Università
Ca'Foscari
Venezia

Master's Degree

in Economics and Finance

Final Thesis

Deep Q-Networks with a LSTM feature extractor for Algorithmic Trading

Supervisor:

Ch. Prof. Claudio Pizzi

Graduand:

Giuseppe Telatin

Matriculation Number 888060

Academic Year:

2023 / 2024

Abstract

This thesis investigates the advancement and utilization of a Deep Q-Network (DQN) incorporating a Long Short-Term Memory (LSTM) feature extractor for algorithmic trading. The suggested model seeks to identify temporal connections in financial time series data and improve decision-making in stock trading. We utilize LSTM to extract useful features from the time series and implement DQN to acquire effective trading strategies via reinforcement learning. The design combines the DQN's capacity to learn optimal policies with LSTM's proficiency in managing sequential data, allowing the model to make more educated trading decisions.

The methodology incorporates experience replay and employs two neural networks, one for online learning and another for target Q-values, to ensure training stability. Hyperparameter tuning is conducted with Optuna, and the model is optimized utilizing the Adam optimizer, incorporating Kaiming Normal weight initialization and layer normalization in the LSTM. We examine two reward functions, focusing not only on performance but also on the agent's risk aversion.

The methodology is assessed across different asset classes, including the S&P 500, gold, and specific stocks such as Disney and Intel, utilizing performance indicators such as the Sharpe Ratio, Sortino Ratio, and Maximum Drawdown for evaluation.

The model showed promising results, being able to generate profits; however, not consistently. This thesis continues the past research on a hybrid architecture that integrates advanced reinforcement learning with time series feature extraction, offering novel insights into the capabilities of deep learning models for financial trading.

Contents

List of Tables

List of Figures

Introduction

Financial markets display complexity and dynamism due to different factors like economic conditions, market sentiment, and global events, which all cause fluctuations in stock prices and other asset classes. To tackle this issue, algorithmic trading has emerged as a powerful tool for enhancing efficient and data-driven decision-making. This nondiscretionary approach enables the development of solutions that are able to identify profitable opportunities for executing trades quickly, thereby minimizing human biases and emotional impacts.

Recent discoveries in machine learning, particularly in deep learning and reinforcement learning, have generated new ways of improving the effectiveness of algorithmic trading methods. This thesis will carefully address reinforcement learning, focusing on Q-Learning, as it has emerged as a significant paradigm for trading applications. The advent of Deep Q-Networks (DQN), which combine the aforementioned Q-Learning with deep neural networks, has provided a method for approximating value functions in high-dimensional state spaces.

The efficacy of reinforcement learning in financial markets for a major part depends on the algorithm's ability to discover temporal correlations within the data. Nevertheless, financial time series data can exhibit temporal patterns and correlations that conventional models inadequately capture. The DQN framework integrates Long Short-Term Memory (LSTM) networks, a specific variant of recurrent neural networks (RNNs), to address this challenge.

LSTM networks excel at managing sequential data because of their ability to retain information over extended periods of time, making them ideal for financial forecasting and decision-making processes. This thesis explores the development of a hybrid trading algorithm that combines DQN with LSTM as a feature extractor. The LSTM component identifies temporal connections in financial time series, learning what to keep and what to discard. On the other hand, the DQN tries to learn the optimal trading policy via interaction with the market environment.

To gently introduce the reader to these architectures, we propose the perceptron as the first architecture. Next, we will discuss the Multi-Layer Perceptron (MLP), which will play the role of the Q-network for computing Q-values.

We will then explain Recurrent Neural Networks, beginning with the Elman Network.

The LSTM will then be carefully discussed, explaining all the operations inside a cell and the process for updating the weights of the architecture. Given that the reader will have all the instruments for learning how a Deep Q-network works. Before delving into the algorithm's specific architecture, we will discuss the use of experience replay to enhance the diversity of training data and stabilize the learning process. The model architecture section presents all the necessary passages, from layer normalization to He initialization, from a theoretical perspective, and concludes with the Adam optimizer.

Next, we carefully explain the experimental setup, highlighting the reward functions. One function solely focuses on maximizing profit, while the other takes into account the agent's risk aversion.

Finally, we evaluate the model on various financial instruments, including indices like the S&P 500 and individual stocks like The Walt Disney Company and Intel, using established financial performance metrics like the Sharpe Ratio, Sortino Ratio, Maximum Drawdown, and Trade Efficiency.

These measures provide a comprehensive assessment of the model's performance, particularly when compared to traditional trading strategies like Buy & Hold.

Literature Review

The foundations of this thesis are in the perceptron, which was proposed by Rosenblatt in the late fifties [1]. This architecture, extensively covered at the beginning of the Deep Learning section, served as a basis for the Multi-Layer Perceptron [2] that overcomed some limitations with respect to the previous one by introducting the hidden layers.
Another major block over which this thesis lies is the seminal papers on Deep Q-Networks by Mnih et al. [3]. This paper, even if does not uses the architecture for trading purposes, had a significant impact on deep reinforcement learning applications in trading. Indeed, as explained in the third chapter the authors demonstrated how a deep neural network can be used to approximate the Q-values in large state spaces.
Deep Q-Networks use Q-Learning, a model free algorithm which was introduced by Watkins in 1989 [4].
Focusing on the reinforcement learning side, Moody and Saffell [5] introduced the application on reinforcement learning to trading. Indeed, in their work the authors explored how reinforcement learning can improve trading strategies by learning through trial and error, maximizing returns while managing risk. The paper laid the groundwork for applying reinforcement learning to portfolio management and automated trading systems.
Afterwards, Nevmyvaka et al. [6] delved deeper into trading execution, a crucial component of a trading algorithm. On their paper they applies reinforcement learning to minimize transaction costs while improving execution quality. Going forward with the architectural complexity, Deng et al. [7] marked a significant advancement in the application of deep reinforcement learning to algorithmic trading. As a matter of fact, the authors proposed the use of deep RL for acquiring financial signal representations and trading strategies. This approach allowed a more sophisticated feature extraction which will be an important theme in the thesis. In fact, the DQN is associated to a LSTM feature extractor [8], a Recurrent Neural Network able to learn sequential data discarding redundant information and keeping the most important one.
When it comes to RNNs, Borovykh et al. [9] investigated the use of standard RNNs and LSTMs, for stock price prediction on the Dow Jones Index. The authors demonstrate how RNNs effectively capture temporal dependencies in financial data, leading to improved forecasting accuracy for stock prices. Among the papers that inspired this thesis there is the one made by Jiang et al. [10] which proposed a model not only for predicting

long/short signals but also the numbers of shares to trade, demonstrating how deep reinforcement learning can improve both action selection and trade volume, resulting in a more profitable algorithm.

The reward functions choice has been inspired by Zhiyi Zeng et al. [11]. The authors presented a novel approach for improving stock trading strategies by combining multi-step reward functions using thompson sampling. Li et al. [12] instead focused on developing a robust reinforcement learning framework for real trading scenarios. The authors did so by proposing a LSTM feature extractor that enriches the input to the Q-network with more informative features. Despite being a relatively recent architecture, DQN inspired the creation of even complex architectures as done by Cao et al. [13] which introduced the Double Deep Q-networks applying them to pairs trading, and by Théate and Ernst [14] that assessed the performance of the TDQN, a new DRL approach inspired from the popular DQN algorithm and significantly adapted to the specific algorithmic trading problem.

In conclusion, also Taghian et al. [15] showed the potential of Deep Q-networks in generating trading signals. In their paper various asset trading rules are proposed experimentally based on different technical analysis techniques with the trial of different architectures as feature extractors.

Chapter 1

Deep and Reinforcement Learning

1.1 Deep Learning

Deep learning has become a prominent method in machine learning during the past couple decades [16]. A defining characteristic of a deep architecture in neural networks is the inclusion of one or more hidden layers positioned between the input layer and the output layer. Hence, it is imperative to commence with the most elementary neural network, namely the perceptron, in order to comprehend the fundamental principles of this field.
After that, we will examine a more intricate feed-forward network known as the Multi-Layer Perceptron, in which the back propagation algorithm is of utmost importance.
Next, our attention will be directed towards recurrent neural networks. We will begin by examining the Simple Recurrent Network (also known as the Elman Network) and subsequently elucidate the Long Short-Term Memory Network.

1.1.1 Perceptrons

In 1958, Rosenblatt introduced the concept of perceptron, aiming to imitate the biological process responsible for learning in the brain [1]. In the brain neurons are connected to each other through axons and dendrites, which form synapses. Similarly, in the perceptron, we establish connections between an input layer and an output layer using specific weights; the output layer is then associated with an activation function.
The subsequent page presents an illustration of a perceptron.
In figure 1.1 the input layer is given by the row vector $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]$ with $x_j \in \mathbb{R}$. We can view each input as a node and assign a value of 1 to the first node, which serves as the basis for computing the bias (the role of the bias will be addressed later).
The weights are expressed as a column vector $\mathbf{w} = [w_0, w_1, w_2 \ldots w_n]^T$ with $w_j \in \mathbb{R}$. Each weight can be interpreted as the measure of the intensity of the connection between the respective input node and output node. Indeed, the greater the weight, the more significant the influence of the input feature on the ultimate outcome.

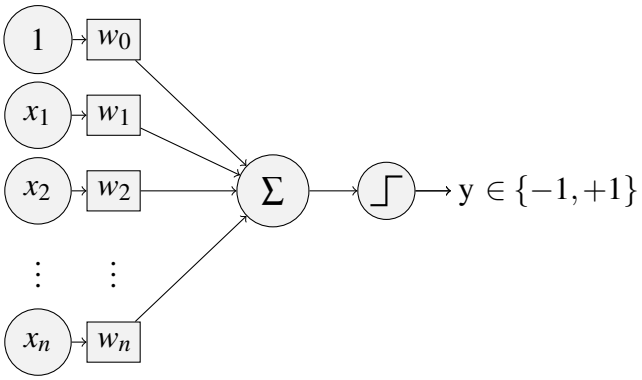

Figure 1.1: a perceptron with a sign activation function.

Usually, a solitary output node encompasses both the summation and the activation function. Nevertheless, we have distinguished them to enhance clarity. The raw output is calculated by taking the weighted total sum of the input layer, which includes the bias. The weighted sum can be expressed as the dot product between the row vector of weights and the column vector of inputs, as reported in equations (1.1) and (1.2).

$$y = \sum_{j=1}^{n} w_j x_j + w_0 \tag{1.1}$$

$$y = \mathbf{w}_i^T \cdot \mathbf{x}_i \tag{1.2}$$

Next, the value of the summation is processed by an activation function, in this case a sign function, which converts the output to either -1 or +1 based on the sign of the weighted sum.

Having familiarized ourselves with the fundamental structure of the perceptron, we can now address the inquiry: "What is the objective of a neural network?". The objective of a neural network is to acquire knowledge of a a function that relates one or more inputs to one or multiple outputs using training examples [17].

Thus, in the aforementioned example, when provided with training data, the perceptron is capable of learning solely linear functions, as evident from equation (1.1) when $j = 1$, or constructing a hyperplane if $j > 1$. In either scenarios, the perceptron will have the capability to execute a binary classification.

The perceptron can successfully complete its duty only if the data is linearly separable, indicating the presence of one or more lines or hyperplanes that can separate the data into two distinct classes. In order to tackle more difficult problems, it is necessary to employ more intricate architectures.

Now the significance of bias becomes more evident. Without a bias term, the decision boundary of the perceptron would always intersect the origin. However, when dealing with imbalanced and sparse data, the model's ability to accurately classify will be compromised. Therefore, the existence of a bias becomes essential.

It is currently necessary to discuss how the perceptron learns, beginning with a qualitative explanation. After receiving the input data and randomly initializing the weights, the perceptron executes the previously described procedure to generate an output. We then compare this output with the correct one. Subsequently, we compute a loss function, and through numerous iterations, our objective is to diminish the loss function by achieving the most accurate classification.

