Each switch triggers the cost penalty, reducing net reward.

3.5.3 Effect on Learning

This reward design:

- Encourages profitable positions by rewarding correct directional bets on price changes.
- Discourages overtrading through explicit transaction cost penalties, promoting stability.
- Adapts to market conditions via volatility scaling, ensuring the agent adjusts position sizes in turbulent markets.

3.5.4 Volatility Scaling

The volatility scaling ensures that our positions are scaled up in low-volatility conditions and scaled down in high-volatility conditions. This normalization method allows rewards from different assets to be comparable, making training more stable. Since our dataset consists of various assets with different price ranges, we normalize rewards across different contracts to maintain a consistent scale.

3.5.5 Transaction Costs

The transaction cost term includes a price factor p_{t-1}, as different assets have different costs. The transaction cost is proportional to the absolute change in the volatility-adjusted position, meaning that switching from a fully long position to a short position incurs higher costs.

3.5.6 Additive vs. Multiplicative Profits

In this work, we use additive profits because they are appropriate when trading a fixed number of shares or contracts. If we were trading a fraction of accumulated wealth at each step, multiplicative profits would be more suitable, requiring a logarithmic transformation. However, logarithmic transformations penalize large wealth growths, making them less desirable for our setup.

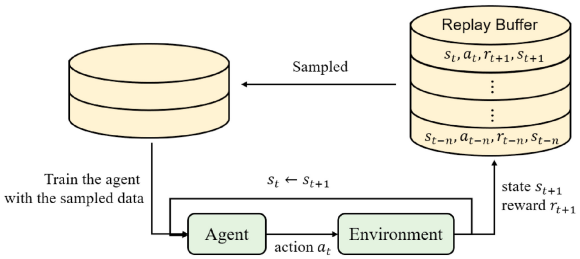


Figure 3.3: Experience Replay Buffer [2]

3.6 Experience Replay

Experience Replay is a crucial technique employed in Deep Q-Networks (DQN) to enhance training stability and efficiency. It involves storing past interactions with the environment in a replay buffer in the form of tuples: (state, action, reward, next state). Rather than learning directly from sequential experiences, which are often highly correlated, the agent samples mini-batches of experiences uniformly at random. This randomization breaks the temporal correlations and reduces the variance of updates, thereby stabilizing and improving the learning process.

Algorithm 1 Experience Replay Implementation

Initialize memory $\mathcal{D}$ with capacity N
procedure PUSH(s_t, a_t, r_t, s_{t+1})
    if len($\mathcal{D}$) $\geq N$ **then**
        Remove oldest transition ▷ FIFO replacement
    end if
    Append (s_t, a_t, r_t, s_{t+1}) to $\mathcal{D}$
end procedure
procedure SAMPLE(B)
    Return B random transitions from $\mathcal{D}$ ▷ Uniform sampling
end procedure

In the context of stock market trading, the DQN agent leverages this replay buffer to make more informed decisions, such as whether to Buy, Hold, or Sell a stock. By reusing past experiences, the agent learns a more generalizable policy that adapts better to dynamic financial environments. While Prioritized Experience Replay (PER) [14], which samples more informative transitions more frequently, can further enhance performance, it was not employed in our experiments.

Overall, the use of Experience Replay significantly improves sample efficiency, prevents overfitting to recent data, and contributes to more stable and robust policy learning in reinforcement learning-based trading systems.

3.7 Baseline Algorithms

We evaluate three baseline trading strategies: Long-Only, Sign(R), and MACD. Each algorithm employs distinct decision-making mechanisms, as detailed below.

Characteristic	**Long-Only**	**Sign(R)**	**MACD**
Signal Basis	Buy & hold	1-period return	EMA crossover
Position Types	Long only	Long/Short	Long/Short
Noise Sensitivity	Low	High	Medium
Market Adaptation	Bullish only	Trending	Medium trends
Transaction Cost	Low	High	Medium

Table 3.1: Baseline Algorithm Comparison

3.7.1 Long-Only Strategy

This strategy operates on the principle that asset values appreciate over time. It exclusively takes long positions, executing buy actions when technical indicators suggest upward momentum (e.g., moving average crossovers) and holding positions until episode termination. Key features include simplicity, low transaction costs, and inherent bullish market bias. Limitations include inability to profit from downward trends and suboptimal performance in sideways markets.

This strategy operates on the principle that asset values appreciate over time. It exclusively takes long positions using price trend detection:

$$\text{Buy Signal} = \begin{cases} 1 & \text{if } SMA_{10}(t) > SMA_{50}(t) \\ 0 & \text{otherwise} \end{cases} \tag{3.13}$$

where $SMA_n(t)$ denotes the simple moving average over n periods. The strategy holds positions until episode termination, featuring low transaction costs but inherent bullish market bias.

3.7.2 Sign(R) Strategy

A momentum-based approach that reacts to the directionality of immediate price changes. The agent buys when the previous timestep's return is positive ($R_t > 0$) and sells when

negative ($R_t < 0$). While computationally efficient and responsive to trends, it suffers from high noise sensitivity and frequent position flipping due to its single-period lookback window.

A momentum-based approach using single-period returns:

$$R_t = P_t - P_{t-1} \tag{3.14}$$

$$\text{Position}_t = \begin{cases} \text{Long} & \text{if } R_t > 0 \\ \text{Short} & \text{if } R_t < 0 \\ \text{Hold} & \text{otherwise} \end{cases} \tag{3.15}$$

This reactive strategy excels in trending markets but suffers high noise sensitivity due to its minimal lookback window.

3.7.3 MACD Strategy

Utilizes the Moving Average Convergence Divergence indicator to identify medium-term trends. Trading signals generate when the MACD line (12-period EMA minus 26-period EMA) crosses its signal line (9-period EMA of MACD). This approach reduces market noise through exponential smoothing but introduces lag in signal generation, particularly during volatile or range-bound market conditions.

Utilizes exponential moving averages (EMAs) for trend identification:

$$MACD_t = EMA_{12}(P_t) - EMA_{26}(P_t) \tag{3.16}$$

$$Signal_t = EMA_9(MACD_t) \tag{3.17}$$

The Exponential Moving Average (EMA) is defined recursively as:

$$EMA_t = \alpha \cdot P_t + (1 - \alpha) \cdot EMA_{t-1} \tag{3.18}$$

where P_t is the current price, and the smoothing factor α is given by:

$$\alpha = \frac{2}{N + 1} \tag{3.19}$$

Here, N is the number of periods used in the EMA (e.g., 12, 26, or 9).

Trading signals generate when:

$$\text{Action}_t = \begin{cases} \text{Buy} & MACD_t > Signal_t \\ \text{Sell} & MACD_t < Signal_t \\ \text{Hold} & \text{otherwise} \end{cases} \tag{3.20}$$

The 9/12/26 EMA configuration reduces noise but introduces lag in volatile markets.

3.8 Agents

3.8.1 Advantage Actor-Critic (A2C)

The Advantage Actor-Critic (A2C) is a policy-based reinforcement learning algorithm designed for stock trading decision-making. Unlike value-based methods like DQN, A2C directly learns a stochastic policy and a value function, enabling the agent to select trading actions (Buy, Sell, or Hold) based on learned action probabilities and state-value estimates.

The A2C model comprises two neural networks—an actor network and a critic network—sharing a common input layer that receives a representation of market conditions (such as stock prices, technical indicators, and volatility measures). The shared layers typically include two fully connected hidden layers with 128 units each and ReLU activation functions, capturing complex patterns in financial data. The actor network outputs a probability distribution over trading actions, while the critic network estimates the value of the current state (V(s)).

In an offline reinforcement learning setup, the A2C agent is trained using historical stock market data. At each time step, the agent observes the current market state extracted using a function such as *get_ state()* and samples an action from the actor's policy distribution. The selected action is executed in a simulated environment, and a reward is calculated based on stock performance, trading costs, and risk exposure. The resulting transition (s, a, r, ś) is used to update both networks.

The critic network is trained to minimize the Mean Squared Error (MSE) between the predicted state value and the discounted return:

$$\text{Loss}_{\text{critic}} = \big(V(s) - \big(r + \gamma V(s')\big)\big)^2$$

The actor network is updated using the advantage function, which measures how much better the selected action was compared to the average (baseline) expected return. The advantage is defined as:

$$A(s, a) = r + \gamma V(s') - V(s)$$

The actor loss is then defined using the policy gradient:

$$\text{Loss}_{\text{actor}} = -\log\left(\pi(a \mid s)\right) \cdot A(s, a)$$

where $\pi(a \mid s)$ is the probability of choosing action a in state s. A negative sign is used

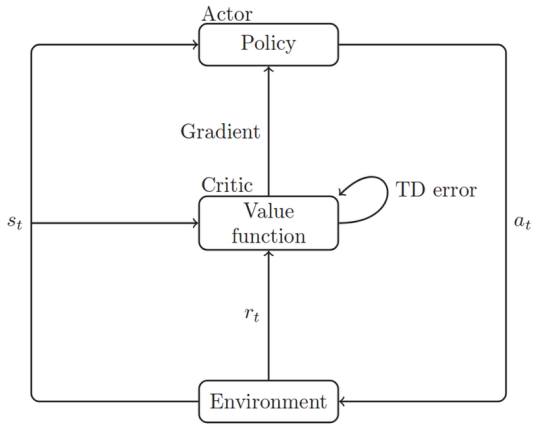


Figure 3.4: Working of Advantage Actor-Critic

because optimization algorithms minimize the loss.

To balance exploration and exploitation, A2C often includes an entropy bonus in the loss function to encourage exploration in the early training stages:

$$\text{Loss}_{\text{total}} = \text{Loss}_{\text{actor}} + c_1 \cdot \text{Loss}_{\text{critic}} - c_2 \cdot \text{Entropy}$$

where c_1 and c_2 are weighting coefficients.

As training starts after a warm-up period (e.g., day 60), allowing for robust market volatility estimation. Collected trajectories are used to compute advantages and returns, and the networks are updated using mini-batches of transitions.

Throughout training, performance metrics such as cumulative rewards, actor and critic losses, and entropy values are monitored. The cumulative reward indicates the profitability of the strategy, actor/critic losses reflect learning progress, and entropy values show the model's transition from exploration to a more deterministic policy.

By integrating value estimation (via the critic) with policy optimization (via the actor), and using the advantage function to reduce variance, A2C provides a stable and efficient framework for learning trading strategies that maximize long-term returns in a dynamic market environment.

