3. Algorithmic Trading Problem Formalisation

This section formalizes the problem of making optimal trading decisions in a systematic way using algorithms and reinforcement learning.

3.1. Algorithmic Trading

- **What is Algorithmic Trading?**
  Algorithmic trading refers to the process of using computer programs to make trading decisions automatically, based on pre-defined mathematical rules. These rules are designed to decide when to buy or sell financial assets like stocks, bonds, or currencies.
- **Quantitative Trading**: Another term sometimes used is "quantitative trading," which highlights the use of numerical and statistical methods to guide trading strategies.
- **Main Advantages**:
  - **Speed**: Since computers can make decisions faster than humans, algorithmic trading enables quick reactions to market changes.
  - **Liquidity**: Algorithms can help improve market liquidity. Liquidity refers to how easily an asset can be bought or sold without significantly affecting its price. Higher liquidity makes the market more efficient.
- **Markets Where It Can Be Applied**:
  - **Stock Markets**: Where shares of companies are traded (e.g., New York Stock Exchange).
  - **FOREX (Foreign Exchange)**: Trading of currencies.
  - **Commodity Futures**: Contracts to buy/sell a commodity at a future date.
  - **Cryptocurrencies**: Newer markets like Bitcoin and Ethereum provide new opportunities for algorithmic trading.
- **Portfolio Management**: The paper simplifies the trading problem by considering a portfolio with only one stock and cash. The trader (or algorithm) tries to manage this portfolio to maximize its value.
- **Order Book**: This is a list of all buy and sell orders in the market. It tells us who wants to buy or sell and at what prices. Traders need to make decisions based on this information to generate profit. A trade only happens when a buy order and a sell order match in price.

3.2. Timeline Discretisation

- **Continuous vs. Discrete Time in Trading**:
  In real life, trading happens continuously; there’s no specific “pause” between trades. However, in mathematical models (like the one used in this paper), we break this down into discrete time steps

to make it easier to analyze. Each step represents a moment in time when the algorithm makes a decision, like once every day.

- **Why Use Discrete Steps?**:
  If the algorithm makes decisions too frequently (like every second), it could become overly complex to handle. In this paper, the algorithm makes one trading decision per day, which is more manageable.
- **Time Step**: The duration between two decisions is called Δt. In this case, since the trading frequency is daily, Δt represents one day. Choosing Δt too short (like seconds or milliseconds) increases computational complexity, while choosing it too long may miss short-term opportunities.

3.3. Trading Strategy

- **How Does the Trading Strategy Work?**
  A trading strategy is a set of rules or instructions that the algorithm follows to decide whether to buy, sell, or hold an asset. These rules are based on the information available at any point in time, such as past prices, market trends, and other economic indicators.
- **Key Steps in a Trading Strategy**:
  i. **Information Update**: The algorithm first updates its knowledge of the current market (e.g., stock price, available cash, number of shares).
  ii. **Decision Making**: Based on this information, the algorithm decides whether to buy, sell, or hold shares. This decision follows a programmed policy (rule-based system).
  iii. **Execution**: The chosen action (buy, sell, or hold) is executed.
  iv. **Repeat**: The system moves to the next time step (usually the next day) and repeats the process.

  This process is **sequential**, meaning each decision depends on the result of the previous one. This is where reinforcement learning (RL) comes into play, helping the system learn which actions lead to better outcomes over time.

3.4. Reinforcement Learning Problem Formalisation

This subsection introduces how RL is used in algorithmic trading to optimize trading decisions. RL is a type of machine learning where an agent learns from interactions with its environment by receiving feedback (rewards or penalties).

3.4.1. RL Observations

- **What Does the Algorithm See?**:
  The **environment** in RL includes all the external factors that the trading agent interacts with, such as the stock market, financial news, and other traders' behaviors. However, the agent only has

partial information about this environment because some things are hidden (e.g., other traders' strategies or insider company information).