Bishop proposed the perceptron criterion as a loss function [18]; the formula is reported below in equation (1.3).

$$L_i = \max\left\{-y_i\left(\mathbf{w}_i^T \cdot \mathbf{x}_i\right), 0\right\} \tag{1.3}$$

Where L_i represents the loss of the i_{th} iteration, y_i represents the true class of the i_{th} set of inputs, and $\mathbf{w}_i^T \cdot \mathbf{x}_i$ represents the model's forecast. It is evident that when the prediction aligns with the true label, the term will have a negative value, resulting in a loss function of zero. If the forecast does not align with the actual value, $-y_i\left(\mathbf{w}_i^T \cdot \mathbf{x}_i\right)$ will result in a scalar greater than zero, thus being chosen as the value of the loss function. Once we have grasped the computation of the loss function, we need to focus on minimizing it, and this is when stochastic gradient descent becomes relevant.

During the 1950s, Robbins and Sutton Monro [19] and Kiefer and Wolfowitz [20] established the foundations for stochastic gradient descent. In the case of a single perceptron, we utilize stochastic gradient descent by calculating the partial derivative of the i-th loss function with respect to each weight in the weights vector. Subsequently, we modify each weight according to a designated learning rate. More formally, we will have:

$$\frac{\partial L_i}{\partial \mathbf{w}_i} = \left[\frac{\partial L_i}{\partial w_{i0}}, \frac{\partial L_i}{\partial w_{i1}}, \dots \frac{\partial L_i}{\partial w_{id}}\right] = \begin{cases} -y_i \mathbf{x}_i & \text{if sign } \left\{\mathbf{w}_i^T \cdot \mathbf{x}_i\right\} \neq y_i \\ 0 & \text{otherwise} \end{cases} \tag{1.4}$$

where w_{i0}, w_{i1}, and w_{id} are the first, second and last weight of $\mathbf{w}$ in the i-th training iteration. As a result, the vector of weights will be updated as follows:

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \eta \nabla_{\mathbf{w}_i} L_i = \mathbf{w}_i + \eta y_i \mathbf{x}_i \tag{1.5}$$

where $\mathbf{w}_i$ represents the vector of weights in the i-th iteration, η denotes the learning rate of the algorithm, and $\nabla_{\mathbf{w}_i} L_i$ signifies the gradient of the loss relative to the vector of weights.

As one can see, we are subtracting and not adding the gradient from the previous version of the weights. This occurs because when calculating the gradient, we are determining the direction of the most rapid ascent of the error function. Since our objective is to decrease it, we need to move in the opposite one.

Modifying the weights can potentially result in misclassifying patterns that were previously classified properly in later iterations, therefore not guaranteeing a decrease in the

loss function with each iteration. Nevertheless, Rosenblatt's result [21] ensures that the perceptron learning algorithm will discover a precise solution within a limited number of iterations if the training data set can be separated linearly.
This particular architecture is not appropriate for our specific goal. However, it did provide a mild initiation into the realm of neural networks. In the upcoming section, we will discuss a more potent architecture, which will be part of the final trading algorithm.
To prevent the notation from becoming cumbersome, we will refrain from using bold type for vectors or matrices henceforth.

1.1.2 Multi-Layer Perceptrons

The perceptron has demonstrated its efficacy in solving issues where data can be separated linearly. Nevertheless, what occurs when this is not the case? Take, for example, the XOR function, a well-known case where a perceptron is unable to accurately categorize the data. For such situations, it is necessary to have a more advanced structure that can acquire knowledge about complex patterns: the Multi-Layer Perceptron (MLP).
An MLP is a specific type of neural network that consists of an input layer, one or more hidden layers, and an output layer. As the Perceptron, the MLP is a feed-forward network, meaning that information flows in one direction from the input layer to the output layer.
In simpler words, the Multi-Layer Perceptron can be seen as a combination of perceptrons, with a small caveat: the Backpropagation algorithm which will be explained later.
The architecture of the MLP is represented beneath.

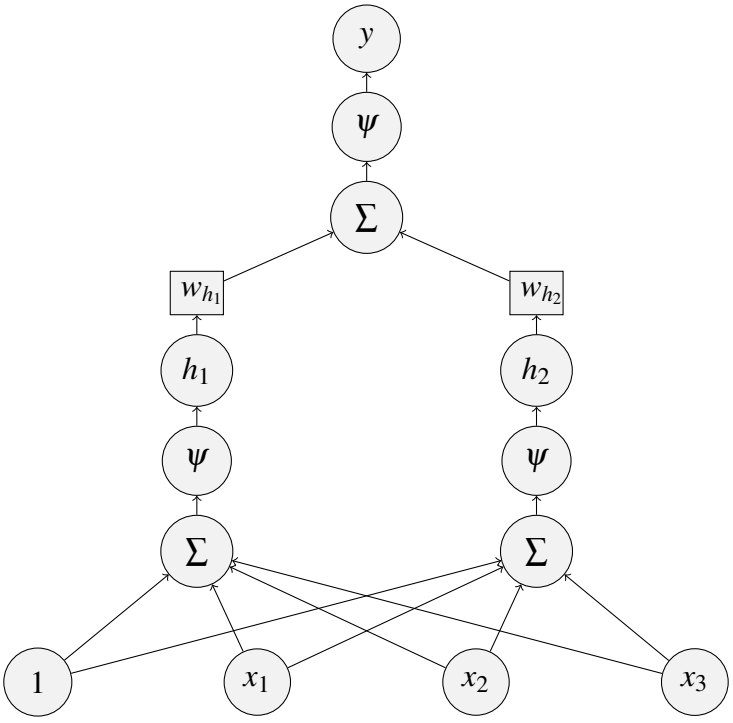

Figure 1.2: a Multi-Layer Perceptron.

In the architecture shown above, the input structure remains the same. To maintain clarity, we suppress the weights. Each connection with the next layer now assigns a specific weight to the inputs, resulting in two distinct weighted sums activated by the function ψ, which is a non-linear activation function. We will then extract h_1 and h_2, the two units of the hidden layer. From this point, it is evident that the workflow follows the same process as the previously introduced perceptron.

The non-linear activation functions are what enable the MLP to learn very complex patterns; more precisely, the MLP can be considered a universal approximator.

In fact, Hornik, Stinchcombe, and White proved in 1989 that MLPs with non-linear activation functions are capable of approximating any measurable function to any desired degree of accuracy in a very specific and satisfying sense, implying that any lack of success in applications must arise from inadequate learning, insufficient numbers of hidden units, or the lack of a deterministic relationship between input and target [22].

Recognizing the critical significance of activation functions, we will now examine a few of them. The sigmoid activation function is a widely utilized and one of the earliest activation functions in neural networks.It maps input values to an output range between 0 and 1 using the formula:

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{1.6}$$

the sigmoid activation function is particularly advantageous in situations where the output can be recognized as a probability. Yet, the sigmoid function has several limitations. In particular, when the input is very large or small, the system's gradients diminish significantly, resulting in the vanishing gradient problem. Because of this, deeper networks, where the signal may weaken as it moves through several layers, benefit less from the sigmoid function.

Other activation functions have been suggested to mitigate certain limitations of the sigmoid function. The Rectified Linear Unit (ReLU) [23] is often adopted as the primary option for most neural networks because of its straightforwardness and efficiency.

The ReLU function is defined as:

$$\text{ReLU}(x) = \max(0,x) \tag{1.7}$$

this function outputs zero for any negative input and the input itself for any positive input, making it computationally efficient.

An important benefit of the Rectified Linear Unit (ReLU) is its capacity to address the issue of vanishing gradient by enabling the smooth flow of gradients when the input is positive. Nevertheless, ReLU is not exempt from its own limitations; it can encounter the "dying ReLU" issue, in which neurons might become inactive and only produce zero outputs if the input values repeatedly result in zero outputs throughout training.

The Exponential Linear Unit (ELU) [24] solves the dead neurons issue and is defined as:

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(e^x - 1) & \text{if } x \leq 0 \end{cases} \tag{1.8}$$

where α is the hyperparameter which determines the saturation point of the ELU for negative net inputs (literally, $\lim_{x\to-\infty}\mathrm{ELU}(x) = -\alpha$).
In contrast to the ReLU, the ELU possesses a gradient that is not zero for negative inputs. This characteristic is advantageous as it facilitates the transmission of gradients during the backpropagation process.
The weight update rule of the perceptron is too basic to handle the complex architectures of MLPs. Throughout the shift from perceptrons to MLPs, a major challenge arises: how to effectively update the weights in networks that have multiple layers. Rumelhart, Hinton, and Williams effectively address this problem in their influential paper [2].
The perceptron's training was straightforward because we could easily compare its output with the expected true output. However, when we add one or more hidden layers, the situation becomes more complex, as we do not know the desired states of the hidden units and the conditions under which they should be active to achieve the desired input-output behavior. Rumelhart et al. were trying to find a powerful modification rule for automatically updating the weights of the MLP, and they came up with the backpropagating algorithm. Before diving into the backpropagation algorithm, it is essential to address several clarifications, specifically regarding syntax and lexicon.
The term "unit" refers to a node or neuron within the MLP. A unit j is a linear function of the outputs, y_i, of the neurons in the previous layer that are connected to j and of the weights, $w_j i$, of these connections. Thus, x_j shall be:

$$x_j = \sum_i y_i w_{ji} \tag{1.9}$$

with the possibility of adding a bias as reported in the perceptron. A neuron gives an output y_j, which in sigmoid function of x_j:

$$y_j = \frac{1}{1 + \mathrm{e}^{-x_j}} \tag{1.10}$$

Based on the premises provided, the total error E can be defined as:

$$E = \frac{1}{2}\sum_c \sum_j \left(y_{j,c} - d_{j,c}\right)^2 \tag{1.11}$$

where c is the number of input-output pairs, j indexes the number of output units, y is the actual output for unit j in case c, and d is the target output.

For a specific case, c, we can minimize E by computing its derivative with respect to the output, yielding the following result:

$$\partial E/\partial y_j = \partial/\partial y_j(\frac{1}{2}\left(y_j - d_j\right)^2) = \frac{1}{2}\cdot 2\left(y_j - d_j\right) = y_j - d_j \tag{1.12}$$

The chain rule can then be used to calculate how E changes with respect to the input x_j:

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot dy_j/dx_j \tag{1.13}$$

remembering that the MLP employs sigmoid activation functions for achieving y_j enables us to determine the value of dy_j/dx_j, thereby obtaining through the quotient rule:

$$dy_j/dx_j = \frac{e^{-x_j}}{(1+e^{-x_j})^2} \tag{1.14}$$

once we understand the value of $1 - y_j$, we can rewrite dy_j/dx_j as follows:

$$1 - y_j = \frac{(1+e^{-x_j})-1}{1+e^{-x_j}} = \frac{e^{-x_j}}{1+e^{-x_j}} \tag{1.15}$$

$$dy_j/dx_j = \left(\frac{1}{1+e^{-x_j}}\right)\left(\frac{e^{-x_j}}{1+e^{-x_j}}\right) = y_j(1-y_j) \tag{1.16}$$

with equation 1.16 at our disposal, we can now calculate the value of $\partial E/\partial x_j$:

$$\partial E/\partial x_j = \partial E/\partial y_j \cdot y_j\left(1-y_j\right) \tag{1.17}$$

thus, we now know how changing x_j will affect the error. Nonetheless, we have already shown that x_j is a linear function of the states and weights of the lower layer.
Therefore, one can calculate how changes in the weights will affect the error knowing $\partial x_j/\partial w_{ji}$ and using again the chain rule.

$$\partial x_j/\partial w_{ji} = \partial/\partial w_{ji}\left(\sum_i y_i w_{ji}\right) = y_i \tag{1.18}$$

$$\partial E/\partial w_{ji} = \partial E/\partial x_j \cdot \partial x_j/\partial w_{ji} = \partial E/\partial x_j \cdot y_i \tag{1.19}$$

Where w_{ji} is the weight of the connection from the lower unit i to the current neuron j, and y_i is the output of the previous unit.
Equation 1.18 allows for an elementary verification of $\partial x_j/\partial y_i = w_{ji}$.
Furthermore, by applying the chain rule for the third time, we can evaluate the impact of a change in y_i on E:

$$\partial E/\partial x_j \cdot \partial x_j/\partial y_i = \partial E/\partial x_j \cdot w_{ji} \tag{1.20}$$

The result corresponds to only a single connection between a neuron i in the penultimate layer and a neuron j in the output layer. Therefore, if we consider all the connections between the unit i and the subsequent layer, we will obtain:

$$\partial E/\partial y_i = \sum_j \partial E/\partial x_j \cdot w_{ji} \tag{1.21}$$

Hence, we have demonstrated the method of recursively calculating gradients from the output layer ($\partial E/\partial y_j$) to the input layer ($\partial E/\partial y_i$). The backpropagation algorithm involves iteratively repeating the procedure described above, layer by layer.