3.8.2 Proximal Policy Optimization (PPO)

The Proximal Policy Optimization (PPO) algorithm is a policy-based reinforcement learning method designed to make stock-trading decisions. PPO directly learns a parameterized policy that outputs probabilities over actions (Buy, Sell, or Hold), allowing the agent to

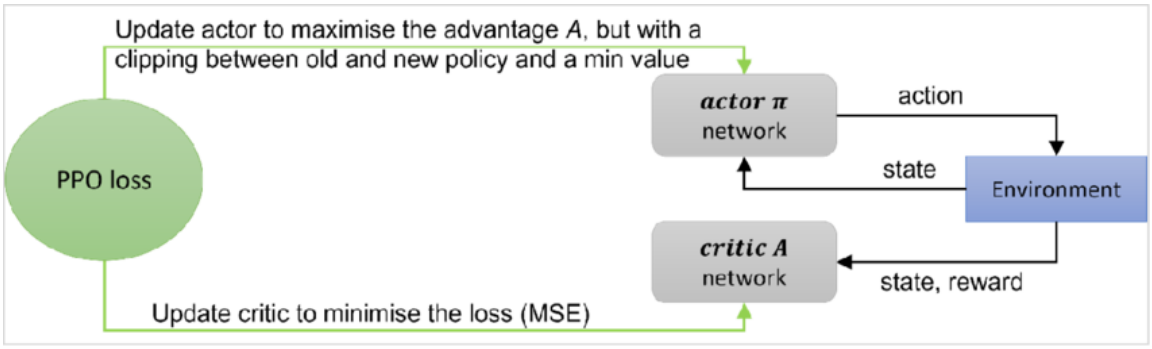


Figure 3.5: Working of the Proximal Policy Optimization

make decisions based on learned distributions rather than estimated value functions.

The PPO model architecture consists of an input layer that ingests a comprehensive state representation derived from market data (e.g., historical stock prices, technical indicators, and volatility measures), followed by two fully connected hidden layers with 128 units each using ReLU activation functions to capture nonlinear dependencies. The network then branches into two heads: one outputting the action probability distribution (policy head) and the other estimating the state value function (value head).

In an offline reinforcement learning setup, the PPO agent is trained on historical market data using pre-collected experience tuples (s, a, r, s'). These experiences are used to perform policy updates while ensuring training stability and preventing excessively large policy shifts. PPO achieves this by optimizing a clipped surrogate objective:

$$L^{\text{CLIP}}(\theta) = \hat{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{3.21}$$

where $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ is the probability ratio between the new and old policies, and $\hat{A}_t$ is the estimated advantage at time t. The clipping parameter ϵ (typically 0.1 to 0.3) ensures that updates do not push the policy too far from the previous one, providing stability.

The advantage function $\hat{A}_t$ is often estimated using Generalized Advantage Estimation (GAE), which balances bias and variance using a parameter λ. Meanwhile, the value head of the network is trained using the Mean Squared Error (MSE) between predicted and actual returns to improve state value estimation.

During training, actions are sampled from the policy distribution using an epsilon-greedy or stochastic sampling strategy. Early in training, higher exploration (via sampling and entropy regularization) allows the agent to discover diverse trading strategies. As training proceeds, the entropy bonus is annealed, encouraging more deterministic, profit-oriented decisions.

Each episode begins after an initial window (e.g., 60 days) to ensure the availability of reliable technical indicators like EWMA-based volatility. At every time step, the current market state is extracted via a function like get_state(), and the policy network outputs action probabilities. The selected action is executed, and the resulting reward—considering profits, transaction costs, and risk measures—is calculated. The transition (s, a, r, s') is stored and used in the subsequent batch update cycle.

Throughout training, key metrics such as cumulative reward, policy loss, value loss, and entropy are tracked. The cumulative reward reflects the profitability of the learned strategy, while policy and value losses indicate the learning progress and stability. Monitoring entropy values also provides insight into the agent's exploration behavior over time.

By optimizing a clipped objective function and incorporating stable policy updates, value estimation, and entropy regularization, the PPO agent effectively learns to navigate the dynamic nature of the stock market and execute trading actions that maximize long-term expected returns.

3.8.3 Deep Q-Network (DQN)

What is Deep Q-network (DQN)? The Deep Q-Network (DQN) is a reinforcement learning algorithm that combines Q-learning with deep neural networks to solve complex decision-making tasks. Traditional Q-learning updates a table of Q-values for each state-action pair, but this becomes impractical in environments with large or continuous state spaces. DQN addresses this by using a deep neural network to approximate the Q-function, mapping states and actions to their expected future rewards.

At each step, the agent interacts with the environment by observing the current state, selecting an action (using an -greedy policy for exploration), and receiving a reward and the next state. These transitions are stored in a replay buffer. During training, mini-batches of transitions are randomly sampled from the buffer to break the correlation between sequential samples and stabilize learning.

DQN also introduces a target network — a periodically updated copy of the main network — to compute stable target Q-values during training, further improving convergence. The network is trained by minimizing the mean-squared error between the predicted Q-values and target Q-values, based on the Bellman equation.

Overall, DQN enables agents to learn effective policies directly from high-dimensional inputs (like raw pixels), and has been foundational in achieving breakthroughs in tasks such as playing Atari games at a human level.

How Does the DQN Work? The Deep Q-Network (DQN) works by combining the principles of **Q-learning** with **deep neural networks** to allow an agent to learn how to act optimally in complex environments.

At each timestep, the agent observes the current state of the environment and uses the Q-network to predict the Q-values (expected future rewards) for all possible actions. The agent then chooses an action based on an ϵ-greedy policy: with probability ϵ, it explores by choosing a random action; otherwise, it exploits by choosing the action with the highest Q-value.

After taking an action, the agent receives a reward and transitions to a new state. This experience—consisting of the tuple (s, a, r, s') (state, action, reward, next state)—is stored in a **replay buffer**. Instead of updating the network immediately with consecutive samples (which could be highly correlated), DQN randomly samples mini-batches from this replay buffer. This process stabilizes learning by breaking the correlations between sequential experiences.

The **target Q-value** for each sample is computed using the **target network**, a separate copy of the Q-network that is updated periodically. The Q-network is trained by minimizing the **mean-squared error** between its predicted Q-values and the target Q-values, based on the Bellman equation:

$$Q_{\text{target}} = r + \gamma \max_{a'} Q(s', a')$$

where γ is the discount factor.

By repeating this process, the agent gradually learns an approximate Q-function that maps states and actions to expected returns, enabling it to act optimally.

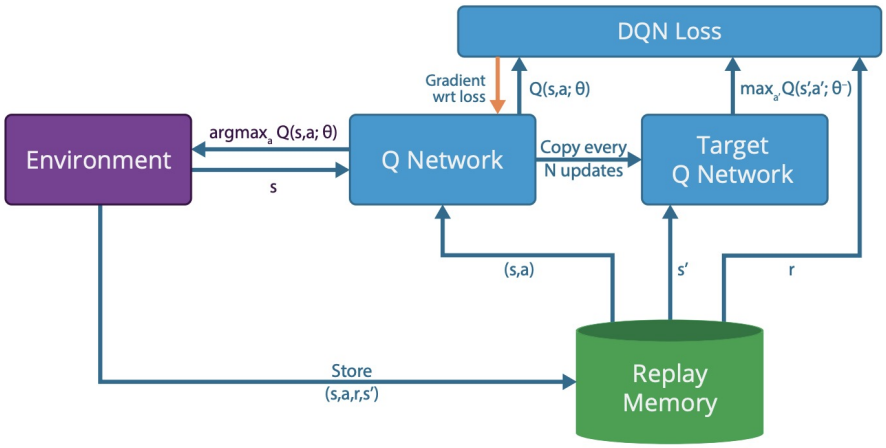


Figure 3.6: Architecture of the Deep Q-Network [3]

3.8.4 Decision Transformer

What is Decision Transformer? The Decision Transformer (DT) [4] is a neural network model that uses transformer-based architecture, originally designed for language tasks, to solve sequential decision-making problems commonly encountered in reinforcement learning (RL). Traditional RL methods train policies through trial-and-error learning in a live environment, which can be inefficient and sometimes infeasible in complex environments. Decision Transformer addresses this by framing RL as a sequence modeling problem, training on a dataset of offline trajectories instead of relying on direct interaction with the environment. DT combines past observations and actions with a target reward to determine optimal future actions, effectively leveraging the transformer's strengths in handling sequential dependencies. Its potential to work with offline datasets (pre-collected experiences) offers a way to bypass some limitations of standard RL, particularly in domains where exploring every possible action or state is impractical or costly, such as robotics, healthcare, and finance.

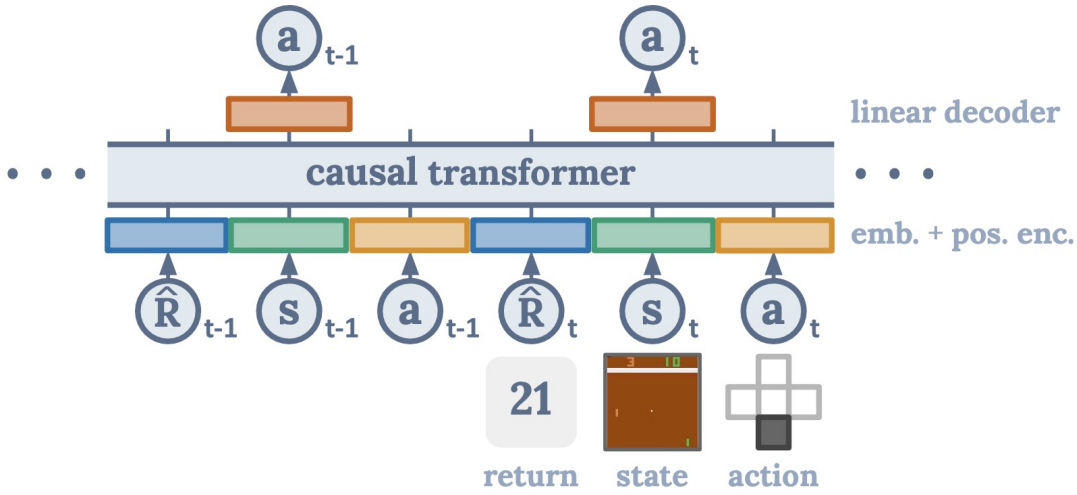


Figure 3.7: Architecture of the Decision Transformer [4]

How Does the Decision Transformer Work? Decision Transformer works by reframing RL as a conditional sequence prediction task. Given a goal (specified by the reward-to-go), the model is conditioned to predict the next best action based on the sequence of past actions, states, and rewards. The training data consists of past trajectories, which include sequences of states, actions, and corresponding rewards observed in prior interactions with the environment. During training, DT uses these trajectories to learn relationships between different actions and their outcomes, conditioned on the reward-to-go.

At each step, DT predicts the action that maximizes the reward when starting from the current state and aiming for the target reward. This approach relies on self-attention, which enables the model to capture long-range dependencies across sequences, making it effective in complex, delayed-reward scenarios. Once trained, the model can generalize by predicting actions for new initial states and desired rewards, effectively guiding the agent toward the highest possible reward in new situations.

The model conditions action selection on the desired return, adapting to different trading objectives dynamically.