- **Key Observations**:
  The agent receives **observations** from the environment at each time step. These observations are the data that the agent can use to make decisions. In the context of trading, this includes:
  - **S(t)**: The state of the agent (e.g., how much cash and how many shares it owns).
  - **D(t)**: Market data, such as the Open, High, Low, Close prices, and Volume (OHLCV) for that day.
  - **T(t)**: Time-related information (e.g., what day it is).
  - **I(t)**: Technical indicators, like moving averages, relative strength index (RSI), and other metrics traders use to predict future price movements.
  - **M(t)**: Macroeconomic data (e.g., interest rates, inflation, exchange rates) that could affect market prices.
  - **N(t)**: News data from various sources (e.g., social media, financial reports), which can influence market sentiment and trading behavior.
- **Sequential Information**: The observations at time t are based not just on the current market but also on the agent's history of actions and past observations. This sequential nature makes the problem more complex, as the agent must remember past events to make better future decisions.

3.4.2. RL Actions

- **What Can the Agent Do?**:
  At each time step, the agent can take one of three possible actions:
  i. **Buy**: Purchase shares of the stock.
  ii. **Sell**: Sell shares of the stock.
  iii. **Hold**: Do nothing and keep the current position.
- **Action Modeling**: The quantity of shares bought or sold at time t is denoted as Qt. A positive value means the agent is buying, while a negative value means it is selling. If Qt = 0, the agent holds its position.
- **Trading System**: The agent interacts with an external system that handles the actual buying and selling, based on the number of shares it decides to trade.

3.4.3. RL Rewards

- **How Does the Agent Get Feedback?**:
  Rewards are given to the agent based on how well its decisions result in profitable outcomes. The primary reward is the **daily return** from trading, which measures the increase (or decrease) in the value of the agent's portfolio. This helps the agent learn which actions lead to the best long-term gains.

- **Formula for Reward**:
  The daily return is calculated as the difference in the portfolio value from one day to the next, normalized by the starting value.

$$r_t = \frac{v_{t+1} - v_t}{v_t}$$

  Where v_t is the portfolio value at time t, and v_{t+1} is the portfolio value at the next time step.

3.5. Objective

- **Sharpe Ratio as a Performance Metric**:
  The ultimate goal of a trading strategy is not just to make a profit but to make it in a **risk-adjusted** way. The Sharpe Ratio is a widely used metric to evaluate the performance of an investment strategy by considering both the return and the risk involved.

$$Sr = \frac{E[Rs - Rf]}{\sigma_r}$$

  - Rs is the return of the trading strategy.
  - Rf is the risk-free return (e.g., return from a government bond).
  - σ_r is the standard deviation (risk) of the trading strategy's returns.

  A higher Sharpe Ratio means the strategy has a better risk-adjusted return, which is a key objective in finance.
- **Maximizing Sharpe Ratio**:
  The agent's ultimate goal is to maximize the Sharpe Ratio across multiple markets. Although the RL algorithm directly maximizes daily returns, this is seen as closely related to maximizing the Sharpe Ratio in the long run. The researchers propose that future work could refine this connection to better align the RL rewards with maximizing the Sharpe Ratio.

4. Deep Reinforcement Learning Algorithm Design

This section is dedicated to explaining the design of the deep reinforcement learning (DRL) algorithm used to tackle the algorithmic trading problem. The main DRL algorithm used in this research is the **Trading Deep Q-Network (TDQN)**, inspired by the well-known Deep Q-Network (DQN).

4.1. Deep Q-Network Algorithm

The **DQN algorithm** is one of the foundational algorithms in deep reinforcement learning, designed to learn control policies from high-dimensional inputs. In simpler terms, it's an algorithm that can learn

how to act in environments by mapping states to actions using deep neural networks (DNNs).

- **Model-Free Nature**: DQN is referred to as **model-free**, meaning it doesn't require a full model of the environment to learn; instead, it learns from the "trajectory" of data it receives during training (i.e., historical price movements in this case).
- **Q-Function**: DQN learns a **Q-function**, which is an estimate of the value of taking a certain action in a given state. The Q-values guide the trading agent in making decisions about whether to buy, sell, or hold stock.
- **Off-Policy Learning**: DQN is **off-policy**, meaning it can learn from past experiences without requiring those experiences to come from the current policy. This allows the algorithm to learn from a batch of past experiences rather than needing to interact with the environment constantly.

The paper acknowledges that the DQN algorithm is foundational but adapts it significantly for the trading environment.