Rumelhart et al. suggested two methods for modifying the weights: in one scenario, the weights can be adjusted after each input-output instance. This is precisely the process of stochastic gradient descent, where the weights are promptly adjusted after each individual training example.

Another method involves aggregating the gradients of the error in relation to each weight across a set of training examples, then subsequently adjusting the weights. The initial method spares memory, whereas the second method has the potential to result in a more reliable convergence.

Assuming that $w(t+1) = w(t) + \Delta w(t)$, the authors introduced two different gradient descent rules for updating $\Delta w(t)$:

$$\Delta w(t) = -\varepsilon \partial E/\partial w(t) \tag{1.22}$$

$$\Delta w(t) = -\varepsilon \partial E/\partial w(t) + \alpha \Delta w(t-1) \tag{1.23}$$

where ε serves the same role as the learning rate η, which we previously observed in the perceptron subsection, and α is an exponential decay factor that defines the impact of the preceding weight update.

Equation 1.22 is the basic form of gradient descent, whereas equation 1.23 incorporates an acceleration method that enhances the algorithm's performance.

With an understanding of how the weights are modified, one can now approach even more intricate designs. Although the MLP has remarkable attributes, it is nonetheless constrained by certain limitations, especially in properly handling temporal relationships in sequential data. This requires the implementation of architectures that are specifically created for managing sequences, such as the Simple Recurrent Network (SRN).

1.1.3 Simple Recurrent Networks

What makes a Multi-Layer Perceptron unsuitable for our objective? The main limitation is its inability to factor in the element of time.

A perceptron does not possess any mechanism for retaining prior occurrences. The system processes each input independently, failing to retain any memory of past states. This poses a difficulty for an MLP to accurately capture temporal relationships or patterns in a time series. In order to overcome this constraint, we employ architectures specifically created to process sequential input, such as the Simple Recurrent Network (SRN).

Jordan discussed this matter in the latter part of the 1980s [25], and this research inspired Elman to introduce the SRN four years later [26].

Elman's objective was to design an architecture that could incorporate the influence of time without transforming the time into a spatial or feature dimension in the model's input. The architecture of the SRN is depicted in the figure below.

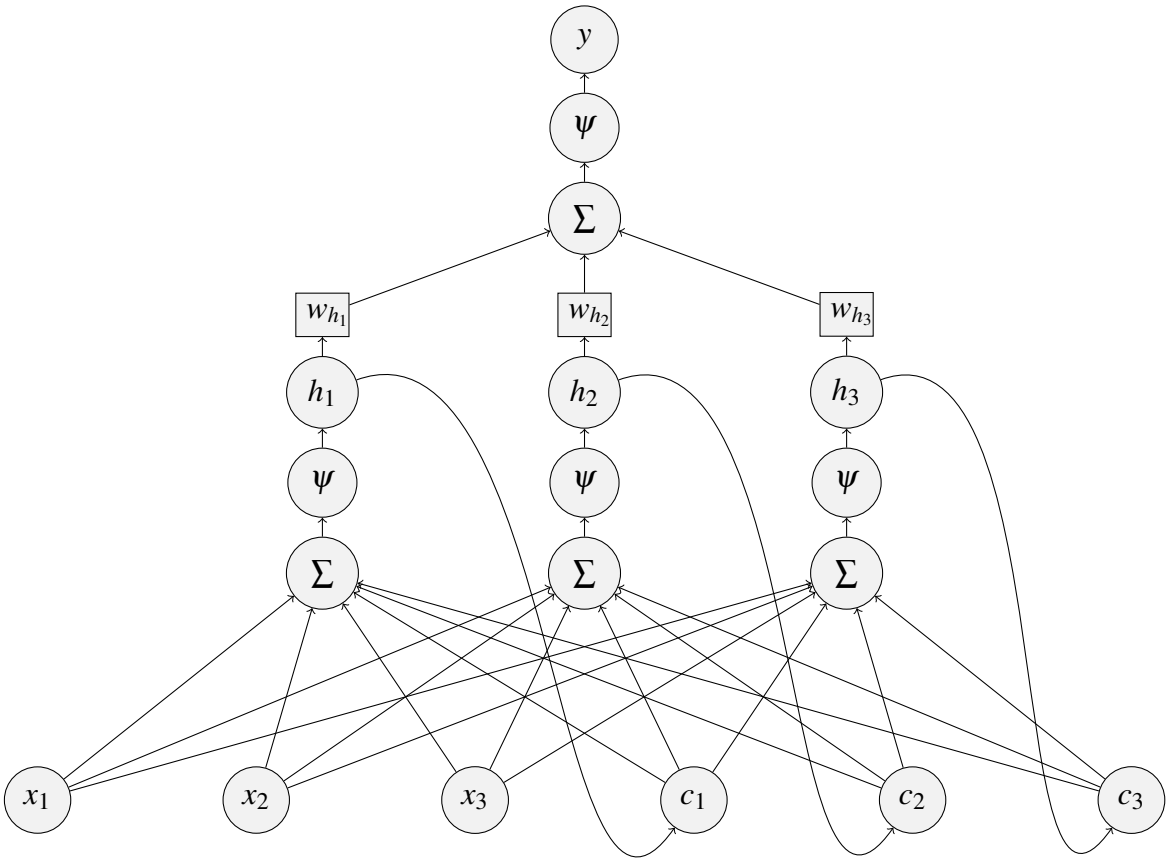

Figure 1.3: a Simple Recurrent Network.

The input nodes in the picture are represented by x_1, x_2, and x_3. On the same level, there are three context units: c_1, c_2, and c_3. To enhance clarity, we have excluded the weights

and the bias. The hidden units in the model are represented by h_1, h_2, and h_3. The term ψ refers to a non-linear activation function, such as the sigmoid or ELU.
The output of the model is denoted by y.
In order to understand how a basic recurrent network operates, the reader should imagine a clock that regulates the delivery of a sequential input to the network. At time $t = 0$, the network will receive the inputs x_1, x_2, and x_3, while initializing the context units to a value of 0.5.The input nodes will be multiplied by their respective weights and the resulting values will be summed for each hidden unit. These hidden units will then be activated using a specified function.
One duty of the hidden units is to feed forward signals to activate the output unit. On the other hand, they also feed back signals to activate the context units. The recurrent connections have a fixed weight of one, which aids in preserving the network's memory of previous information.
At $t = 1$, the process resumes, but now the context units will be assigned the values of the hidden units at $t = 0$, allowing the network to retain memory of previous events.

1.1.4 Long Short-Term Memory Networks

Although SRNs provide a significant advancement over feed-forward networks by enabling the processing of sequential input, they still have drawbacks.
The primary difficulty with SRNs is their ability to acquire and maintain long-term dependencies. When dealing with time series data, it is frequently essential for the network to retain information from earlier in the sequence in order to make precise predictions. However, because of their architecture, SRNs face difficulties in achieving this. In addition, SRNs also encounter difficulties with the vanishing gradient problem, which was previously discussed while explaining the activation functions.
Indeed, during the process of backpropagation, the gradients of the loss function in relation to the weights become extremely small as they pass through multiple layers, especially when working with lengthy sequences. As a consequence, there are only slight modifications made to the weights of the preceding layers, which essentially hinders the network from acquiring knowledge of long-term relationships.
Furthermore, SRNs do not include a mechanism to determine which information should be retained or discarded as time passes. The current prediction is equally influenced by all previous states, irrespective of their importance. This can lead to the preservation of irrelevant data and the omission of potentially significant particulars.
In order to overcome these constraints, Hochreiter and Schmidhuber introduced Long Short-Term Memory (LSTM) networks in 1997 [8], and Schmidhuber again, along with Gers and Cummings, subsequently developed an updated version of the model [27].
The following discussion on LSTM networks will be based on the two papers presented

above. In the previously described instances, we consistently encountered an input layer, one or more hidden layers, and an output layer. When it comes to the LSTM Network, in place of the customary hidden layer, Hohcreiter and Schmidhuber suggested an architecture that included one or more memory blocks. Figure 1.4 depicts the structure of a memory block.

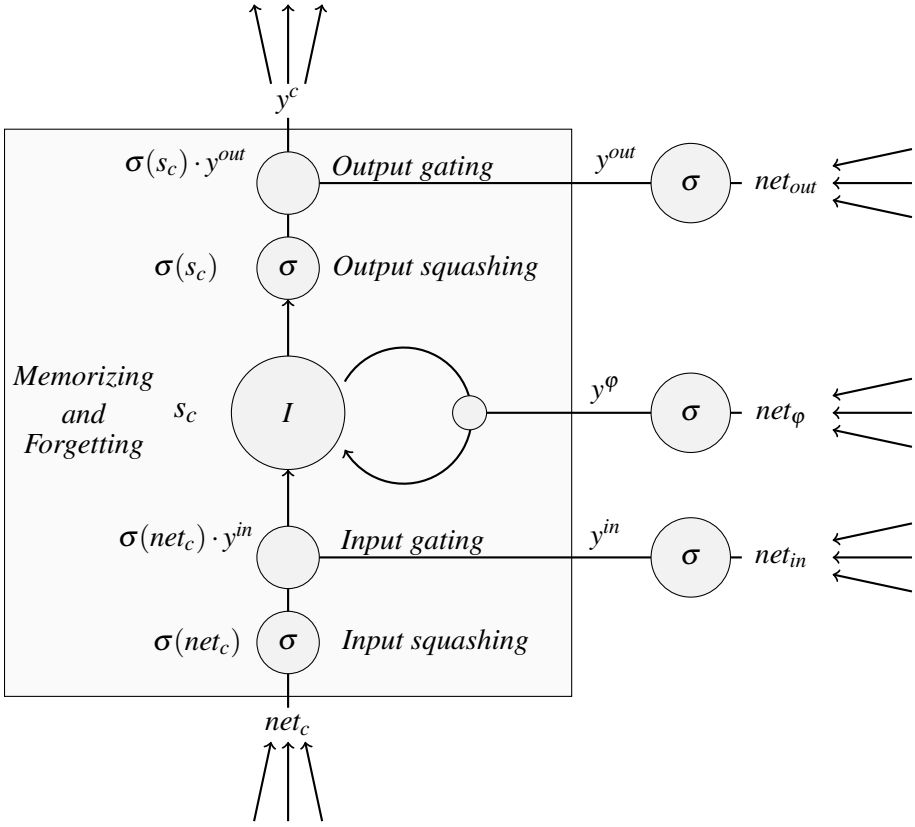

Figure 1.4: a memory block with a single cell.