The *return-to-go* (RTG) [15] at timestep t is computed as the cumulative future reward:

$$\hat{R}_t = \sum_{t'=t}^{T} r_{t'} \tag{3.22}$$

which serves as a conditioning signal. By specifying a desired target return $\hat{R}_0$ at the start, the model is guided to generate actions that aim to achieve that performance.

These elements are embedded into a shared latent space and interleaved to form a temporally ordered input sequence. Positional encodings are added to retain the order of events. The model then uses self-attention mechanisms to learn temporal dependencies, enabling it to predict actions that maximize cumulative future rewards based on historical context. The final output at each timestep is projected into action logits, and the model is trained using cross-entropy loss on ground-truth actions. Crucially, a causal attention mask is applied to ensure that each token only attends to previous tokens, preserving the autoregressive nature of the model and preventing information leakage from future timesteps during both training and inference.

Illustrative Example of Causal Masking:

Suppose $n = 3$, and for one head we compute the unmasked score matrix

$$S = \frac{QK^\top}{\sqrt{d_k}} = \begin{pmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{pmatrix}.$$

The causal mask is

$$M = \begin{pmatrix} 0 & -\infty & -\infty \\ 0 & 0 & -\infty \\ 0 & 0 & 0 \end{pmatrix}.$$

Adding gives

$$S + M = \begin{pmatrix} s_{11} & -\infty & -\infty \\ s_{21} & s_{22} & -\infty \\ s_{31} & s_{32} & s_{33} \end{pmatrix}.$$

After the softmax each row i normalizes only over the $\{j \leq i\}$ entries:

$$((S+M)_{i,*}) = \begin{cases} (1, 0, 0) & \text{if } i = 1, \\ \left(\frac{e^{s_{21}}}{e^{s_{21}}+e^{s_{22}}}, \frac{e^{s_{22}}}{e^{s_{21}}+e^{s_{22}}}, 0\right) & \text{if } i = 2, \\ \left(\frac{e^{s_{31}}}{\sum_{j=1}^{3} e^{s_{3j}}}, \ldots, \frac{e^{s_{33}}}{\sum_{j=1}^{3} e^{s_{3j}}}\right) & \text{if } i = 3. \end{cases}$$

Thus, each position i only attends to positions $1, \ldots, i$, enforcing causality.

Offline Reinforcement Learning Using Decision Transformer: Offline reinforcement learning (offline RL) is a subset of RL where the agent learns from a fixed dataset of experiences, as opposed to interacting with the environment. Decision Transformer's suitability for offline RL lies in its ability to learn directly from previously collected trajectories without needing online exploration. In offline RL using DT, the model is trained on sequences of state-action-reward pairs that represent past experiences. By modeling the sequence of these experiences, DT can predict the next optimal action based on a target cumulative reward, effectively mimicking a policy learned through typical RL methods. This approach is beneficial in high-stakes environments, like healthcare or autonomous driving, where unsafe or costly exploration can be avoided. DT can learn policies in offline RL scenarios by leveraging the reward-to-go (a measure of remaining rewards in a trajectory) as guidance, effectively simulating the outcome of actions without real-time experimentation. As a result, offline RL with DT offers a powerful solution for sequential decision-making in environments where collecting new data is impractical or expensive.

We implement this in a **offline RL** manner where the offline dataset(logged data) used for training the Decision Transformer consists of trajectories collected from a fixed set of episodes. Each trajectory contains sequences of states, actions, and rewards. For

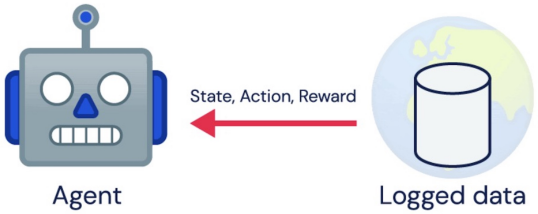


Figure 3.8: Offline RL

training, these trajectories are segmented into fixed-length sequences using a sliding window approach. The corresponding RTGs are computed for every timestep in the episode, and the resulting sequence of triplets:

$$\left(\hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, \ldots, \hat{R}_K, s_K, a_K\right) \tag{3.23}$$

is used as input to the transformer. During training, the model is optimized to predict the next action in the sequence, using either a cross-entropy loss (for discrete actions).

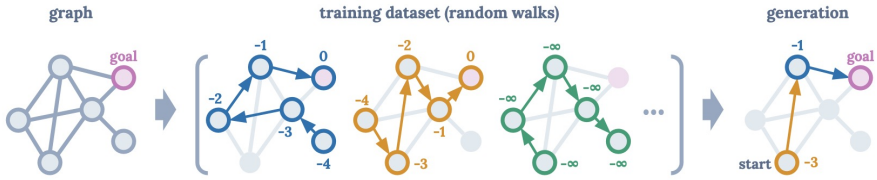


Figure 3.9: Illustration of shortest path finding as reinforcement learning: A fixed graph (left) with training data from random walks and per-node returns-to-go (middle). Decision Transformer generates optimal paths by maximizing returns.[4]

This design elegantly integrates return conditioning with the autoregressive generation of actions, enabling DT to effectively "plan" by targeting desired future returns while leveraging the powerful sequence modeling capabilities of transformers.

Work of Decision Transformer in Action Prediction Model to Increase Profit in Financial Market: In the stock market, predicting actions that maximize profits is a critical application of sequential decision-making. Decision Transformer can serve as an action prediction model by learning optimal buying and selling actions from historical market data. Here, the state inputs could include technical indicators, historical price movements, trading volume, and other market metrics, while the target reward could represent the profit objective or risk-adjusted return. The model is trained on past sequences of actions and market states to learn the most profitable patterns. Given a target profit (reward-to-go) and the sequence of past market data, DT can predict the next best

action—whether to buy, hold, or sell a particular stock. Since DT leverages offline datasets, it can be trained on extensive historical data without requiring risky live trading, reducing potential losses during the learning phase. In real-time trading, DT would take recent market data as input, along with a desired profit, and predict actions aligned with maximizing the expected profit. This approach is particularly advantageous in volatile markets, where identifying profitable patterns across various historical trajectories can give traders a strategic edge in making informed, data-driven decisions.

3.9 Model Architecture

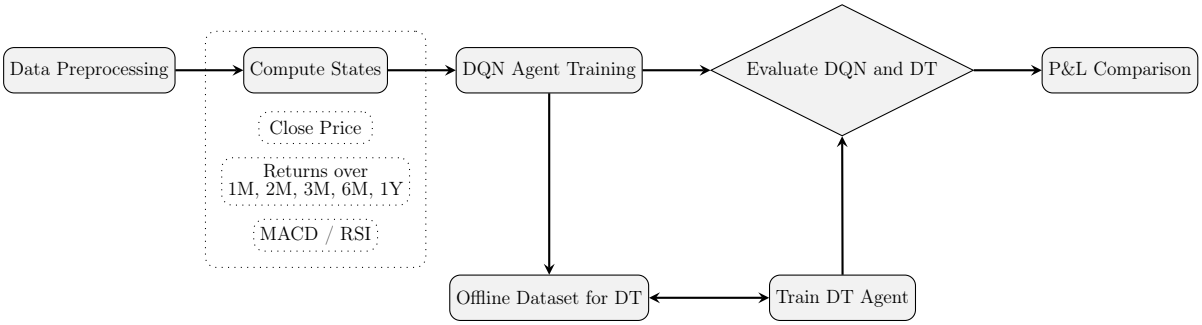


Figure 3.10: Architecture overview

As we can see from Figure 3.10, prior to model training, the data undergoes preprocessing to extract informative features. These include a range of technical indicators—such as the Moving Average Convergence Divergence (MACD) and Relative Strength Index (RSI)—along with multi-scale return computations (e.g., daily, weekly returns). All features are normalized to ensure consistent scale and efficient learning.

For state representation, each input sample is structured as a rolling window of 60 timesteps, where each timestep captures the full set of derived features. This temporal windowing allows models to learn historical patterns and dependencies in market behavior.

A Deep Q-Network (DQN) agent is initially trained using an epsilon-greedy exploration policy and experience replay buffer, simulating interactions with a trading environment. The DQN generates a dataset of offline trajectories—sequences of states, actions, and rewards—that serve as expert demonstrations.

These trajectories are then utilized to train a Decision Transformer (DT), which leverages the power of sequence modeling to map past market states and returns-to-go to optimal actions. Unlike the DQN, the DT learns offline, relying entirely on the quality of the collected trajectories.

We evaluate and compare the performance of the DT model, the DQN agent, and a traditional Buy-and-Hold strategy across all eight stocks. Evaluation metrics include cumulative profit and loss (P&L), cumulative rewards, action distributions, and decision interpretability. This comprehensive comparison helps assess both the financial effectiveness and behavioral robustness of the models in diverse market conditions.

3.9.1 Deep Q-Network (DQN)

Network Architecture: The Deep Q-Network (DQN) architecture is designed to approximate the action-value function Q(s,a) using a neural network. It typically consists of an input layer that receives the state representation, followed by one or more hidden layers with nonlinear activation functions, and an output layer that produces a Q-value for each possible action. For instance, a common configuration includes two fully connected hidden layers with 128 neurons each, utilizing ReLU activations. The output layer's size corresponds to the number of possible actions, providing the estimated Q-values that guide the agent's decision-making process. This architecture enables the agent to learn optimal policies directly from high-dimensional sensory inputs, such as images or sensor readings, by mapping states to action values through deep learning. The DQN is parameterized by a feedforward neural network $Q_\theta : R^d \rightarrow R^A$, where d is the dimension of the state feature vector and A is the number of discrete actions. As shown in Figure 3.6, the architecture of the Deep Q-Network consists of:

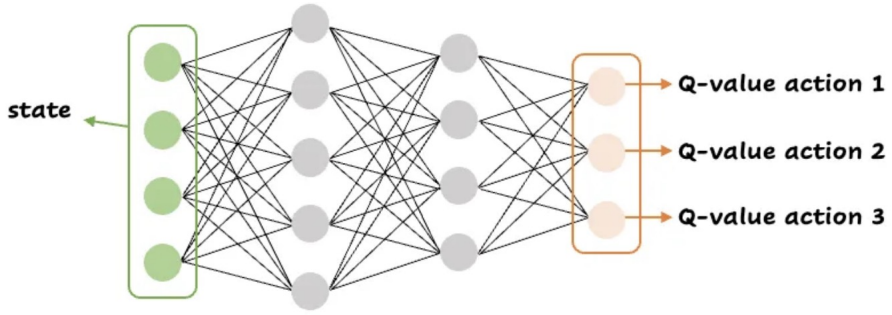


Figure 3.11: DQN network

DQN Architecture:

1. Input Layer: The input to the Deep Q-Network (DQN) is a vector representation of the current environment state. In the context of stock trading, this state may encapsulate various types of data such as historical stock prices, technical indicators (like RSI, MACD), statistical measures (such as moving averages or Bollinger bands), and even market sentiment metrics or volatility estimates. The goal is to transform raw market data into a form that the neural network can process to approximate the optimal Q-values. This preprocessed state vector captures the essential characteristics of the current market condition and serves as the foundation upon which the DQN makes action decisions (e.g., Buy, Sell, Hold).