4.2. Artificial Trajectories Generation

In an ideal world, a trading agent would train on real-world stock market interactions. However, obtaining such data is difficult and limited, so the authors generate **artificial trajectories** to simulate more trading scenarios. Here's how they do it:

- **Historical Data as a Base**: The authors start with a set of historical stock market data (like daily prices) and consider this the "true trajectory" where the trading agent was inactive.
- **Creating New Trajectories**: Based on this historical data, the authors create **artificial trajectories**, where the trading agent performs various actions (like buying, selling) at different times. This allows the agent to learn without actually influencing the real market, which is an important constraint.
- **Assumption of No Market Influence**: For this technique to work, it is assumed that the actions of the agent (like buying or selling) do not affect the overall market. This is generally true when the agent's trading volume is small compared to the total liquidity in the market.
- **Exploration**: To improve the exploration of different trading possibilities, the authors use a technique where at each time step, the opposite action of the one taken by the agent is also executed in a copy of the environment. For example, if the agent buys shares, the simulation also tries a selling strategy in a separate "copy" of the market environment.

4.3. Diverse Modifications and Improvements

The original DQN algorithm was adapted with various modifications to make it more suitable for the trading environment:

1. **Deep Neural Network (DNN) Architecture**: The original DQN used convolutional neural networks (CNNs) since it was built for tasks like image processing. However, for financial data, the authors replaced the CNN with a feedforward DNN with leaky ReLU activation functions.
2. **Double DQN**: The DQN algorithm has a tendency to **overestimate** the values of certain actions, which can degrade performance. To combat this, the **Double DQN** variant was used, which separates action selection from action evaluation, reducing overestimation.
3. **Optimiser Change**: The optimiser was switched from **RMSProp** (used in the original DQN) to **ADAM**, which improves training stability and speed.
4. **Gradient Clipping**: A technique used to avoid **exploding gradients**, which can destabilize training, especially with deep networks.
5. **Xavier Initialization**: A method for initializing the weights in the neural network, ensuring that gradients maintain a steady variance across layers, which helps with convergence.
6. **Batch Normalisation**: This is used to improve training speed and generalisation by normalizing inputs to each layer of the network.
7. **Regularisation Techniques**: To prevent the model from overfitting to the training data, techniques like **Dropout**, **L2 regularisation**, and **Early Stopping** were implemented.
8. **Data Augmentation**: Given the limited amount of high-quality financial data, the authors applied techniques like **signal shifting**, **signal filtering**, and **adding noise** to artificially create more training data. These techniques helped the model generalize better.

5. Performance Assessment

This section explains how the authors evaluated the performance of their proposed TDQN algorithm. Performance assessment is crucial in algorithmic trading because the profitability of a strategy isn't the only metric — the associated risks must also be considered.

5.1. Testbench

The authors designed a **testbench** to rigorously assess their algorithm. Here's how:

- **Multiple Stocks**: Instead of evaluating the algorithm on a single stock (which may not give a reliable result), the authors used a set of **30 different stocks** from various sectors (technology, financial services, energy, automotive, etc.) and regions (America, Europe, Asia).
- **Time Period**: The time period for evaluation covered **8 years** from 2012 to 2019. They divided this into a **training set** (2012-2017) and a **test set** (2018-2019), ensuring that the model learned on past data and was evaluated on future (unseen) data.

- **Fixed Hyperparameters**: To ensure fairness and reduce overfitting, the authors did not change the algorithm's hyperparameters across the different stocks. This was done to see how well the algorithm generalised across diverse markets.

5.2. Benchmark Trading Strategies

To properly assess the TDQN algorithm, the authors compared its performance with some standard trading strategies:

1. **Buy and Hold (B&H)**: A passive strategy where the trader simply buys a stock and holds it for the entire trading period.
2. **Sell and Hold (S&H)**: A passive strategy that involves shorting (betting against) the stock and holding this position.
3. **Trend Following (TF)**: An active strategy where trades are made based on the detection of trends in the stock's price (e.g., buying during an upward trend).
4. **Mean Reversion (MR)**: Another active strategy, but opposite to trend-following, where the assumption is that stock prices tend to revert to their average levels over time.