The image illustrates a more intricate flow of information. In the context of a Multi-Layer Perceptron (MLP), the single unit in the hidden layer was only the weighted product of the units in the preceding layer; now instead, we will speak of "cell state", which will rely on several sources of information. The cell state s_c will depend on its past state and three input sources: net_c, net_{in}, and net_{out}, whose values are detailed in the following equations.

$$\text{net}_{out_j}(t) = \sum_u w_{out_j u} y^u(t-1) \tag{1.24}$$

$$\text{net}_{in_j}(t) = \sum_u w_{in_j u} y^u(t-1) \tag{1.25}$$

$$\text{net}_{c_j}(t) = \sum_u w_{c_j u} y^u(t-1) \tag{1.26}$$

The equations above denote a broader interpretation of Figure 1.4. Indeed, multiple cells may exist, and to eliminate ambiguity, we will designate the j-th cell as c_j. Given that, $\text{net}_{out_j}(t)$ represents the net input to the output gate of the j-th memory cell at time t. The

summation indicing u represents input units, gate units, memory cells (even traditional hidden units, if any). $w_{out_j u}$ denotes the weight linked to the connection between unit u and the output gate of c_j, while $y^u(t-1)$ signifies the activation of unit u at the preceding time step. Similarly, $\text{net}_{in_j}(t)$ represents the net input to the input gate of c_j at t, with a different set of weights and not necessarily the same connections to the other units u.
Conversely, $\text{net}_{c_j}(t)$ does not influence the gate values but directly signifies the net input to the cell state, with $w_{c_j u}$ and $y^u(t-1)$ maintaining the behavior of their peers previously elucidated.
As stated before, the principal weaknesses of the preceding systems are their difficulties in gradient management during backpropagation and their inadequacy in recognizing intricate, long-term patterns. Hochreiter and Schmidhuber addressed these challenges by introducing the "Constant Error Carousel" (CEC), which features only S_c and its fixed self-connection without considering y^φ (that will be explained in a few lines), facilitating a constant error flow and aiming to prevent the vanishing or uncontrolled escalation of error signals.

$$s_{c_j}(t) = s_{c_j}(t-1) + y^{in_j}(t)\,\sigma(\text{ net }_{c_j}(t)) \tag{1.27}$$

According to equation 1.27, the cell's state at time t is equivalent to the preceding state (which corresponds to the fixed self-connection in the CEC) plus $y^{in_j}(t)$ multiplied (gated) by $\sigma(\text{ net }_{c_j}(t))$. In this context, the latter term represents net_{c_j} squashed by the logistic sigmoid, whereas the former is equal to net_{in_j} similarly activated using a sigmoid function.
Hochreiter and Schmidhuber experimented this version of the LSTM (with the simple fixed self-connection on s_c) on different tasks. For example, one of them consisted of learning the embedded Reber grammar, a popular recurrent net benchmark. However, in each experiment, they used a set of different sequences to train the network, resetting s_c to 0 at the beginning of each sequence.
Though, Gers et al. discovered that continuous input feeding to the network (as stock prices) can cause the cell state to grow without bound, ultimately leading to the activation function h's saturation. Two main problems arise from this saturation: the derivative of h approaches zero, hindering the proper spread of the error signal; and the cell's output solely relies on the activation of the output gate, thereby complicating the network's output adjustment. This is why Gers et al. created the solution depicted in Figure 1.4, taking into account not only s_c's fixed self-connection but also y^φ.

$$\text{net}_{\varphi_j}(t) = \sum_u w_{\varphi_j u} y^u(t-1) \tag{1.28}$$

$$s_{c_j}(t) = y^{\varphi_j}(t) s_{c_j}(t-1) + y^{in_j}(t)\,\sigma(\text{ net }_{c_j}(t)) \tag{1.29}$$

$\text{net}_{\varphi_j}(t)$ exhibits a structure similar to the prior net inputs, and $s_{c_j}(t)$ is nearly equivalent

to its preceding version. The sole distinction is that $\text{net}_{\varphi_j}(t)$ is now squashed by a logistic sigmoid and subsequently multiplied by the previous state. This method regulates the extent to which the previous state is transmitted to the present one.

Therefore, we are now equipped to understand the entire information flow within a memory block containing a single cell.

Firstly, we consider discrete time steps $t = 1, 2, \ldots$. A single iteration encompasses the update of all units (forward pass) and the calculation of error signals for all weights (backward pass). We will now focus on the forward pass, where net_{c_j}, representing the weighted sum of a combination of input units, other cell units, and hidden units, is squashed by a centered logistic sigmoid function σ with a range of [-2, 2], resulting in $\sigma(\text{ net }_{c_j}$. This term is then gated by y^{in_j}, where y^{in_j} corresponds to net_{in_j} activated by a logistic sigmoid function with a range of [0, 1]. All activations in the gates will employ this logistic sigmoid function. y^{in_j} operates as a filter, regulating the degree to which the cell state integrates new information from $\sigma(\text{ net }_{c_j})$. The information subsequently arrives in the core of the cell, where I denotes the identity function. The initial version of the network fully transmitted s_{c_j}'s previous state to the new one, whereas Gers et al.'s version did not. In fact, the variant of Gers et al. incorporates the forget date, which governs the retrieval of past information. Specifically, with $s_{c_j}(0) = 0$, the cell state at time t will equal the filtered information from net_{c_j} plus a fraction between 0 and 1 of the preceding state, given by the multiplication to y^{φ_j}. A centered logistic function with a range of [-1, 1] will subsequently compress the resulting s_{c_j}. Afterwards, the output gate will determine the extent of information leaving the cell by gating $\sigma(s_{c_j})$ to y^{out_j}. Thus, the inner workings of a memory block containing a single cell are now evident.

On a more general level, the forward and the recurrent connections in a LSTM network are as depicted in the next page. The figures illustrate the model proposed by Gers et al.; focusing on the topology, the model consists of three distinct layers, with seven input units connected to a hidden layer of four memory blocks containing two cells, resulting in a total of eight cells and twelve gates. Cell outputs are linked to all the cell inputs, gates, and output units, with output units having supplementary connections to the input ones. All gates and output units are subjected to bias, with initial bias weights initialized in blocks: -0.5 for the first block, -1.0 for the second, -1.5 for the third, and so on, making the opening of the gates more difficult at the beginning.

As training advances, biases gradually become less negative, facilitating sequential activation of cells. Forget gates are initialized with symmetric positive values, and all remaining weights, including the output bias, are initialized randomly within the interval [-0.2, 0.2].

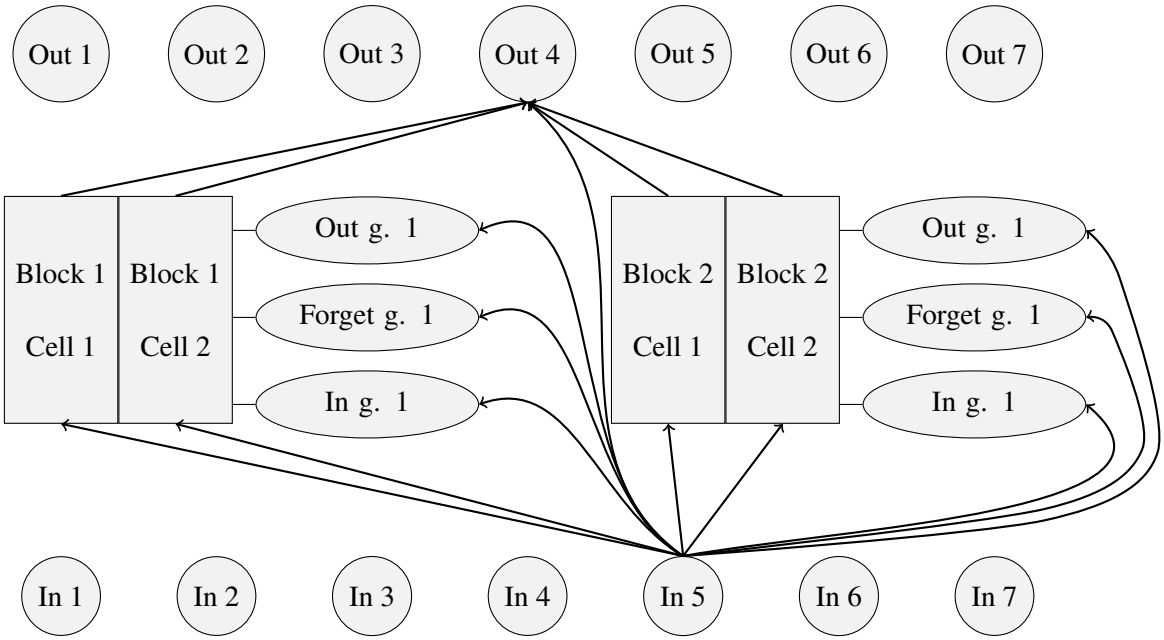

Figure 1.5: forward connections in a LSTM. Blocks and connections are partially shown.

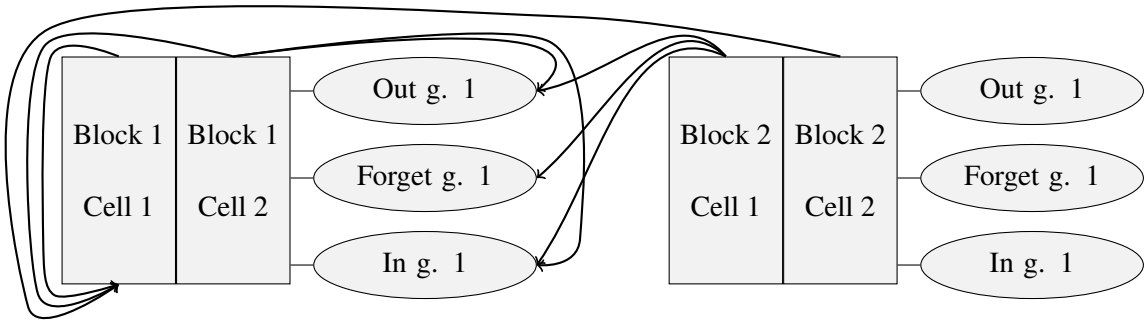

Figure 1.6: recurrent connections in a LSTM. Blocks and connections are partially shown.

Nevertheless, it is still unclear how the weights update in this network, and we are going to address this issue instantly. Gers et al. proposed a solution that combines Backpropagation Through Time (BPTT), a similar algorithm to the one of the MLP, with Real-Time Recurrent Learning (RTRL)[1], a specifically designed algorithm for recurrent networks. In this case, $E(t)$ is the squared error function to minimize.

$$e_k(t) = t^k(t) - y^k(t) \tag{1.30}$$

$$E(t) = \frac{1}{2}\sum_k e_k(t)^2 \tag{1.31}$$

Where $e_k(t)$ is the error for the k-th unit at time t, calculated as the difference between the target value $t^k(t)$ and the actual output $y^k(t)$. To minimize $E(t)$, we employ gradient

[1]Going into detail about these two algorithms is beyond the scope of the thesis; for more information, one can check [28] and [29].

descent, adjusting the weights as follows:

$$\Delta w_{lm}(t) = \alpha \delta_l(t) y^m(t-1) \tag{1.32}$$

The equation denotes the variations in the weights linking the l-th unit to the m-th unit. In addition, α represents the learning rate and serves the same function as η in the previously discussed scenarios, whereas $\delta_l(t)$ represents the error signal for unit l at time t, and $y^m(t-1)$ signifies the activation of unit m at the preceding time step. Equation 1.32 will be used to update the weights of the output unit and output gate. Since α is an arbitrary parameter, and the activation of the m-th unit is already known, the only thing left is the error signal. We compute the error signal of the output in the manner below:

$$\delta_k(t) = f'_{\text{out}_k}(\text{net}_{\text{out}_k}(t))\, e_k(t) = y^{out_k\prime} e_k(t) \tag{1.33}$$

Meaning that the error signal for the output of the k-th unit depends on the derivative of the output gating component times $e_k(t)$. This equation can be interpreted as the measure of how much $e_k(t)$ contributes to the error signal for the output unit k. Moreover, the error signal for the output gate of the υ-th memory block is:

$$\delta_{\text{out}_\upsilon}(t) = y^{out_\upsilon\prime} \left(\sum_j h\left(s_\xi(t)\right) \sum_k w_{k\xi}\, \delta_k(t) \right) \tag{1.34}$$