2. Hidden Layers: The extracted features from the input layer are passed through two fully connected (dense) hidden layers, each consisting of 128 neurons. These layers utilize the Rectified Linear Unit (ReLU) as their activation function, which is defined as:

$$\text{ReLU}(x) = \max(0, x)$$

ReLU introduces non-linearity, allowing the network to capture and model complex relationships in high-dimensional input spaces. These layers are instrumental in abstracting the raw features into higher-level representations that are more relevant for learning value-based policies. The depth and width of these hidden layers provide the model with sufficient capacity to recognize temporal patterns and interdependencies present in financial data.

3. Output Layer: The final layer of the DQN outputs a vector of Q-values, each corresponding to a possible action the agent can take. For a trading agent, this typically includes actions like Buy, Sell, or Hold. The output vector size equals the number of discrete actions $|A|$. Each Q-value $Q(s, a)$ represents the expected cumulative discounted reward the agent will receive if it takes action a in state s, and then continues to act optimally. The policy is implicitly defined by choosing the action with the highest Q-value, i.e., $\pi(s) = \arg\max_a Q(s, a)$.

4. Experience Replay: As the agent interacts with the environment, it collects experiences in the form of tuples (s_t, a_t, r_t, s_{t+1}), which are stored in a replay buffer. During training, instead of using the latest experience, the agent samples mini-batches randomly from this buffer. This process breaks the strong correlation between consecutive transitions and improves training stability. Random sampling also enables better generalization since the network learns from diverse situations repeatedly. The replay buffer allows for efficient reuse of past experiences and makes the learning process more data-efficient. It is particularly valuable in reinforcement learning scenarios where data collection is expensive or time-consuming.

5. Target Network: A separate target network, parameterized by θ^-, is maintained alongside the primary Q-network (with parameters θ). The target network is a slowly updated copy of the primary network and is used to compute stable target Q-values during training. By periodically synchronizing the weights of the target network with the primary one (every C_{target} steps), the learning process avoids oscillations and divergence that would

arise from chasing constantly shifting targets. The target value for a given transition is:

$$y_t = r_t + \gamma \max_{a'} Q_{\theta^-}(s_{t+1}, a')$$

where $\gamma \in [0, 1]$ is the discount factor for future rewards. The use of this slowly evolving target helps reduce overestimation bias and improves convergence.

6. Q-Learning Update Rule: The DQN employs the Q-learning update rule to adjust its value estimates. For each transition sampled from the replay buffer, the TD (Temporal Difference) target is calculated using the target network. The core Q-learning update rule is expressed as:

$$Q_{\text{target}} = r + \gamma \max_{a'} Q_{\theta^-}(s', a')$$

The main Q-network predicts $Q_\theta(s, a)$, and the goal is to bring this estimate closer to Q_{target}. This is achieved by minimizing the error between the prediction and the target, thus refining the policy through repeated exposure to diverse samples. The max operator in the target encourages the policy to prefer actions that lead to higher long-term returns.

7. Loss Calculation: To quantify the error between the predicted Q-values and the TD target values, the DQN uses the Mean Squared Error (MSE) loss function. Given a mini-batch of size N, the loss is computed as:

$$\text{Loss} = \frac{1}{N} \sum_{i=1}^{N} (Q_\theta(s_i, a_i) - Q_{\text{target},i})^2$$

This loss function penalizes large deviations and provides a smooth optimization landscape. It ensures that the network learns to approximate the Q-function accurately over time. By minimizing this loss iteratively, the model aligns its predicted Q-values with the expected discounted returns observed from experience.

8. Optimization Step: The network parameters θ are optimized using stochastic gradient descent techniques such as the Adam optimizer. After computing the loss for a mini-batch, gradients with respect to θ are calculated and used to update the parameters:

$$\theta \leftarrow \theta - \alpha \nabla_\theta L_{\text{DQN}}(\theta)$$

where α is the learning rate. This process enables the Q-network to incrementally refine its value function estimates. Choosing an appropriate optimizer and learning rate is crucial; a learning rate too high may cause divergence, while one too low may lead to slow convergence.

9. Action Selection (Exploration vs. Exploitation): To balance learning new information and leveraging existing knowledge, DQN uses an ϵ-greedy strategy. With probability ϵ, a random action is selected to encourage exploration, while with probability $1-\epsilon$, the agent selects the action that maximizes the Q-value. The exploration rate decays over time:

$$\epsilon \leftarrow \max(\epsilon_{\min}, \epsilon \cdot \epsilon_{\text{decay}})$$

This decay ensures that early in training, the agent explores broadly, while later it exploits learned policies. Proper tuning of ϵ, $\epsilon_{\min}$, and ϵ_{decay} ensures effective exploration without excessive randomness, ultimately improving long-term performance.

10. Training Monitoring: To ensure that the learning process is effective, key metrics are continuously monitored. These include cumulative rewards per episode, average loss per update step, and the decay curve of the exploration rate ϵ. Visualization of these metrics helps detect overfitting, stagnation, or instability during training. Moreover, periodic evaluation on validation data or in a separate environment ensures the policy generalizes well. Proper monitoring is essential for diagnosing training dynamics, verifying hyperparameter effectiveness, and validating the stability of the learned strategy in dynamic market conditions.

Algorithm 2 DQN Training Implementation

Input: price series $\{p_t\}_{t=1}^{T}$, features precomputed in $s_t \in R^d$
Initialize Q-network Q_θ, target network $Q_{\theta^-} \leftarrow Q_\theta$
Initialize replay buffer $\mathcal{B}$, $\epsilon \leftarrow \epsilon_0$
for epoch $= 1$ to N_{epochs} **do**
    Reset episode: $p_{t-2}, p_{t-1} \leftarrow p_1$, $a_{t-2}, a_{t-1} \leftarrow 0$
    for $t = 60$ to T **do**
        $s_{t-1} \leftarrow$ get windowed feature vector at time $t-1$
        if RandomUniform$(0,1) < \epsilon$ **then**
            $a_t \sim$ Uniform random action from $\{0, \ldots, A-1\}$
        else
            Compute Q-values: $q \leftarrow Q_\theta(s_{t-1})$
            Select action: $a_t \leftarrow \arg\max_a q[a]$
        end if
        Compute reward $r_t \leftarrow R(p_t, p_{t-1}, p_{t-2}, a_t, a_{t-1}, \ldots)$
        Store (s_{t-1}, a_t, r_t, s_t) in $\mathcal{B}$
        if $|\mathcal{B}| \geq B_{\min}$ **then**
            Sample minibatch $\{(s, a, r, s')\}$ from $\mathcal{B}$
            $y \leftarrow r + \gamma \max_{a'} Q_{\theta^-}(s', a')$
            $\mathcal{L} \leftarrow \frac{1}{B} \sum (Q_\theta(s,a) - y)^2$
            $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$
            if step $\%$ $C_{\text{target}} = 0$ **then** $Q_{\theta^-} \leftarrow Q_\theta$
            end if
        end if
        Update $\epsilon \leftarrow \max(\epsilon_{\min}, \epsilon \cdot \epsilon_{\text{decay}})$
    end for
end for

3.9.2 Decision Transformer (DT)

The Decision Transformer (DT) reframes reinforcement learning as a sequence modeling problem using a causal Transformer architecture. Rather than learning value functions or policies directly, DT conditions on past states, actions, and desired return-to-go to autoregressively generate the next action. This approach enables the use of powerful Transformer models developed for natural language processing and leverages large offline datasets without requiring online interaction.

Network Architecture: DT consumes input sequences consisting of states, return-to-go (RTG), and actions over a fixed context window L:

- $S = (s_t, s_{t+1}, \ldots, s_{t+L-1}) \in R^{L \times d_s}$ – sequence of states,
- $G = (g_t, g_{t+1}, \ldots, g_{t+L-1}) \in R^L$ – sequence of return-to-go values,
- $A = (a_t, a_{t+1}, \ldots, a_{t+L-1}) \in \{0, \ldots, A-1\}^L$ – sequence of actions.

As discussed in the previous section, each component is embedded into a common latent space of dimension m:

$$\begin{aligned} E_s &: R^{d_s} \to R^m && \text{(state embedding)} \\ E_r &: R \to R^m && \text{(return embedding)} \\ E_a &: \{0, \ldots, A-1\} \to R^m && \text{(action embedding)} \end{aligned}$$

A learned positional encoding $P \in R^{L \times m}$ is added to retain the temporal structure of the sequence. The final input to the Transformer is:

$$H = E_s(S) + E_r(G) + E_a(A) + P \in R^{L \times m} \tag{3.24}$$

This is processed through a masked (causal) Transformer encoder:

$$Z = \text{TransformerEncoder}(H) \in R^{L \times m} \tag{3.25}$$

Decision Transformer Architecture:

1. Input Tokens: The Decision Transformer (DT) models trajectories in reinforcement learning by converting them into sequences of input tokens. At each timestep, the model takes three distinct inputs: the return-to-go ($\hat{R}$), the current environment state (s), and the previous action taken (a). The return-to-go represents the remaining cumulative reward expected from that timestep onward, guiding the model toward high-reward behavior. The state encapsulates the agent's observation of the environment, while the action represents the decision made at that step. Each of these components is embedded into a shared latent space, forming a unified representation suitable for Transformer-based sequence modeling.

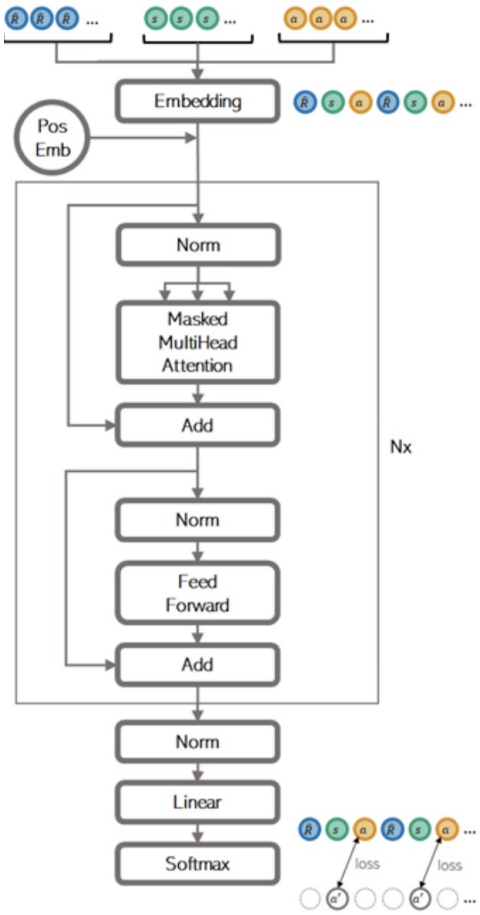


Figure 3.12: DT network architecture: inputs are embedded, summed with positional encodings, and passed through a causal Transformer.