Each of these benchmark strategies has different risk profiles, and by comparing them to the TDQN algorithm, the authors could better understand where their model stood.

5.3. Quantitative Performance Assessment

Finally, the authors describe the metrics they used to quantify the performance of the trading strategies:

1. **Sharpe Ratio**: Measures risk-adjusted returns. A higher Sharpe ratio means the strategy is earning more returns per unit of risk.
2. **Profit and Loss (P&L)**: The total profit or loss generated by the strategy.
3. **Annualised Return**: The yearly average return from the strategy.
4. **Annualised Volatility**: The riskiness of the strategy, measured as how much the strategy's returns fluctuate over time.
5. **Profitability Ratio**: The percentage of trades that were profitable.
6. **Profit and Loss Ratio**: The average profit made on winning trades compared to the average loss on losing trades.
7. **Sortino Ratio**: Similar to the Sharpe ratio but only considers downside risk (penalising the strategy more for negative returns).
8. **Maximum Drawdown**: The largest loss from a peak to a trough during the trading period.
9. **Maximum Drawdown Duration**: The longest time taken to recover from a drawdown.

These metrics allowed the authors to rigorously compare the TDQN algorithm with other strategies and determine how well it performed in both profitable and risk-adjusted terms.

This detailed breakdown of **Sections 4 and 5** covers both the algorithm design and performance assessment, highlighting the thought process behind the adaptations made to DQN for trading and how the algorithm's success was measured.

The pseudocode in **Section 4** of the paper outlines the main steps of the **TDQN (Trading Deep Q-Network)** algorithm. Let's break it down and explain each step in detail.

TDQN Algorithm Pseudocode Breakdown

Initialisation:

1. **Initialise the experience replay memory M of capacity C**:
   - The **experience replay memory** is a buffer that stores past experiences (state, action, reward, next state) that the agent encountered during training. This allows the agent to learn from past experiences in a batch rather than needing to learn from each interaction with the environment in real-time.
   - **Capacity C** refers to the maximum number of experiences the memory can store. Once it is full, the oldest experiences are removed to make space for new ones.
2. **Initialise the main Deep Neural Network (DNN) weights θ (Xavier initialisation)**:
   - The TDQN algorithm uses a **Deep Neural Network (DNN)** to approximate the Q-function, which helps the agent decide the best action (buy, sell, or hold) based on the current state of the market.
   - θ are the weights of the neural network, and they are initialized using the **Xavier method**, which is designed to keep the variance of the gradients consistent across layers, helping the network converge faster.
3. **Initialise the target DNN weights $\theta^- = \theta$**:
   - The **target network** is a copy of the main DNN but is updated less frequently to provide more stable learning. The target network uses θ^- as its weights, which are initially set equal to the main network's weights (θ).
   - This is a common trick in DRL algorithms to stabilize learning. The main DNN continuously updates while the target network is updated less frequently, reducing fluctuations in the Q-value estimates.

4. **For episode = 1 to N do**:
    - This loop starts the training process. The agent will go through N **episodes**, where each episode represents a full trading period (e.g., a series of days).

Within Each Episode:

5. **Acquire the initial observation $o1$ from the environment E and preprocess it**:
    - **Observation $o1$** is the initial state information the agent gets from the environment (such as stock prices, technical indicators, etc.).
    - **Preprocessing** might include normalization, removing noise, or transforming the data to make it easier for the DNN to process.

Time-Step Loop:

6. **For t = 1 to T do**:
    - At each time step t (which could represent a day in the trading process), the agent will go through the following steps to decide what action to take.
7. **With probability ε, select a random action at from action space A. Otherwise, select $at = arg\ max_{a \in A}\ Q(ot, a; \theta)$**:
    - **ε-greedy policy**: The agent uses an **exploration-exploitation** strategy, which means that:
        - With probability ε, it **explores** by selecting a random action from the action space A (e.g., buying, selling, or holding). This helps the agent discover new actions that might lead to better outcomes.
        - With probability $1-\varepsilon$, the agent **exploits** what it has already learned by selecting the action that maximizes the Q-value given by the DNN (based on the current observation ot and the weights θ).
    - Over time, ε decreases, meaning the agent explores less and exploits more as it becomes more confident in its learned policy.
8. **Copy the environment $E- = E$**:
    - The algorithm makes a copy of the environment (called $E-$). This is used for simulating the opposite action in parallel (which helps in exploration).