Where s_ξ stands for $s_{c_\upsilon^j}$. Meaning the cell state in the j-th cell of the υ-th memory block. We will temporarily adopt this solution to avoid a too cumbersome notation.
Let $y^{out_\upsilon\prime}$ denote the derivative of the net input to the output gate in the υ-th memory block, activated by the logistic sigmoid; then, $\sum_j h\left(s_\xi(t)\right)$ represents the summation of all squashed cell states for each j-th cell in the υ-th memory block; finaly, $\sum_k w_{k\xi}\, \delta_k(t)$ signifies the summation of all weights linked to the connections between the k-th output unit and the specific memory cell in the υ-th memory block, multiplied by the signal error calculated in equation 1.33.
For computing the weight updates of the connections leading to the input gates, the forget gates, and directly to the cell, Gers et al. used then a different method. Firstly, the researchers defined the internal state error as follows:

$$e_{s_{c_\upsilon^j}}(t) = y^{out_\upsilon}(t) h'\left(s_\xi(t)\right) \left(\sum_k w_{k\xi}\, \delta_k(t) \right) \tag{1.35}$$

with $h'\left(s_\xi(t)\right)$ representing the derivative of the cell state activation function, multiplied by the weighted sum of the error signals for the output units. Next, we can compute the

partial derivatives of the cell state with respect to the weights in the following manner:

$$\frac{\partial s_\xi(t)}{\partial w_{l_\upsilon m}} = \frac{\partial s_\xi(t-1)}{\partial w_{l_\upsilon m}} y^{\varphi_\upsilon}(t) + \Theta_l(t) y^m(t-1) \tag{1.36}$$

with $l \in \{\varphi, in, c_\upsilon^j\}$. The equation is the product of the partial derivative of the cell state from the preceding time step and the activation of the forget gate, aggregated with $\Theta_l(t)$ multiplied by the activation of a unit m at a previous step (which may be an input unit, a cell unit, or a hidden unit). We must clarify theta; its value is depending upon the partial derivative being calculated.

$$\Theta_{c_\upsilon^j}(t) = g'\left(\text{ net }_\xi(t)\right) y^{in_\upsilon}(t) \tag{1.37}$$

$$\Theta_{in}(t) = g\left(\text{ net }_\xi(t)\right) y^{in_\upsilon\prime}(t) \tag{1.38}$$

$$\Theta_\varphi(t) = s_\xi(t-1) y^{\varphi_\upsilon\prime}(t) \tag{1.39}$$

In the three equations above, $\Theta_\xi(t)$ is equal to the product between the derivative of the net input to the cell, squashed by a sigmoid function, and the activation of the net input. Then $\Theta_{in}(t)$ is the product of the net input to the cell squashed and the derivative of the activation of the net input. In conclusion, $\Theta_\varphi(t)$ corresponds to the past cell state times the derivative of the activation of the net input to the forget gate.
Knowing the last three equations, the weights will be modified as outlined below:

$$\Delta w_{\xi m}(t) = \alpha e_{s_\xi}(t) \frac{\partial s_\xi(t)}{\partial w_{\xi m}} \tag{1.40}$$

$$\Delta w_{lm}(t) = \alpha \sum_j e_{s_\xi}(t) \frac{\partial s_\xi(t)}{\partial w_{lm}} \tag{1.41}$$

At first glance, the formulas might seem puzzling; in fact, one may wonder why the formulas for the input of the cell (the one obtained by net_c to avoid confusion) are different from the ones of the input gate and of the forget gate. In the first case, the internal state error is simply $e_{s_\xi}(t)$, as each cell has its own set of weights.
This means that the weights only impact the state of the cell in question, without affecting any other cells. In the other case, it must be pointed out that the gates are shared among all cells withing the block. As addressed in the discussion of Figures 1.5 and 1.6, for 4 blocks each containing 2 cells, we had 12 gates instead of 24, indicating that each block, not each cell, had 3 gates. This means that the gate weights affect the states of all cells in the block, so the total influence of $w_{lm}(t)$ on the network's error is the sum of its effects on all cell states.
More precisely, equation 1.40 is given by the product between the learning rate α, the internal state error of the j-th cell in the υ-th block, and the partial derivative of the cell

state with respect to the specific weights giving life to net_c. Equation 1.41 instead gives the weights updates, with $l \in \{\varphi, in\}$, by multiplying the learning rate to the summation of all internal state errors of the j cells in the block, and then multiplying this result by the respective partial derivatives.

1.2 Reinforcement Learning

Reinforcement learning is learning what to do—how to map situations to actions—so as to maximize a numerical reward signal [30]. Typically, these techniques involve an agent learning without explicit instructions about which actions to perform. In fact, the agent must ascertain which behaviors yield the greatest returns through trial. The agent's actions will not only determine the present reward; they will also influence the environment in which the agent operates, affecting all subsequent rewards.
This section will delve into the aforementioned concepts, beginning with an overview of Markov Decision Processes (MDPs). Upon studying these processes, the reader will encounter the Bellman equation, regarded as the cornerstone of Dynamic Programming (DP), which encompasses a set of algorithms designed to calculate optimal policies based on a perfect knowledge of the environment, usually represented as a MDP. The next section will focus on the Q-learning algorithm, since it is used in the trading algorithm presented in this thesis. The discussion will also briefly encompass the temporal difference error. Finally, we will examine the trade-off between exploration and exploitation.

1.2.1 Markov Decision Processes

Richard Bellman [31] and Ronald Howard [32] can be considered among the fathers of Markov Decision Processes and Dynamic Programming.
Markov Decision Processes (MDPs) offer a straightforward framework for tackling the challenge of learning via interaction to achieve a certain objective. The agent is the entity tasked with learning and decision-making, whereas all external factors are classified as the environment. The agent and environment participate in ongoing interaction, wherein the agent selects actions and the environment reacts by introducing new scenarios. Furthermore, the environment produces rewards, represented as numerical values, which the agent seeks to optimize over time through the selection of suitable actions. Assuming that the agent operates during a series of discrete time steps, we can set $S_t \in S$, $R_t \in R$, and $A_t \in A(s)$. At every time step, the agent engages with S_t, one of the potential states in S, and then selects an action from the available actions A, considering the current state. Afterwards, at each successive step, the agent receives a reward and enters the new state. Based on the reward and the new state, the agent chooses another action, and the cycle repeats. One might question the distinction between S and R, and $A(s)$.

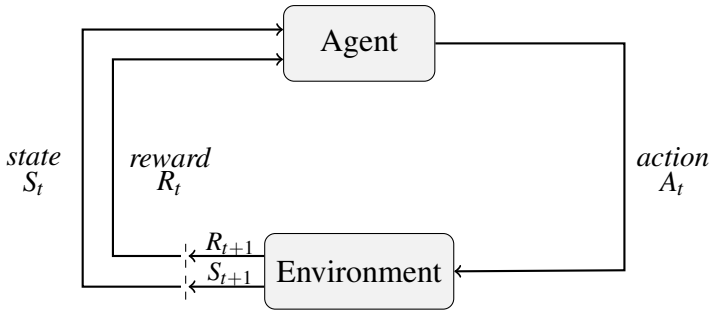

Figure 1.7: a Markov Decision Process.

This occurs because the sets of possible actions depend upon the agent's operational state. This section will address finite MDPs, meaning that the sets S, A, and R are finite. As a result, we can regard R_t and S_t as random variables, with realizations s' and r occurring with probability:

$$p\left(s', r \mid s, a\right) \doteq \Pr\left\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\right\} \tag{1.42}$$

The definition above denotes the probability of moving to state $s\prime$ and obtaining reward r subsequent to executing action a in state s. This definition is valid solely if the Markov property is satisfied, indicating that the agent's state must contain all information regarding every aspect of prior agent-environment interactions. In other words, if the outcomes of R_t and S_t are dependent only upon the prior state and action (and not the precedent ones), the Markov property is satisfied. If that is the case, then equation 1.43 is satisfied.

$$\sum_{s' \in S} \sum_{r \in R} p\left(s', r \mid s, a\right) = 1 \tag{1.43}$$

This implies that the total probability of all potential future states and rewards must encompass all scenarios, resulting in a probability of 1.
An additional important definition in Markov decision processes is the state-transition probability:

$$p\left(s' \mid s, a\right) \doteq \Pr\left\{S_t = s' \mid S_{t-1} = s, A_{t-1} = a\right\} = \sum_{r \in R} p\left(s', r \mid s, a\right) \tag{1.44}$$

The aforementioned formulation specifies the probability of getting the realization s', given the state s and action a. Furthermore, the computation of the expected reward, which the agent will then endeavor to optimize, is of major significance. Sutton and Barto [30] defined it in two distinct manners, contingent upon the quantity of arguments in the

function:

$$r(s,a) \doteq \mathbb{E}\left[R_t \mid S_{t-1}=s, A_{t-1}=a\right] = \sum_{r\in R} r \sum_{s'\in S} p\left(s', r \mid s, a\right) \tag{1.45}$$

$$r\left(s,a,s'\right) \doteq \mathbb{E}\left[R_t \mid S_{t-1}=s, A_{t-1}=a, S_t=s'\right] = \sum_{r\in R} r \frac{p\left(s', r \mid s, a\right)}{p\left(s' \mid s, a\right)} \tag{1.46}$$

In the first scenario, the expected reward with respect to the past state and action is the summation of the rewards times the summation of the respective conditional proabilities. In contrast, in the second scenario, the expected reward is determined by the rewards multiplied by the ratio of the probability of s' and r occurring, given s and a, to the state-transition probability.

In a MDP, the agent's objective of maximizing reward refers not to the immediate benefit, but to the long-term cumulative reward. An example suitable for chess enthusiasts is the following: when training a chess engine such as Stockfish or Leela Chess Zero, the objective is centered on securing victory rather than pursuing subgoals like capturing the opponent's pieces at the expense of losing the game. Therefore, our objective is to optimize the expected return, which can be articulated as follows:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{1.47}$$

In the aforementioned sequence, T represents the final time step. Equation 1.47 is applicable to cases where the interaction between the agent and the environment is segmented into subsequences, referred to as Trials or Episodes. Each episode, which is comparable to a complete chess match, commences with an initial state, followed by various interactions between the agent and the environment, culminating in a terminal state (the final move of the chess game).

In certain instances, such as stock prices, the interaction between the agent and the environment does not naturally segment into episodes; in these ongoing activities, we want to maximize the expected discounted return.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{1.48}$$

Where γ represents the discount rate. As γ increases, the agent progressively assigns greater significance to future rewards, transitioning from valuing solely the current benefit ($\gamma = 0$) to equally considering all rewards ($\gamma = 1$).

From the preceding equation, it can be observed that:

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots && \text{(by (1.48))} \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1} && (1.49) \end{aligned}$$

Consequently, rewards at successive time intervals are related to each other.There is no question that the reward an agent can expect to receive is contingent upon its behavior. The agent must acquire appropriate behavior by learning a policy. A policy is a mapping from state to probability of selecting each possible action.

At time t, $\pi(a \mid s)$ denotes the probability that the agent will execute action $A_t = a$ when in state $S_t = s$; hence, the policy indicates the likelihood of the agent undertaking a specific action in a particular state.

Upon comprehending the concept of policy, we can compute the state value function and the action value function for policy π.

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \tag{1.50}$$

The state-value function is shown above. This function represents the expected return when starting in state s and following policy π afterwards. On the other hand, the action-value function also takes into account the agent's actions.

$$q_\pi(s,a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{1.51}$$

The calculation of the expected return in this case presumes that the agent is in state s, performs action a, and then adheres to policy π. Consequently, by employing value functions, we can now assess the long-term returns of states or state-action pairs, depending upon a certain policy followed by the agent. This indicates that we can assess the potential advantages for the agent of being in a specific state or undertaking a particular action inside that state while following a designated policy.