2. Input Sequence Formation: To model trajectories as a temporal sequence, DT interleaves the embedded tokens for return-to-go, state, and action across timesteps. This results in an ordered input sequence like

$$(\hat{R}_{t-1}, s_{t-1}, a_{t-1}, \hat{R}_t, s_t, a_t, \dots)$$

where each group of three corresponds to one timestep. This interleaving ensures that the model can capture both intra-timestep relationships (e.g., how the return and state relate to the resulting action) and inter-timestep dependencies (e.g., how previous actions and returns influence future ones). Position embeddings are added to each token to encode their position in the sequence and preserve temporal ordering.

3. Embeddings in Decision Transformer: The Decision Transformer (DT) adapts the Transformer architecture to reinforcement learning by tokenizing sequences of returns, states, and actions. Each of these components is embedded into a shared vector space before being processed by the Transformer.

4. Input Embedding: Each raw input modality is mapped into a shared d-dimensional embedding space via a learned linear projection, followed by LayerNorm:

$$\begin{aligned}
\mathbf{e}_t^R &= \text{LayerNorm}\big(W_R\, R_t + b_R\big), \quad W_R \in R^{d\times 1},\ b_R \in R^d,\\
\mathbf{e}_t^S &= \text{LayerNorm}\big(W_S\, S_t + b_S\big), \quad W_S \in R^{d\times \text{state_dim}},\ b_S \in R^d,\\
\mathbf{e}_t^A &= \text{LayerNorm}\big(W_A\, \mathbf{1}\{A_t\} + b_A\big), \quad W_A \in R^{d\times |\mathcal{A}|},\ b_A \in R^d,
\end{aligned}$$

where $\mathbf{1}\{A_t\}$ is the one-hot encoding of action A_t. These produce token embeddings $\mathbf{e}_t^R, \mathbf{e}_t^S, \mathbf{e}_t^A \in R^d$.

5. Positional Embedding: To inject information about the ordering of the $3K$ input tokens, we learn a distinct embedding vector for each position $i = 1, 2, \dots, 3K$. Each positional embedding is a vector $P_i \in R^d$, where d is the model's embedding dimension. These vectors are treated as learnable parameters and are jointly optimized with the rest of the model during training. For a sequence of input embeddings $\{e_1, e_2, \dots, e_{3K}\}$, where $e_i \in R^d$ is the modality embedding (e.g., return, state, or action) at position i, the final input to the Transformer at each position is computed as:

$$z_i = e_i + P_i, \quad \text{for } i = 1, 2, \dots, 3K.$$

Stacking all positional embeddings forms a matrix $P \in R^{3K\times d}$, and the full input embedding matrix $Z \in R^{3K\times d}$ is given by:

$$Z = E + P,$$

where $E \in R^{3K \times d}$ is the matrix of modality embeddings $[e_1^\top; e_2^\top; \ldots; e_{3K}^\top]$.

In the attention computation, these positional embeddings directly affect the query-key dot products. Let $Q = ZW^Q$, $K = ZW^K$, and $V = ZW^V$ be the query, key, and value matrices obtained via learned projections with weights $W^Q, W^K, W^V \in R^{d \times d_k}$. The self-attention output is:

$$\text{Attention}(Z) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V.$$

Because Z includes the positional embeddings, the dot product $QK^\top$ incorporates relative position information:

$$QK^\top = (EW^Q + PW^Q)(EW^K + PW^K)^\top,$$

which expands to four terms involving both modality and positional components. This shows that positional information influences attention scores both directly and through interactions with content embeddings.

Furthermore, because P is optimized during training, the model learns how to use positional information adaptively for the downstream task—whether to emphasize nearby positions (short-range dependencies) or to generalize over longer temporal contexts.

6. How Does a Feedforward Neural Network Work? A **Feedforward Neural Network (FNN)** is the simplest type of artificial neural network in which information flows in one direction—from the input layer, through one or more hidden layers, to the output layer—without any cycles or loops.

Each layer in the network consists of a set of neurons (also called units). Each neuron computes a weighted sum of its inputs, adds a bias term, and passes the result through a non-linear activation function. Mathematically, the output $\mathbf{h}^{(l)}$ of the l-th layer is given by:

$$\mathbf{h}^{(l)} = f\left(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)}\right)$$

where:

- $\mathbf{h}^{(l-1)}$ is the input to layer l,
- $\mathbf{W}^{(l)}$ is the weight matrix of layer l,

- $\mathbf{b}^{(l)}$ is the bias vector of layer l,
- $f(\cdot)$ is the activation function (e.g., ReLU, sigmoid, tanh).

The final layer (output layer) produces predictions. For classification tasks, a softmax activation is often applied to obtain class probabilities:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

Training is performed using a dataset of input-output pairs. The predicted outputs are compared with true labels using a loss function (e.g., cross-entropy or mean squared error), and the network's weights are updated using backpropagation and gradient descent.

Through multiple iterations over the training data, the feedforward network learns to approximate complex functions that map inputs to outputs.

7. Feeding into the Transformer We then stack the sequence of token representations

$$[\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_{3K}]$$

and feed it into a causal Transformer model. A causal attention mask ensures that token i can only attend to tokens $1, \ldots, i-1$. The model is trained autoregressively to predict the next action token $\mathbf{e}_t^A$ given previous returns and states.

8. Linear Decoders: After passing through the transformer, the hidden states corresponding to particular tokens (typically state or return tokens) are extracted. These hidden states are fed into linear decoder layers to predict the next action a_t. Thus, the model predicts actions conditioned on the past returns, states, and actions.

9. Training Objective: During training, the predicted actions are compared against the ground-truth actions from the offline dataset. The model is optimized by minimizing the cross-entropy loss (for discrete action spaces) or mean-squared error loss (for continuous action spaces).

10. Autoregressive Action Generation: At each timestep i, the Transformer produces a hidden representation $Z_i \in R^m$, summarizing the encoded context of previous states, actions, and returns. This representation is passed through a linear projection layer $W_a \in R^{A \times m}$ to generate raw action logits:

$$z_i = W_a Z_i \tag{3.26}$$

A softmax function is applied to these logits to compute the probability distribution over the action space:

$$\hat{A}(i) = \text{softmax}(z_i) \tag{3.27}$$

This distribution defines the model's policy, producing a conditional probability $P(a_i \mid s_{1:i}, g_{1:i}, a_{1:i-1})$, which is used for selecting the next action based on prior observations and the return goal.

11. Causality nature of Decision Transformer: The GPT architecture, including variants such as the Decision Transformer, leverages the causal self-attention mask mechanism to ensure that each token's representation is only influenced by preceding tokens. This is essential for autoregressive modeling, where future information must not leak into the present.

The self-attention mechanism, as introduced [16], is defined as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{3.28}$$

Here:

- $Q \in R^{n \times d_k}$ is the matrix of queries,
- $K \in R^{n \times d_k}$ is the matrix of keys,
- $V \in R^{n \times d_v}$ is the matrix of values,
- d_k is the dimensionality of the keys.

In the GPT setting, a **causal mask** $M \in R^{n \times n}$ is applied to prevent attention to future positions. The modified attention becomes:

$$\text{MaskedAttention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}} + M\right)V \tag{3.29}$$

The causal mask M is defined element-wise as:

$$M_{ij} = \begin{cases} 0, & \text{if } i \geq j \\ -\infty, & \text{if } i < j \end{cases} \tag{3.30}$$

This masking ensures that the attention score for any position i does not incorporate information from positions $j > i$, thus preserving the autoregressive property.

This formulation prevents information leakage from future tokens during training.

12. Comparison of Multi-Head vs. Masked Multi-Head Attention:

	Multi-Head Attention	**Masked Multi-Head Attention**
Mask M	Not used (all positions attend freely)	$M_{ij} = 0$ if $i \geq j$, $-\infty$ if $i < j$
Attention scores	$\frac{Q_i K_i^\top}{\sqrt{d_k}}$	$\frac{Q_i K_i^\top}{\sqrt{d_k}} + M$
Causality	Bidirectional; peeks at future	Unidirectional; prevents peeking
Use-case	Encoder-to-encoder or encoder-to-decoder	Decoder self-attention in autoregressive models

Table 3.2: Comparison between standard multi-head attention and its masked (causal) variant.

13. Loss Function: The training loss is defined using cross-entropy between the predicted and ground-truth actions:

$$L_{DT}(\theta) = -\sum_{(S,G,A)\in D} \sum_{i=1}^{L} \log \hat{A}(i)[A(i)] \tag{3.31}$$

Here, $\hat{A}(i)[A(i)]$ denotes the predicted probability for the correct action $A(i)$. This loss promotes accurate imitation of the training trajectories and guides the model to learn effective action generation from high-return examples.

14. Optimization: The parameters θ of the Decision Transformer are optimized using the Adam optimizer:

$$\theta \leftarrow \theta - \eta \nabla_\theta L_{DT}(\theta) \tag{3.32}$$

Adam offers adaptive learning rates and momentum, which help the model converge efficiently. Proper regularization and learning rate scheduling are critical to prevent overfitting and ensure generalization.

15. Inference Strategy: During inference, given an initial return-to-go g_0, the model predicts actions in an autoregressive manner:

$$a_i^* = \arg\max_a P_\theta(a \mid s_{1:i}, g_{1:i}, a_{1:i-1}) \tag{3.33}$$

This allows the agent to generate behavior trajectories that are likely to achieve the target return g_0, by selecting the most probable action at each step based on the observed sequence.