Interaction with the Environment:

9. **Interact with the environment `E` (action `at`) and get the new observation `ot+1` and reward `rt`**:
   - The agent takes the action `at` (e.g., buys, sells, or holds stock) in the environment. As a result of this action:
     - The environment returns a new state or **observation `ot+1`** (e.g., updated stock prices).
     - The agent receives a **reward `rt`**, which indicates how profitable or unprofitable the action was.
10. **Perform the same operation on `E-` with the opposite action `a-t`, getting `o-t+1` and `r-t`**:
    - The same action is performed in the copy of the environment `E-`, but with the **opposite action** of what the agent chose.
    - For example, if the agent bought shares in the real environment `E`, in the copied environment `E-`, it will sell shares. This is a trick used to ensure better exploration during training.

Store Experience and Replay:

11. **Preprocess both new observations `ot+1` and `o-t+1`**:
    - Before storing the new experiences, the observations from both the real and copied environments are preprocessed to make them suitable for the DNN.
12. **Store both experiences `et = (ot, at, rt, ot+1)` and `e-t = (ot, a-t, r-t, o-t+1)` in memory `M`**:
    - The agent stores two experiences in the **replay memory**:
      a. The real experience `et` from the actual action it took.
      b. The artificial experience `e-t` from the opposite action taken in the copied environment.
    - This helps the agent learn from both its actual actions and from hypothetical ones.

Training the Neural Network:

13. **If `t % T' = 0` then**:
    - Every `T'` **time steps**, the agent will update its neural network (rather than after every single step), which allows it to batch several experiences and train more effectively.
14. **Randomly sample from `M` a minibatch of `Ne` experiences `ei = (oi, ai, ri, oi+1)`**:
    - The agent randomly selects a batch of experiences from the memory. This random sampling breaks the correlation between consecutive experiences and improves learning.

15. **Set $y_i = r_i$ if the state s_{i+1} is terminal, otherwise $y_i = r_i + \gamma Q(o_{i+1}, \arg\max_{a \in A} Q(o_{i+1}, a; \theta); \theta^-)$** :
    - y_i is the target Q-value that the network is trying to predict for each experience in the minibatch.
    - If the next state s_{i+1} is terminal (i.e., the episode has ended), the target value is simply the reward r_i .
    - If it's not terminal, the target is the reward r_i plus the discounted future reward, where γ is the **discount factor** that determines how much the agent values future rewards.
    - The future reward is estimated using the **target network** with weights θ^- .
16. **Compute and clip the gradients based on the Huber loss $H(y_i, Q(o_i, a_i; \theta))$** :
    - The **Huber loss** is computed, which measures how far the predicted Q-value is from the target Q-value.
    - **Gradient clipping** is applied to prevent the gradients from becoming too large, which can destabilize training.
17. **Optimise the main DNN parameters θ based on these clipped gradients**:
    - The main DNN's weights θ are updated using the computed gradients, moving it closer to predicting the target Q-values.
18. **Update the target DNN parameters $\theta^- = \theta$ every N^- steps**:
    - Every N^- **steps**, the target network θ^- is updated to match the weights of the main network θ . This periodic update provides more stability during training, as the target values are more consistent between updates.

Exploration Decay:

19. **Anneal the ε-Greedy exploration parameter ε** :
    - As training progresses, the exploration parameter ε is **annealed** (gradually reduced). This means the agent will explore less and exploit more over time, relying more on what it has learned rather than random actions.

End of Episode:

20. **End of episode**: The process repeats for the next episode until all episodes are completed.

Summary of Key Ideas:

- The pseudocode follows the **DQN structure** with modifications like:
  - Generating **artificial trajectories** using a copy of the environment for better exploration.
  - Using a **target network** for more stable learning.
  - Applying techniques like **Huber loss**, **gradient clipping**, and **Xavier initialization** to improve training performance.

This structure allows the TDQN algorithm to efficiently learn from both real and simulated experiences, helping it develop a profitable trading strategy in a challenging, dynamic market environment.