Similar to the previously observed rewards, it can be demonstrated that the aforementioned value functions obey to recursive relationships. Concentrating on the state value function, we have previously established that:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s \right] \qquad \text{(by (1.50))}$$

The expected reward for being in state s is equivalent to the expected reward at $t+1$ plus the discounted future return at $t+1$. By expanding the expectation, we derive:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} \sum_r p\left(s', r \mid s, a\right) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} \mid S_{t+1} = s'\right]\right] \tag{1.52}$$

Now, we can ascertain that G_t, beginning from state s', is equivalent to $v_\pi(s')$. The aforementioned equation can be reformulated as:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s', r} p\left(s', r \mid s, a\right) \left[r + \gamma v_\pi\left(s'\right)\right] \tag{1.53}$$

This constitutes the Bellman equation [31]. The Bellman equation asserts that the value of the current state, representing the expected total reward an agent can accumulate from state s, is equivalent to the summation over all possible actions a that the agent can take in state s, weighting the transition probability of going to state s' and receiving reward r from state s taking action a, over all possible next states and rewards r that the agent might encounter (after taking action a in state s). All multiplied by the summation of the immediate reward and the discounted value of the next state, which represent the expected cumulative return starting from state s'. Our objective extends beyond merely identifying a policy for the agent; indeed, we seek to construct an optimal policy.
Our goal is to ascertain a policy π_* in which $v_*(s)$ exceeds each $v_(s)\ \forall\ s \in S$. In a finite Markov Decision Process (MDP), there exists a minimum of one optimal policy, which we will represent as π_*, with the optimal state value function designated as:

$$v_*(s) \doteq \max_\pi v_\pi(s) \tag{1.54}$$

And the optimal action value function indicated as:

$$q_*(s,a) \doteq \max_\pi q_\pi(s,a), \tag{1.55}$$

In turn, it is evident that the value of a state under an optimal policy must equal the expected return for the best action from that state. Consequently, we conclude that:

$$v_*(s) = \max_{a \in A(s)} q_{\pi_*}(s,a) \tag{1.56}$$

The term $q_{\pi_*}(s,a)$ can be redefined as the expected return of G_t, conditional upon a and s. Thus, we shall obtain:

$$v_*(s) = \max_a \mathbb{E}_{\pi_*} \left[G_t \mid S_t = s, A_t = a\right] \qquad \text{(by (1.51))}$$

Subsequently, by applying the return formula, we can derive an alternative formulation of the right term:

$$v_*(s) = \max_a \mathbb{E}_{\pi_*} \left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a \right] \tag{1.57}$$

Hence, applying the previous rationale, the maximum future return G_{t+1} must equal $v_*(S_{t+1})$. Therefore, The Bellman optimality equation for the optimal state-value function is expressed as:

$$v_*(s) = \max_a \mathbb{E} \left[R_{t+1} + \gamma v_* \left(S_{t+1} \right) \mid S_t = s, A_t = a \right] \tag{1.58}$$

The expected value can be expanded, leading to the next equation:

$$v_*(s) = \max_a \sum_{s',r} p \left(s', r \mid s, a \right) \left[r + \gamma v_* \left(s' \right) \right] \tag{1.59}$$

In the aforementioned equation, the action that optimizes the expected return is chosen, after which the transition probabilities across all potential states s' and rewards are multiplied by the sum of the immediate reward and the discounted future reward.
Conversely, the Bellman optimality equation for $q_*(s,a)$ is:

$$\begin{aligned} q_*(s,a) &= \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} q_* \left(S_{t+1}, a' \right) \mid S_t = s, A_t = a \right] \\ &= \sum_{s',r} p \left(s', r \mid s, a \right) \left[r + \gamma \max_{a'} q_* \left(s', a' \right) \right]. \end{aligned} \tag{1.60}$$

The Bellman optimality equations for $v_*(s)$ and $q_*(s,a)$ can be regarded as specific instances of the Bellman equation, wherein the agent seeks to maximize its overall expected return. In the Bellman optimality equation for the state-value function, the agent concentrates on a state's value and selects the optimal action from it. In the alternative scenario, the agent considers the value of a particular action in a state and then selects the optimal action for future states.
The Bellman optimality equations are essential to reinforcement learning and MDPs; they offer a recursive breakdown of the value functions for optimal policies. Nevertheless, we cannot implement them directly in algorithms. In fact, solving them is generally impractical for many reasons. In numerous scenarios, such as trading, the state space can be vast, rendering the solution computationally unfeasible. Furthermore, the transition probabilities in most instances are not explicitly known. Consequently, our trading algorithm will implement an alternative solution: Q-learning.

1.2.2 Q-Learning

Q-learning is a fundamental technique in reinforcement learning, established by Watkins in 1989 [4], aimed at addressing the challenge of learning optimal policies in Markov Decision Processes. In contrast to conventional dynamic programming methods that necessitate a comprehensive knowledge of the environment, Q-learning permits an agent to acquire knowledge through interaction, progressively enhancing its performance by exploring different sequences of actions and assessing their results.
Q-learning exhibits parallels with the temporal-difference learning techniques proposed by Sutton [33]. In both methodologies, the agent acquires knowledge from its experiences by balancing the trade-off between immediate rewards and long-term gains. By conducting numerous trials and examining all potential actions across all states, Q-learning progressively refines its assessments of each action's worth, eventually achieving convergence on the best policy.
The versatility of Q-learning has resulted in its extensive application in diverse fields, including financial trading, due to the fact that the algorithm does not necessitate complete knowledge of transition probabilities or reward functions.
We will base the discussion of this subsection on the previous work by Watkins [4], Sutton [33], and Watkins and Dayan [34].
We examine an agent facing a discrete, finite environment, selecting one action from a limited set at each time step, as in MDPs. At time t, the agent is in state s and chooses an action a. This pair will yield a reward for the agent in the subsequent state. The environment will alter in accordance with the previously indicated transition probability:

$$p\left[S_{t+1} = s' \mid S_t = s, A_t = a\right] = p_{ss'}[a] \tag{1.61}$$

The agent seeks to acquire a policy that optimizes the total discounted expected reward, denoted as:

$$v_\pi(s) = \mathbb{E}_\pi\left[G_t \mid S_t = s\right] = r(s,\pi(s)) + \gamma \sum_{s'} p\left(s' \mid s, \pi(s)\right) v_\pi\left(s'\right) \tag{1.62}$$

Given that the agent adheres to policy π from state s, $v_\pi(s)$ represents the total expected reward. Then, using the Bellman optimality equation, we know that if the agent follows an optimal policy π_*, the state-value function will consequently be:

$$v_*(s) = \max_a \left\{ r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) v_*(s') \right\} \tag{1.63}$$

For the moment, we were still recapping some important concepts with a slightly different notation in order to adhere to [34]; novelties will emerge from this point. Watkins and

Dayan describe Q-learning as a form of incremental dynamic programming, wherein the agent determines the optimal policy gradually, progressively adjusting its
Q-value estimates (the values of state-action pairs) based on observed state transitions and rewards at each time step. For a policy π, the Q-values are:

$$Q_\pi(s,a) = r(s,a) + \gamma \sum_{s'} p(s' \mid s,a) v_\pi(s') \tag{1.64}$$

This resembles the previously observed action-value function. In the aforementioned equation, $Q_\pi(s,a)$ value represents the expected discounted reward for performing action a in state s and then adhering to policy π. The objective of Q-learning is to estimate the Q-values that will lead to an optimal policy. Thus, we require an appropriate method for their computation, and this is when temporal-difference methods step in. Temporal-difference methods are useful for making predictions, leveraging past experiences within an incompletely known system to forecast its future behavior.
These methods yield two notable outcomes: TD(0), referred to as one-step TD, and the temporal difference error [33].

$$V(s) \leftarrow V(s) + \alpha \left[r + \gamma V\left(s'\right) - V(s) \right] \tag{1.65}$$

The update method 1.65 tells us how the estimate of the state-value function changes as time goes by. In fact, the estimate is equal to itself plus the temporal difference error multiplied by a learning rate. The term inside squared brackets is the temporal difference error, where r is the reward after taking action a at state s, γ is the discount factor, and $V(s')$ is the estimate of the next state value. The goal is to minimize the TD error so that the summation of the immediate reward and the estimate of the next state value are equal to the current state value. Using equation 1.56, we can rewrite the temporal difference error in terms of Q, obtaining:

$$\delta = r + \gamma \max_{a'} Q(s',a') - Q(s,a) \tag{1.66}$$

Thus, using the one-step TD, the Q-value update rule will be:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right] \tag{1.67}$$

By factoring out the terms for α, and utilizing equation 1.56 once again, we derive the subsequent method for regulating the adjustments of Q-values:

$$Q_{t+1}(s,a) = \begin{cases} (1-\alpha_{t+1}) Q_t(s,a) + \alpha_{t+1} \left[r_{t+1} + \gamma V_t(s') \right] & \text{if } S_{t+1} = s' A_{t+1} = a', \\ Q_t(s,a) & \text{otherwise.} \end{cases} \tag{1.68}$$

This implies that the new Q-value will be equivalent to the previous one, weighted by $(1-\alpha_{t+1})$, plus the total of the immediate reward and the discounted expected reward, all of which are weighted by the learning rate.
This update will only occur when the specific s' and a' are reached. A practical example could aid in understanding this. Assume an environment with 100 possible states, populated by an agent that can perform 4 possible actions. Then the possible pairs are 400. Reaching a specific pair, the agent will update only the Q-value of that specific pair, leaving the Q-values of all the other 399 the same.
Watkins and Dayan proved that, as $t \to \infty$ in a finite markov decision process, if the agent experiences enough times all the state-action pairs, its behavior will converge to the optimal policy, assuming that the rewards are bounded and that $0 < \gamma < 1$.
The more careful readers may have observed that now we denote the learning rate as α_{t+1}. The reason for this is that we now permit α to vary over time. Numerous reinforcement learning techniques diminish the learning rate progressively to stabilize the learning process. An elevated learning rate in Q-learning renders the agent more responsive to new information, this promotes exploration as the agent is more inclined to swiftly adjust its Q-value estimates in response to new actions and outcomes, potentially uncovering superior strategies. Nonetheless, although this may encourage the agent to explore farther, it can also render learning unstable if the agent repeatedly alters its policy in response to short-term fluctuations in rewards. A balance is essential to guarantee sustained convergence.
The issue above sheds light on the dilemma between exploration and exploitation. In order to reap significant rewards, a reinforcement learning agent must prioritize actions that it has previously experimented proven to be effective in generating rewards. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain a reward, but it also has to explore in order to make better action selections in the future. The challenge lies in the fact that solely pursuing exploration or exploitation could lead to task failure.
We can utilize the game of chess again as an example to better understand the concepts discussed above. In certain situations, the algorithm may adopt a suboptimal policy that, instead of leading to a checkmate in two moves, leads to "just" a winning position in more moves. For a more theoretical example, consider a scenario where an algorithm performing gradient descent remains stuck in a local minimum.
There are different strategies that help the algorithm find the right balance. Kaelbling et al. [35] explain various strategies such as the ε-greedy exploration, the Boltzmann exploration, and the Kaelbling's interval estimation algorithm. Another technique is Thompson sampling, proposed by Thompson [36]. In this thesis, we will adopt ε-greedy exploration, meaning that the algorithm will maximize the Q-value with a probability of $1-\varepsilon$, and then explore the environment by performing another action with a probability of ε.