Algorithm 3 Decision Transformer Implementation with Causal Masking

Embed & Model Init:
  State embed $E_s : R^d \to R^m$, RTG embed $E_r : R \to R^m$
  Positional embeddings $P \in R^{L \times m}$, Transformer encoder Enc, action head W_a
Build Offline Dataset:
for each episode $(s_{1:T}, a_{1:T}, r_{1:T})$ **do**
    Compute return-to-go: $g_t = \sum_{k=t}^{T} r_k$
    for $t = 1$ to $T - L + 1$ **do**
        Collect $(S_{t:t+L-1}, G_{t:t+L-1}, A_{t:t+L-1})$ into $\mathcal{D}$
    end for
end for
procedure FORWARD(S, G, A)
    Embed states: $H_s = E_s(S) \in R^{B \times L \times m}$
    Embed RTG: $H_g = E_r(G) \in R^{B \times L \times m}$
    Embed actions: $H_a = E_a(A) \in R^{B \times L \times m}$
    Interleave tokens: $H = \texttt{concat}(H_g, H_s, H_a) \in R^{B \times (3L) \times m}$
    Add positional embeddings: $H \leftarrow H + P$
    Apply causal mask in Transformer: $Z = \text{Enc}(H, \texttt{mask=causal})$
    Extract action positions: $Z_a = Z[:, 2 :: 3, :]$ ▷ every third token is an action
    Predict logits: $\hat{A} = \text{softmax}(W_a Z_a)$
    return $\hat{A}$
end procedure
procedure TRAIN$(\mathcal{D}, L, \alpha)$
    for $(\tau_s, \tau_a, \tau_r) \in \mathcal{D}$ **do**
        Compute discounted RTG: $g_t = \sum_{k=t}^{T} \gamma^{k-t} r_k$
        Slice context: $(S, G, A) \leftarrow (\tau_{s[t:t+L]}, \tau_{g[t:t+L]}, \tau_{a[t:t+L]})$
        Predict: $\hat{A} = \text{Forward}(S, G, A_{:L})$
        Compute loss: $\mathcal{L} = -\sum_{i=1}^{L} \log \hat{A}^{(i)}[A^{(i)}]$
        Update: $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$
    end for
end procedure
procedure EVALUATE$(s_{1:T})$
    Initialize context $C \leftarrow \emptyset$, RTG $g_0 \leftarrow G_{\text{init}}$
    for $t = 1$ to T **do**
        Observe s_t, update RTG: $g_t = g_{t-1} - r_{t-1}$
        Update context: $C \leftarrow C \cup \{(g_t, s_t, a_{t-1})\}$
        Pad if $|C| < L$: $\tilde{C} = \{\mathbf{0}\}^{L-|C|} \cup C$
        Separate $\tilde{C}$ into (G, S, A)
        Predict: $\hat{a}_t = \arg\max(\text{Forward}(S, G, A)^{(L)})$
        Execute $a_t = \hat{a}_t$, observe r_t
    end for
end procedure

3.10 Simulating Market Behavior

The environment offers a structured yet effective framework for simulating stock market behavior. In this setup, the trading agent interacts with historical price data, making sequential decisions based on the current market state at each time step. Through the reinforcement learning process, agents are trained to optimize their strategies, aiming to maximize cumulative returns while managing and minimizing associated risks.

This simulation methodology is instrumental for both developing trading algorithms and backtesting various strategies. By providing a controlled environment with clearly defined rules and dynamics, agents can systematically learn to navigate the complexities and uncertainties inherent in stock trading, with the ultimate objective of making profitable and robust decisions in real-world financial markets.

CHAPTER 4

EXPERIMENTS

4.1 Experimetnal Setup

The experimental framework is built upon historical daily stock market data from a diverse set of major publicly traded companies: Apple (AAPL), Amazon (AMZN), Google (GOOGL), Disney (DIS), IBM, Netflix (NFLX), Microsoft (MSFT), and Tesla (TSLA). Each dataset includes key market indicators such as the daily Open, High, Low, Close, Volume, and Open Interest values.

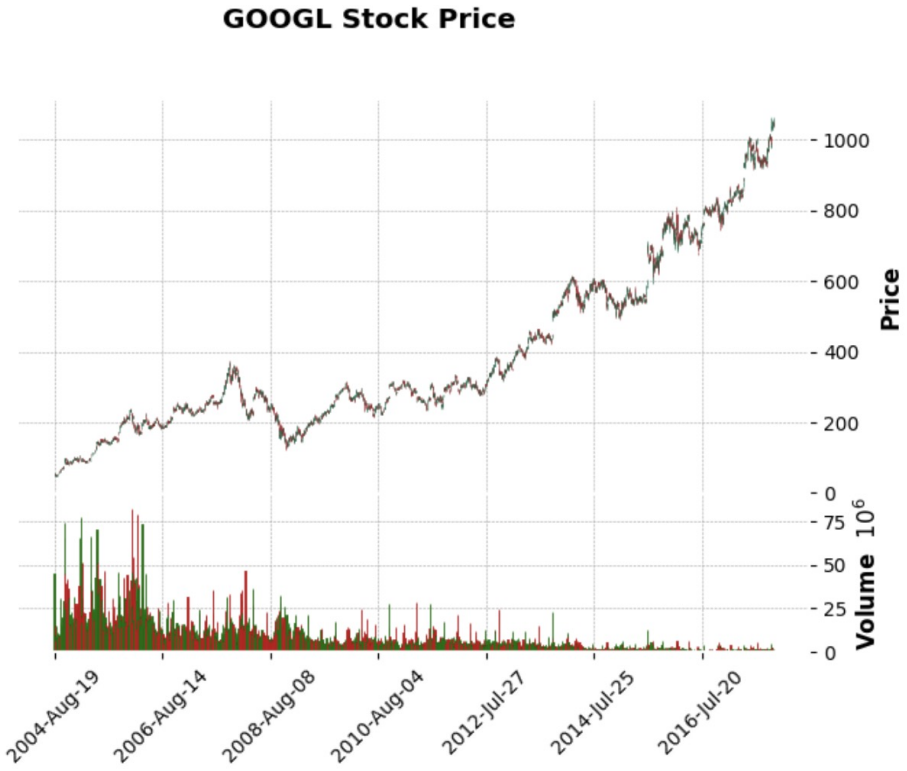


Figure 4.1: Candle stick diagram of GOOGL

4.1.1 Hyper Parameters

Hyperparameter	DQN Value
Learning Rate	1e-3
Discount Factor (γ)	0.99
Batch Size	64
Replay Buffer Size	10000
Target Update Frequency	500
Target volatility (σ_{tgt})	1
Epsilon decay (ε_{decay})	0.96
Cost term (C)	0.0001

Table 4.1: DQN Hyperparameters

Hyperparameter	DT Value
Learning Rate	1e-4
Number of Layers	6
Number of Heads	8
Embedding Dimension	128
Batch Size	128
Context Length	20
Weight Decay	1e-4
Dropout	0.1

Table 4.2: DT Hyperparameters

4.2 Results

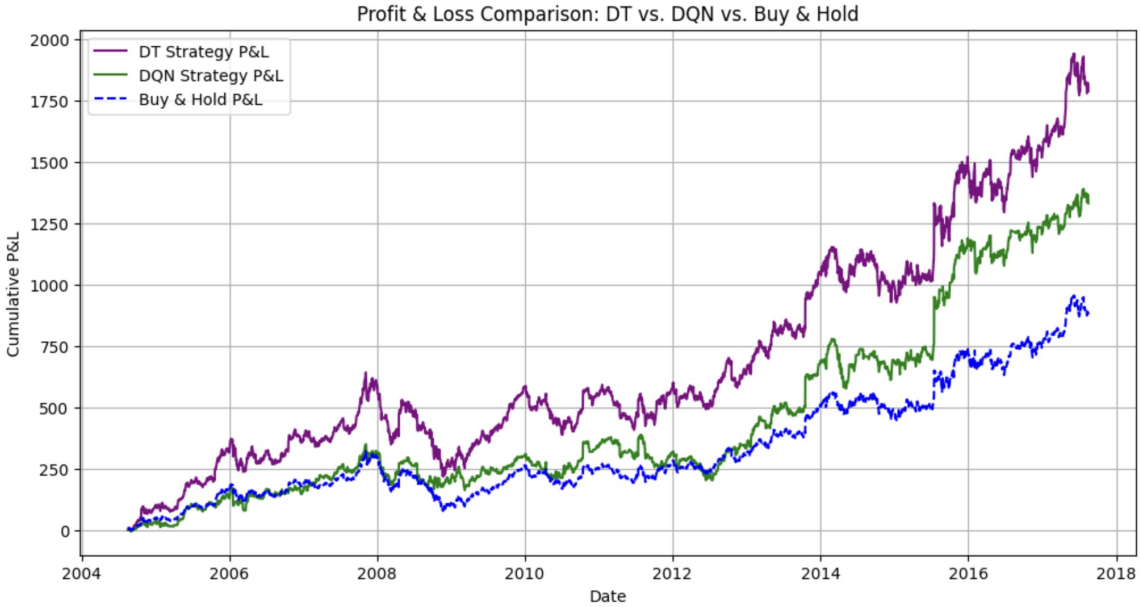
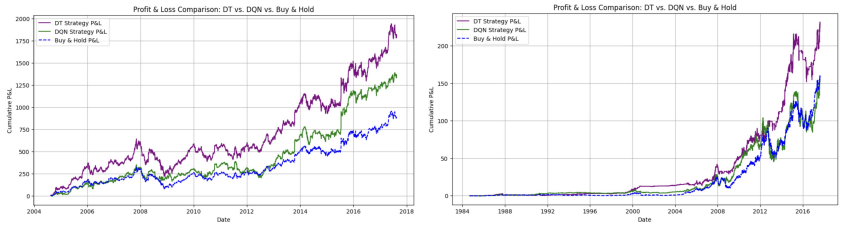


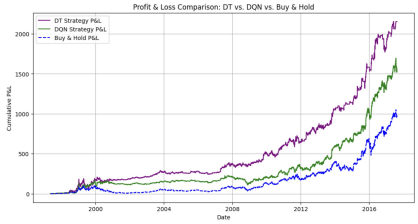
Figure 4.2: Profit and loss comparison

The Figure 4.2 illustrates a profit and loss comparison between the Decision Transformer model, DQN model, and the Buy and Hold strategy on Google stock data. The DT model performs better than both alternatives, achieving higher cumulative P and L over time, indicating its superior ability to optimize trading decisions and maximize returns.

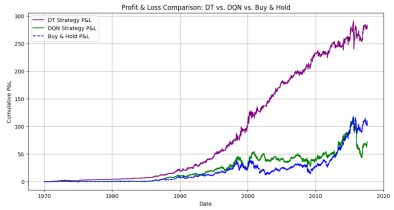


(a) GOOGL

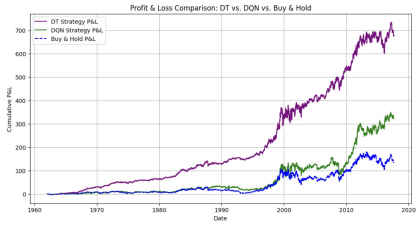
(b) AAPL



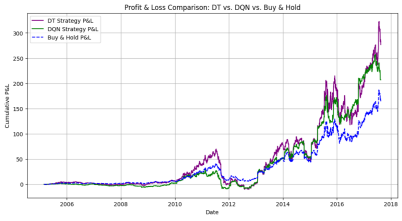
(c) AMZN



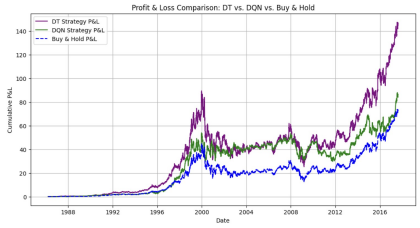
(d) DIS



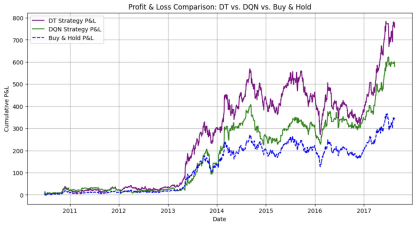
(e) IBM



(f) NFLX



(g) MSFT



(h) TSLA

Figure 4.3: Profit and loss graphs for GOOGL, AAPL, AMZN, DIS, IBM, NFLX, MSFT, and TSLA stocks.