Algorithm 1 Q-Learning with ε-greedy exploration

Initialize $Q(s,a)$ arbitrarily for all states s and actions a
for each episode **do**
    Initialize state s
    while not terminal state **do**
        With probability ε, select a random action a
        Otherwise, select $a = \arg\max_a Q(s,a)$
        Take action a, observe reward $R_{t+1} = r$ and next state s'
        Update Q-value:

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

        Update state $s \leftarrow s'$
    end while
    Optionally decay ε
end for

Chapter 2

Methodology

In this chapter, the discussion will delve into the core of the thesis. We will guide the reader through the foundational architecture of the trading algorithm, the Deep Q-Network, a recent approach that combines deep neural networks with Q-learning.
Next, the specific architecture used for implementing the trading algorithm will be explained. All the distinctions between the architectures proposed in the previous sections, beginning with the various activation functions within the LSTM, are pointed out.
We will then discuss the application of layer normalization to the LSTM hidden state in order to facilitate the learning of the algorithm.
After that, the Kaiming initialization is tackled before delving deeper into the Adam optimizer, which involves a finer method of updating the model parameters.

2.1 Deep Q-Networks

The theoretical chapter addressed a variety of architectures, from the simple perceptron to the LSTM. Now, those architectures will be useful for understanding how Deep Q-networks (DQN) work.
DQN have been introduced by Mnih et al. in 2015 [3]. This new technique, which combines Q-learning with deep neural networks to estimate the Q-value function, marked a significant advancement in the deep reinforcement learning field. In fact, traditional Q-learning approaches, although effective in smaller, discrete state spaces, face constraints in high-dimensional environments. DQN address the issue by approximating the Q-value function through the utilization of deep neural networks. This enables the agent to excel in intricate, high-dimensional settings like video games and financial markets.
In addition, the authors pointed out other drawbacks regarding Q-learning.
Indeed, the sequences provided to the algorithm may be correlated, the same problem may arise between the Q-values and the target values; moreover, minor alterations in the Q-values may subsequently induce fluctuations in the policy, changing the set of expe-

riences (s, a, r, s') from which the agent learns.
The authors tackled these issues by implementing experience replay and by establishing two instances of the identical deep neural network for the computation of Q-values. One instance, referred to as the online network, modifies the action-values at each iteration depending on another instance, the target network, which changes its values only every c iterations. Regarding experience replay, this innovation entails the storage of experiences $e_t = (s_t, a_t, r_t, s_{t+1})$ at each time step t within a set $D_t = e1, \ldots, e2$. Subsequently, a mini-batch of events will be randomly sampled uniformly from the replay memory. As a result, the agent will abstain from gaining knowledge from subsequent experiences, preventing correlations among them.
Mnih et al. used a convolutional neural network [37] for the task because to its capacity to identify patterns in images and shapes; conversely, this thesis will employ a multilayer perceptron.
The problem is formulated as a Markov Decision Process (MDP) involving an agent engaging in Atari games; the process is repeated for different atari games. The agent receives a sequence of previous actions and states, represented as images from the Atari emulator, and selects an action a_t, receiving a reward reflected by the change in the game score. The agent selects its behavior to optimize future rewards. The authors define the future discounted return as:

$$R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \tag{2.1}$$

Where γ as usually represents the discount rate and is set to 0.99. The maximum expected return achievable is again shaped by the Bellman equation:

$$Q^*(s,a) = \mathbb{E}_{s'}\left[r + \gamma \max_{a'} Q^*\left(s', a'\right) \mid s, a\right] \tag{2.2}$$

The novelty appears here; specifically, the authors employed a neural network, namely a Q-network, to estimate the action-value function, $Q(s,a;\theta) \approx Q^*(s,a)$. This network can learn by modifying its weights, θ_i, at each iteration to minimize the mean squared error in the Bellman equation. Nevertheless, given the inability to compute the ideal target values, we replace them with approximated target values derived from the parameters θ_i^- of the target instance of the Q-network. Consequently, the loss function will assume the subsequent form:

$$L_i(\theta_i) = \mathbb{E}_{s,a,r}\left[\left(\mathbb{E}_{s'}[y \mid s,a] - Q(s,a;\theta_i)\right)^2\right] \tag{2.3}$$

$$= \mathbb{E}_{s,a,r,s'}\left[\left(y - Q(s,a;\theta_i)\right)^2\right] + \mathbb{E}_{s,a,r}\left[\mathbb{V}_{s'}[y]\right] \tag{2.4}$$

with $y = r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right)$. The value of y marks a significant distinction from supervised learning; indeed, the targets depend on the model itself, specifically on the

weights θ_i^- of the target network. In contrast, supervised learning typically involves static targets that remain unchanged during the training process.
The initial formulation of the equation represents the mean squared error between the target Q-values and the predicted Q-values. The expectation is computed based on the sampled experiences inside the minibatch. In the alternative version of the loss, the second component denotes the variance of the target Q-values. The optimization process excludes this component because it is independent with respect to the online network parameters. Of course the evolution of the target network must be controlled; so, the target network weights are fixed for a specified number of iterations to promote learning stability. As seen in the preceding chapter, to derive a method for updating the weights, it is necessary to compute the derivative of the loss function with respect to the weights, resulting in the following gradient:

$$\nabla_{\theta_i} L(\theta_i) = \mathbb{E}_{s,a,r,s'}\left[\left(r + \gamma \max_{a'} Q\left(s', a'; \theta_i^-\right) - Q(s, a; \theta_i)\right) \nabla_{\theta_i} Q(s, a; \theta_i)\right] \tag{2.5}$$

Where $\nabla_{\theta_i} Q(s, a; \theta_i)$ represents the gradient of the predicted Q-values in relation to the network parameters. Figuring out all the expectations for the aforementioned equation would be computationally demanding; hence, the authors chose to employ stochastic gradient descent to update the network weights with solely the information contained within the minibatches. DQN can be considered a model-free algorithm, as it uses minibatches for learning and does not explicitly estimate reward or transition probabilities (1.44). Furthermore, the method can be classified as off-policy, indicating that during training, the agent attempts to learn the greedy policy without adhering to it. In other words, the agent tries to learn the optimal policy without behaving optimally. This occurs because the agent follows the previously explained ε-greedy policy. Mnih et al. proposed a policy in which ε, starting from 1, anneals linearly to 0.1.

2.2 Model Architecture

For the trading algorithm, we have employed a deep Q-network integrated with a Long Short-Term Memory layer, as suggested by Li et al [12]. This architecture is designed to extract temporal dependencies in sequential data, such as stock prices, via LSTM and employ the fully connected layers of the MLP to convert these extracted features into actionable outputs. The section outlines the fundamental components of the architecture, including the LSTM, the two instances of the MLP, activation functions, layer normalization, and weight initialization method.
Instead of populating the replay memory with the the agent collects at each time step at the beginning of the trading until the experience memory reaches maximum capacity, we have adopted the following solution: rather than starting directly the training at time t, we

initiate it at $t-(capacity+k-1)$.
Between $t-(capacity+k-1)$ and t, the agents perform random actions, gathering experiences, while keeping all the parameters θ frozen. Therefore, when it arrives at time t, the agent will have the replay memory at full capacity. In this way, we make sure that when the training starts, the batches will have a big enough set for extracting experiences, thus avoiding sequential data.
The LSTM layer is the primary component of the architecture and has the role of capturing temporal dependencies in sequential financial data. As seen before, LSTM networks are a variant of recurrent neural networks that specialized in sequence processing by retaining information over prolonged periods via its internal memory cells, as shown in Section 1.1.4. In the algorithm, the LSTM is fed with tensors of size $(batch, k, n_{features})$, where the batch size is equal to 64 as proposed by Gyeeun and Kim [38]. Each sequence has a length k, which represents the number of historical prices, including the current one. The $n_{features}$ is set to 1, indicating that we are solely feeding the LSTM with stock prices. For instance, if we were incorporating current stock prices and the past relative strength index values, $n_{features}$ would have been 2. The default activation functions proposed by PyTorch are slightly different than the ones previously seen in Section 1.1.4. The input, output, and forget gates use similar activation functions, but net_c, representing the candidate cell state, and S_c, representing the cell state, use the hyperbolic tangent activation. The computation of this function yields values ranging from -1 to 1, as shown below:

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.6}$$

The LSTM possesses 128 memory blocks, each with one cell. The Q-network, an MLP in our scenario, will receive 64 vectors each long 128 from the LSTM. To understand why, we need to have a more in-depth approach, which will be based on the insights by Goodfellow, Bengio, and Courville. [39]. The gates of an LSTM can be reformulated as follows:

$$gate_j^{(t)} = \sigma\left(b_j^{\iota} + \sum_m x_m^{(t)} U_{j,m}^{\iota} + \sum_n h_n^{(t-1)} W_{j,n}^{\iota}\right) \tag{2.7}$$

In this context, $gate_j^{(t)}$ represents the value of the gate of the j-th cell at time t, m denotes the number of input features (1 in our case), h denotes the dimension of the output layer at each time step, which is 128, all considered with $\iota \in \{in, \varphi, out\}$ meaning that every gate has its own specific set of weights.

$$s_t = y_t^{\varphi} \odot s_{t-1} + y_t^{in} \odot \tanh\left(b + x_t U + y_{t-1} W\right) \tag{2.8}$$

Where s_t is the cell state at time t. y_t^{φ} is the forget gate, which has dimension (64, 128) and is element-wise multiplicated by the previous cell state having the same shape. The

two terms previously mentioned are summed to $y^i n_t$, which is multiplied by the squashed value of Net_c. Net_c is equal to the vector of biases b, which has dimension (1,128) plus the multiplication of the vector of inputs x_t (64,1) with the input weights matrix U that has a dimension (number of features, 128) so in this case collapses to a vector, and the product between the past hidden states y_{t-1} (64,128) with the recurrent weights matrix W (128,128). By performing the computation, it is possible to verify that the output of the LSTM at each time step will have the shape of (64, 128). This occurs due to the broadcasting of the bias vector [40], meaning that each element of the j-th column in the matrix $x_t U + y_{t-1} W$ is summed to the j-th element of the vector of biases. After that, the cell state is squashed by the tanh activation function and then multiplied element-wisely to y_t^{out}, thus not changing the final shape of the output. By repeating this process for each price in the sequence, we will ultimately have five feature vectors. The Q-network will only receive the most recent one. What is the rationale for this? The reader should remember that at each stage, the LSTM gathers important information while eliminating redundant data. The final output incorporates information derived from all the preceding hidden states, providing a valuable input to the Qs-network. Prior to entering the Q-network, the output of the LSTM undergoes the layer normalization method developed by Ba et al. [41] [42]. The normalization is applied as shown in the following equation.

$$y_t = \gamma \frac{x_t - \mathrm{E}[x_t]}{\sqrt{\mathrm{Var}[x_t] + \varepsilon}} + \beta \tag{2.9}$$

Where y, one of the 64 vectors of normalized features in the batch at time t, is given by the i-th hidden state vector minus the expected value of its components, all divided by the variance of the elements of $x_{t\,i}$ summed to ε that grants numerical stability (default: 1e-5). The vectors γ and β act as a scaling and shifting components and they are again broadcasted among the batch dimension; the LSTM learns how to properly adjust them as the training goes by. The normalization occurs for each vector in the batch. Implementing layer normalization ensures that the inputs to the Q-network maintain a stable mean and variance, hence mitigating excessive fluctuations in the hidden states thereby improving the learning process.

The information then flows, with full connections, to the first layer of the Q-network, which has 256 neurons. Next, we proceed to a correspondingly sized fully connected layer, and ultimately, the output layer is reached. The output layer contains three fully connected units that will give the estimated Q-values for the actions short, close, and long. Between each layer, we opted for ELU activation functions. The authors of the paper introducing ELU explicitly cited He et al. [43] for the initialization of weights, and we will adopt this same approach. For faster convergence and better performance the weights are initialized using Kaiming, also known as He initialization. The He initialization changes depending on the activation function used. There is no specific version for

ELU activations [44]; Clevert et al. [24] used the version of Kaiming initialization for RELU activations. We will employ the He initialization for LeakyReLu [45] activation, as this activation function bears the closest resemblance to the ELU. LeakyReLu indeed allows small activations for negative input values :

$$\text{LeakyReLU}\,(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{otherwise} \end{cases} \tag{2.10}$$

with α, also referred to as the negative slope parameter, typically equal to 0.01. The kaiming initialization normally samples the weights using the distribution $\mathcal{N}\left(0, \text{std}^2\right)$. The standard deviation is computed as reported below:

$$std = \frac{\sqrt{\frac{2}{1+\alpha^2}}}{\sqrt{128}} \tag{2.11}$$

Therefore, given α = 0.01 and 128 being the number of input units from the LSTM, one can obtain $std \approx 0.125$.
As previously mentioned, deep Q-networks consist of two instances: one for online learning and another for serving as a target. In the preceding lines, we were describing the online network, the one that changes at each iteration. The target network instead is frozen and serves for stabilizing the training and computing the target Q-values.