In Figure 4.3, we present experiments across multiple stocks, comparing three strategies: the Decision Transformer (DT) model (Purple), the DQN model (Green), and the traditional Buy-and-Hold approach (Blue). The results demonstrate that the DT model consistently achieves superior performance over the other two methods across all stock datasets.

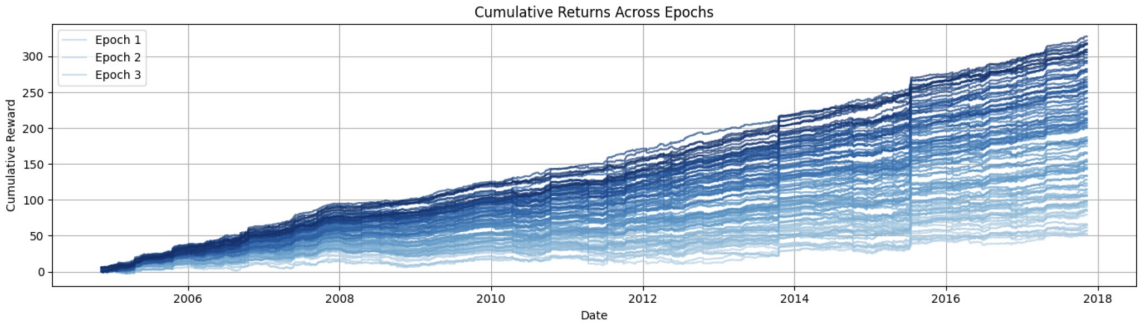


Figure 4.4: Rewards for 100 epochs

Above Figure 4.4, illustrates the cumulative rewards obtained by DQN model across multiple epochs. As the number of epochs increases, the color intensity of the graphs darkens, indicating improved performance and higher cumulative rewards over time, demonstrating the learning progression of the model.

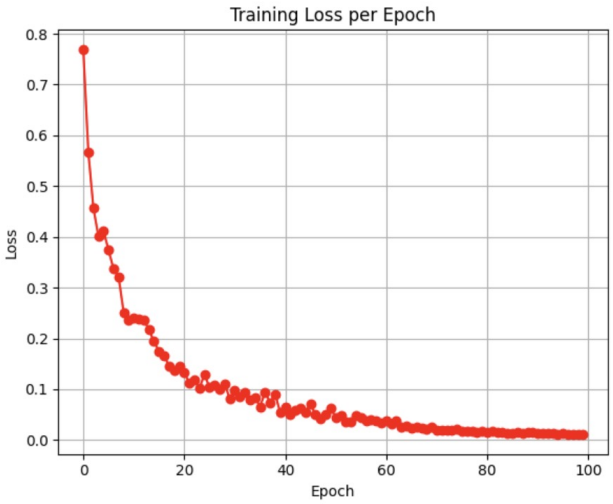


Figure 4.5: Loss curve for 100 epochs

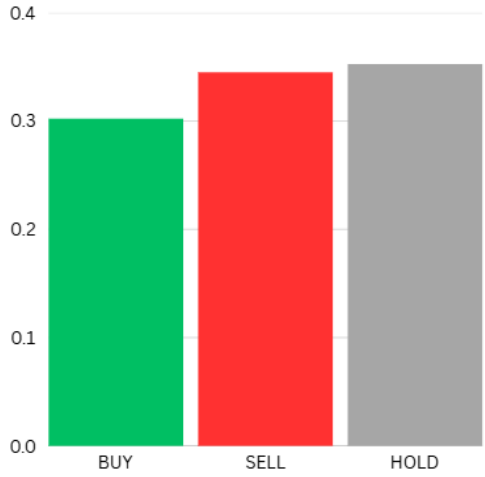


Figure 4.6: Action Distribution

In the Figure 4.5, The loss curve over 100 epochs shows a steady decline, indicating that the model is effectively learning patterns from the data. A smooth convergence suggests that the optimization process is stable, reducing the risk of overfitting and ensuring generalization to unseen data.

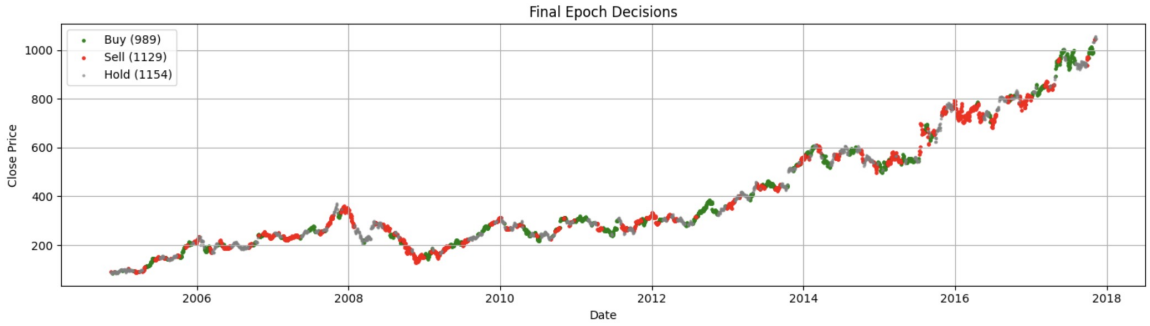


Figure 4.7: Actions taken by agent at each time step

Figure 4.7 shows a line chart depicting the Google stock's closing price from 2005 to 2018, overlaid with color-coded markers for the DQN model's buy, sell, and hold decisions. The model executed 989 buy, 1129 sell, and 1154 hold actions.

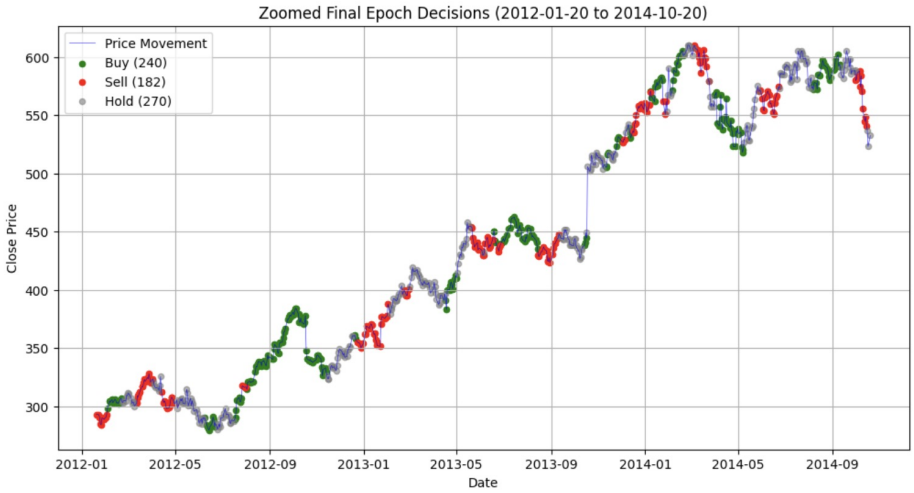


Figure 4.8: Zoomed in actions

Figure 4.8 offers an zoomed view of the DQN model's trading behavior. It distinctly marks buy actions with green dots, sell actions with red, and hold actions with grey. This detailed snapshot exposes subtle decision patterns and transitions, providing insight into how the model dynamically responds to market changes.

Figure 4.9: Magnified decisions

Figure 4.10: Cumulative Returns Over Time for Different Baseline Algorithms

Figure 4.10 This graph shows cumulative returns over time for different trading strategies, including PPO, A2C, DQN, and traditional baselines like Long Only, Sign(R), and MACD.

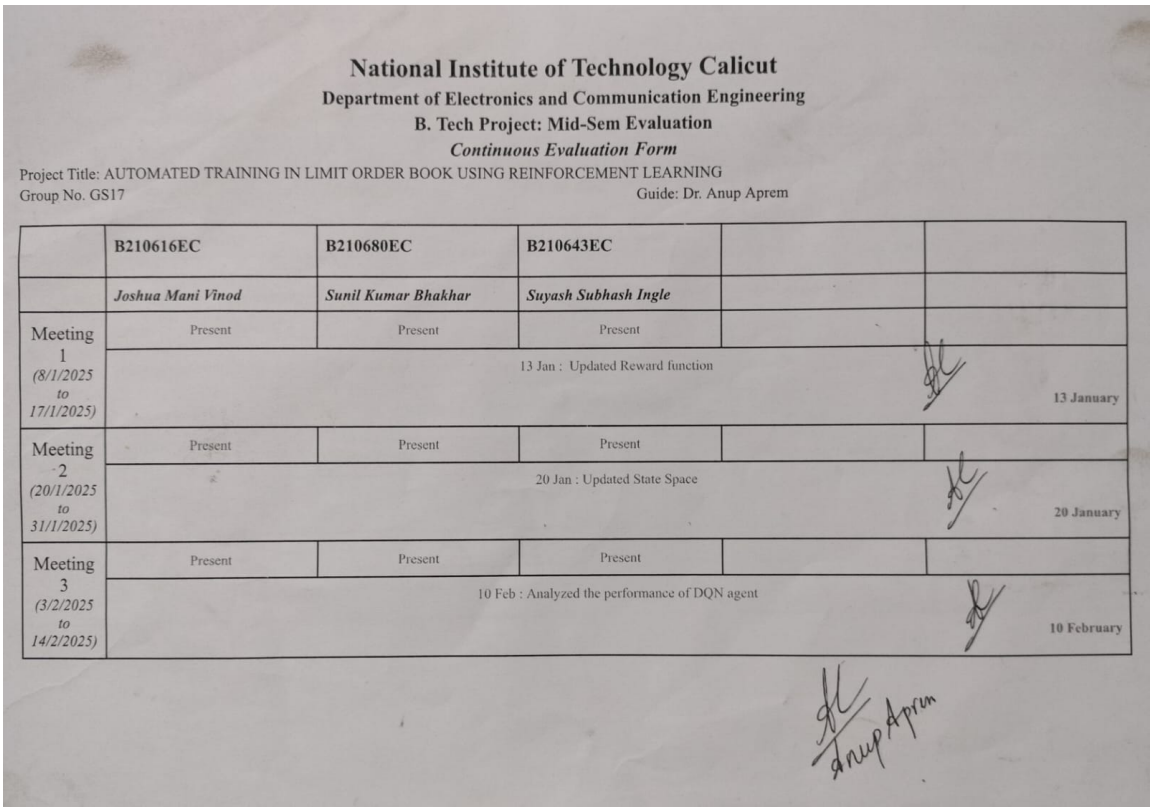
CHAPTER 5

CONCLUSION

Our results indicate that the Decision Transformer (DT) architecture outperforms the Deep Q-Network (DQN) approach in terms of cumulative returns. By leveraging an offline reinforcement learning framework, the DT effectively utilizes historical data to develop a more refined trading strategy. Its sequential learning capabilities enhance action selection and enable better adaptation to market dynamics, leading to optimal performance over traditional methods.

In our study, we also implemented and evaluated a Deep Q-Network (DQN) agent for stock trading, focusing on optimizing buy, hold, and sell decisions. The DQN successfully learned patterns in stock price movements, demonstrated stable convergence through its loss curve, and dynamically adjusted its actions based on market conditions across various stocks (GOOGLE, APPLE, MICROSOFT, AMAZON). This competitive performance highlights the agent's ability to identify profitable opportunities using historical data.

Overall, these findings underscore the potential of reinforcement learning-driven trading strategies. Future work may integrate advanced risk management techniques, reward shaping, and hybrid approaches, further enhancing the capabilities of both DT and DQN methodologies in developing sophisticated AI-based financial models.



National Institute of Technology Calicut

Department of Electronics and Communication Engineering

B. Tech Project: Mid-Sem Evaluation

Continuous Evaluation Form

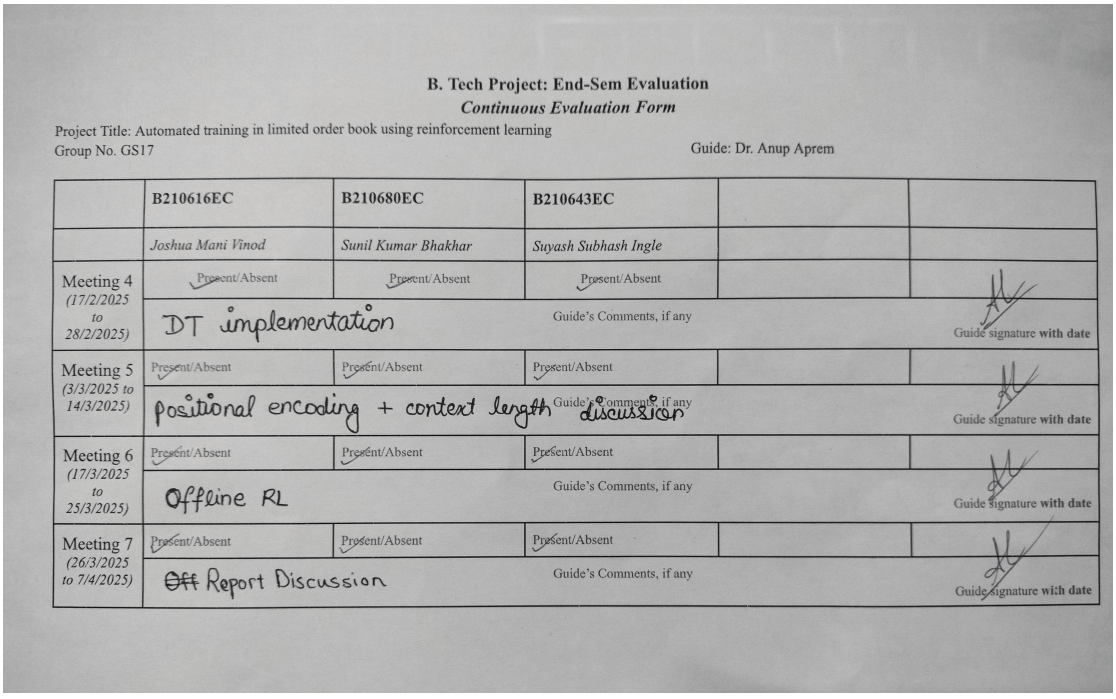
Project Title: AUTOMATED TRAINING IN LIMIT ORDER BOOK USING REINFORCEMENT LEARNING

Group No. GS17 Guide: Dr. Anup Aprem

	B210616EC	B210680EC	B210643EC		
	Joshua Mani Vinod	*Sunil Kumar Bhakhar*	*Suyash Subhash Ingle*		
Meeting 1 *(8/1/2025 to 17/1/2025)*	Present	Present	Present		
	13 Jan : Updated Reward function				13 January
Meeting 2 *(20/1/2025 to 31/1/2025)*	Present	Present	Present		
	20 Jan : Updated State Space				20 January
Meeting 3 *(3/2/2025 to 14/2/2025)*	Present	Present	Present		
	10 Feb : Analyzed the performance of DQN agent				10 February

Anup Aprem

Figure 5.1: CEF before mid-sem



B. Tech Project: End-Sem Evaluation

Continuous Evaluation Form

Project Title: Automated training in limited order book using reinforcement learning

Group No. GS17 Guide: Dr. Anup Aprem

	B210616EC	B210680EC	B210643EC		
	Joshua Mani Vinod	*Sunil Kumar Bhakhar*	*Suyash Subhash Ingle*		
Meeting 4 *(17/2/2025 to 28/2/2025)*	Present/Absent	Present/Absent	Present/Absent		
	DT implementation		Guide's Comments, if any		Guide signature with date
Meeting 5 *(3/3/2025 to 14/3/2025)*	Present/Absent	Present/Absent	Present/Absent		
	positional encoding + context length discussion		Guide's Comments, if any		Guide signature with date
Meeting 6 *(17/3/2025 to 25/3/2025)*	Present/Absent	Present/Absent	Present/Absent		
	Offline RL		Guide's Comments, if any		Guide signature with date
Meeting 7 *(26/3/2025 to 7/4/2025)*	Present/Absent	Present/Absent	Present/Absent		
	~~Off~~ Report Discussion		Guide's Comments, if any		Guide signature with date

Figure 5.2: CEF after mid-sem

BIBLIOGRAPHY

[1] H. T. Tse and H. fung Leung, "Exploiting semantic epsilon greedy exploration strategy in multi-agent reinforcement learning," 2022, accessed: 2023-02-18. [Online]. Available: https://arxiv.org/abs/2201.10803

[2] S. Ravichandiran, *Hands-On Reinforcement Learning with Python: Master Reinforcement and Deep Reinforcement Learning Using OpenAI Gym and TensorFlow.* Birmingham, UK: Packt Publishing, 2018.

[3] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. D. Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver, "Massively parallel methods for deep reinforcement learning," 2015.

[4] L. Chen, K. Lu, A. Srinivas, T. Darrell, and P. Abbeel, "Decision transformer: Reinforcement learning via sequence modeling," *arXiv preprint arXiv:2106.01345*, 2021.

[5] M. Xiao, Z. Jiang, L. Qian, Z. Chen, Y. He, Y. Xu, Y. Jiang, D. Li, R.-L. Weng, M. Peng, J. Huang, S. Ananiadou, and Q. Xie, "Enhancing financial time-series forecasting with retrieval-augmented large language models," 2025. [Online]. Available: https://arxiv.org/abs/2502.05878

[6] S. Peng and S. Yang, "Distributional uncertainty of the financial time series measured by g-expectation," *arXiv preprint arXiv:2011.09226*, 2020. [Online]. Available: https://arxiv.org/abs/2011.09226

[7] Z. Zhang, "Deep reinforcement learning for trading," in *Hierarchial modelling for financial data*. Wellington Square, Oxford OX1 2JD, UK: University of Oxford, 2020, pp. 76–92.

[8] C. Zhou, Y. Huang, K. Cui, and X. Lu, "R-DDQN: Optimizing algorithmic trading strategies using a reward network in a double DQN," *Mathematics*, vol. 12, no. 11, p. 1621, 2024. [Online]. Available: https://www.mdpi.com/2227-7390/12/11/1621

[9] S. Huang, A. Kanervisto, A. Raffin, W. Wang, S. Ontañón, and R. F. J. Dossa, "A2c is a special case of ppo," 2022. [Online]. Available: https://arxiv.org/abs/2205.09123

[10] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," in *arXiv preprint arXiv:1707.06347*, 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[11] C. Y. Huang, "Financial trading as a game: A deep reinforcement learning approach," in *Deep Reinforcement Learning in Financial Applications*. Wellington Square, Oxford OX1 2JD, UK: University of Oxford, 2024, pp. 76–92, accessed: 2023-02-18. [Online]. Available: https://arxiv.org/abs/1807.02787

[12] J. Baz, N. Granger, C. R. Harvey, N. Le Roux, and S. Rattray, "Dissecting investment strategies in the cross section and time series," *SSRN Electronic Journal*, 2015. [Online]. Available: https://ssrn.com/abstract=2695101

[13] J. W. Wilder, *New Concepts in Technical Trading Systems*. Greensboro, NC: Trend Research, 1978. [Online]. Available: https://archive.org/details/newconceptsintec00wild

[14] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *International Conference on Learning Representations (ICLR)*, International Conference on Learning Representations. OpenReview, 2016. [Online]. Available: https://arxiv.org/abs/1511.05952

[15] J. Schmidhuber, "Reinforcement learning upside down: Don't predict rewards - just map them to actions," *Advances in Reinforcement Learning*, 2021.

[16] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017. [Online]. Available: https://arxiv.org/abs/1706.03762

[17] L. Graesser and W. L. Keng, *Foundations of Deep Reinforcement Learning: Theory and Practice in Python*. Hoboken, NJ, USA: Pearson, 2020.

[18] J. Kang, R. Laroche, X. Yuan, A. Trischler, X. Liu, and J. Fu, "Think before you act: Decision transformers with working memory," *arXiv preprint arXiv:2306.12229*, 2023. [Online]. Available: https://arxiv.org/abs/2306.12229

[19] M. Siebenborn, J. Huang, B. Belousov, and J. Peters, "How crucial is transformer in decision transformer?" *Technical University of Darmstadt, Department*

of Computer Science, 2025, accessed: 2025-04-26. [Online]. Available: https://www.tu-darmstadt.de/

[20] Y. Zhuang, L. Liu, and C. Singh, "Learning a decision tree algorithm with transformers," *UC San Diego, Microsoft Research*, 2025, accessed: 2025-04-26. [Online]. Available: https://www.microsoft.com/en-us/research

[21] K.-H. Lee, O. Nachum, M. Yang, L. Lee, D. Freeman, W. Xu, S. Guadarrama, I. Fischer, E. Jang, H. Michalewski, and I. Mordatch, "Multi-game decision transformers," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022. [Online]. Available: https://arxiv.org/abs/2205.15241

[22] T. Shabeera, S. M. Kumar, and P. Chandran, "Curtailing job completion time in mapreduce clouds through improved virtual machine allocation," *Computers & Electrical Engineering*, vol. 58, pp. 190 – 202, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045790616305286

[23] M. P. Gilesh, S. D. M. Kumar, and L. Jacob, "HyViDE: A Framework for Virtual Data Center Network Embedding," in *Proceedings of the Symposium on Applied Computing*, ser. SAC '17. New York, NY, USA: ACM, 2017, pp. 378–383. [Online]. Available: http://doi.acm.org/10.1145/3019612.3019629