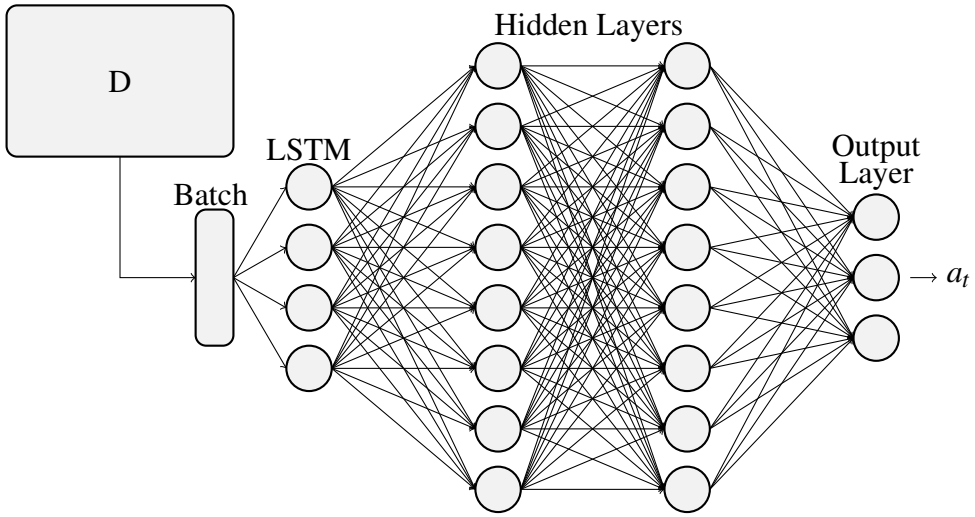

Figure 2.1: a simplified version of a DQN with a LSTM feature extractor. Only the online network is shown.

From a qualitative perspective, the process works as follows: at iteration t, the parameters of the online network are set equal to the parameters of the target network. At iteration $t+1$, the online network undergoes changes due to updates in weights and biases, while the parameters of the target network remain unchanged. The online network will compute the estimated Q-values, whereas the target network will compute the target Q-values.

Then the process will repeat for a specific window size until the parameters of the online network are set equal again to the ones of the target network.
The window size of the target update is an important hyperparameter that can hugely affect the performance of the agent. There will always be a trade-off between a smaller window size, which will enable the agent to respond more quickly to short-term changes in the environment but may result in less stable training, and a larger window size, which offers more stable learning but may cause the agent to answer more slowly to changes in the environment.
Having comprehensively examined the forward flow of information, we will now explore the procedure for updating the architecture's parameters.
In the preceding sections, we analyzed the backpropagation algorithm to elucidate the conventional method of updating weights in a multilayer perceptron (MLP). Instead, the architecture employs an alternative approach, referred to as Adam (or Adam Optimizer) proposed by Diederik P. Kingma and Jimmy Ba [46].
In order to understand how the optimizer works, we need to initialize some important parameters that will come useful during the explanation of the algorithm. The first parameter is the learning rate α, this time called stepsize, which influences the degree of the parameter modifications and is set to 0.001 (the default value). Then there are β_1 and β_2. The former has a default value of 0.9 whereas the latter has a default value of 0.999. The hyperparameters β_1 and β_2 controls respectively the exponential decay rates of the vectors m_0 and υ_0 which will be addressed in few lines, remembering that at $t = 1$ they are vectors of zeros.
After initializing the aforementioned vectors, we start the training process, obtaining the weight updates by calculating the gradient of the loss. We have selected an alternative loss function in our approach, distinct from that of Mnih et al.; in fact, we opted for the Smooth L1 Loss [47]. This loss, closely related to the Huber loss [48], demonstrates quadratic behavior for little errors and linear behavior for significant errors, thus mitigating the influence of outliers on algorithm training. The calculation of the Smooth L1 loss is outlined below:

$$l_n = \begin{cases} \frac{1}{2}(x_n - y_n)^2 / \omega, & \text{if } |x_n - y_n| < \omega \\ |x_n - y_n| - \frac{1}{2}\omega, & \text{otherwise} \end{cases} \tag{2.12}$$

Where the default value of ω in PyTorch is 1, l_n represents the n-th loss in the batch, y_n represents the n-th target Q-value, and x-n represents the n-th estimated Q-value in the batch. So in the end, we will have 64 losses. Then the gradient g_t will be computed on the mean of the losses. Once the gradient with respect to the θ_{t-1} parameters in known

we can perform the following updates:

$$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1)\, g_t \tag{2.13}$$

$$\upsilon_t \leftarrow \beta_2 \upsilon_{t-1} + (1-\beta_2)\, g_t^2 \tag{2.14}$$

Consequently, the reader should now understand the role of m_t and υ_t more clearly. In fact, they are the exponential moving averages of the gradient and the squared gradient, respectively. They will provide estimates for the mean and the uncentered variance of the gradient.

Nonetheless, as previously mentioned, the initialization of m_t and υ_t biases them towards zero. Therefore, we will correct this bias with the upcoming updates.

$$\hat{m}_t \leftarrow m_t / \left(1-\beta_1^t\right) \tag{2.15}$$

$$\hat{\upsilon}_t \leftarrow \upsilon_t / \left(1-\beta_2^t\right) \tag{2.16}$$

These bias corrections allow for more accurate estimates, which are used to update the architecture's parameters as follows:

$$\theta_t \leftarrow \theta_{t-1} - \alpha \hat{m}_t / \left(\sqrt{\hat{v}_t} + \varepsilon\right) \tag{2.17}$$

With ε once more guaranteeing numerical stability. In normal gradient descent, the update is a constant fraction of the gradient; meanwhile, with the Adam optimizer, it may vary based on the mean and variance of the gradient, enabling improved learning. Below is presented the pseudocode of the algorithm as proposed in the paper by Kingma and Ba.

Algorithm 2 Adam: Stochastic Optimization Algorithm

Require: α: stepsize
Require: $\beta_1, \beta_2 \in [0,1]$: exponential decay parameters
Require: $L(\theta)$: loss to minimize
Require: θ_0: initial parameter vector
Initialize: $m_0 \leftarrow 0$ (Vectors and timestep initialization)
Initialize: $v_0 \leftarrow 0$
Initialize: $t \leftarrow 0$
while $t < T$ **do**
$t \leftarrow t+1$
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. $L(\theta)$ at timestep t)
$m_t \leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t$ (Update biased moment estimates)
$v_t \leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2$
$\hat{m}_t \leftarrow m_t / (1-\beta_1^t)$ (Compute bias-corrected moment estimates)
$\hat{v}_t \leftarrow v_t / (1-\beta_2^t)$
$\theta_t \leftarrow \theta_{t-1} - \alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \varepsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

2.3 Experimental Setup

We will test the trading algorithm with five separate time series of daily adjusted closes, considering the data from the last 21 years, ranging from October 15, 2003, to September 20, 2024. The selected five time series encompass several asset classes and have exhibited distinct behaviors over the past years; this selection was made to evaluate the algorithm's generalization capabilities. Our primary selection is the future price of gold (GC=F). We have chosen gold for our analysis, as it is frequently seen as a source of security during economic turmoil, demonstrating more stable behavior in stormy periods. Our second selection is the S&P 500, which has undergone a significant increase within the specified timeframe. We selected it to include an index in our analysis. We have identified three equities that have demonstrated markedly distinct trajectories over the previous ten years. Initially, we selected The Coca-Cola Company (KO), which demonstrated consistent growth over the past two decades. We picked The Walt Disney Company (DIS), which has undergone a significant drop during the few past years. The ultimate selection is Intel Corporation (INTC), which has observed a considerable volatility over the last twenty years, resulting in an almost 50% decline in its stock price. The source of the data

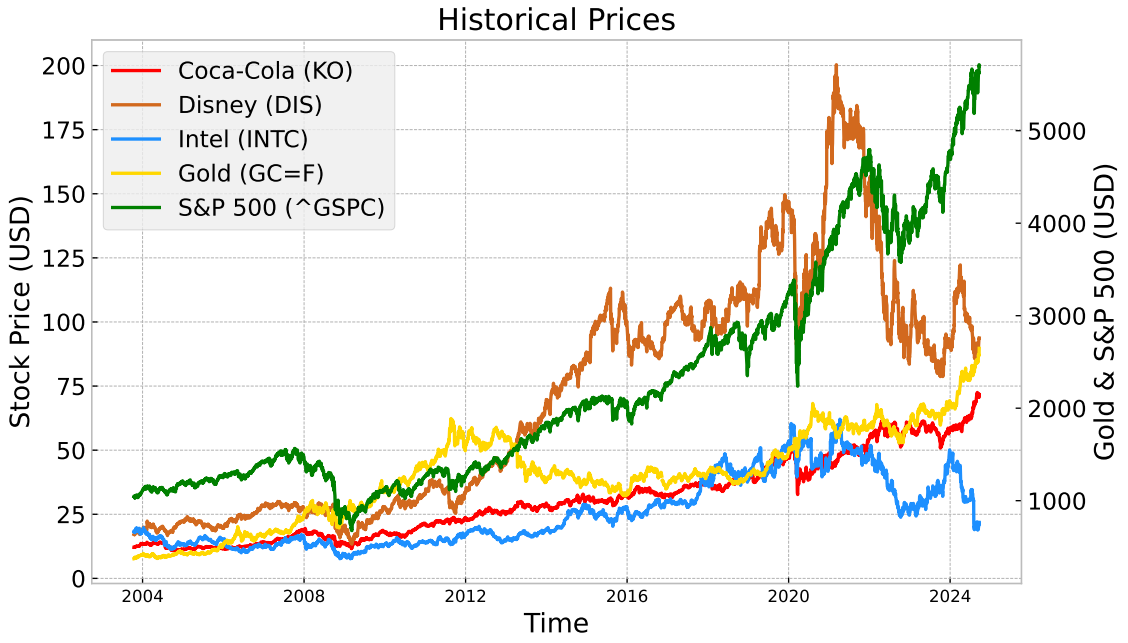

Figure 2.2: stock Prices of Visa, Disney, Intel vs. Gold & S&P 500 from 2004 to 2024

is YahooFinance and the computer programming language used is Python. Every time series will comprise 5269 values. The NaNs have been subsituted by the geometric mean of the two nearest values; for consecutive NaNs, we have applied a geometric progression to ensure a smooth transition between the two closest non-NaN values.

The algorithm has two categories of input data to facilitate its learning. Raw data (the adjusted closes) to precisely compute cumulative returns and learning rewards, and scaled

data as illustrated below:

$$p_t \; scaled = \frac{p_t - \min p_{1:T}}{\max p_{1:T} - \min p_{1:T}} \tag{2.18}$$

This scaling is done for feeding the data to the Deep Q-network. Usually the data set is split in training, validation, and testing; in our case, it is split in 4 subsets. This occurs because of the architecture of the Deep Q-networks. In fact, as mentioned earlier the algorithm utilizes the initial segment of data to populate the replay memory, a reservoir that retains previous experiences for learning. Upon filling the replay memory, the algorithm allocates the remaining data into three segments: the initial 70% for training, the next 20% for validation, and the concluding 10% for algorithm testing. The agent will obtain

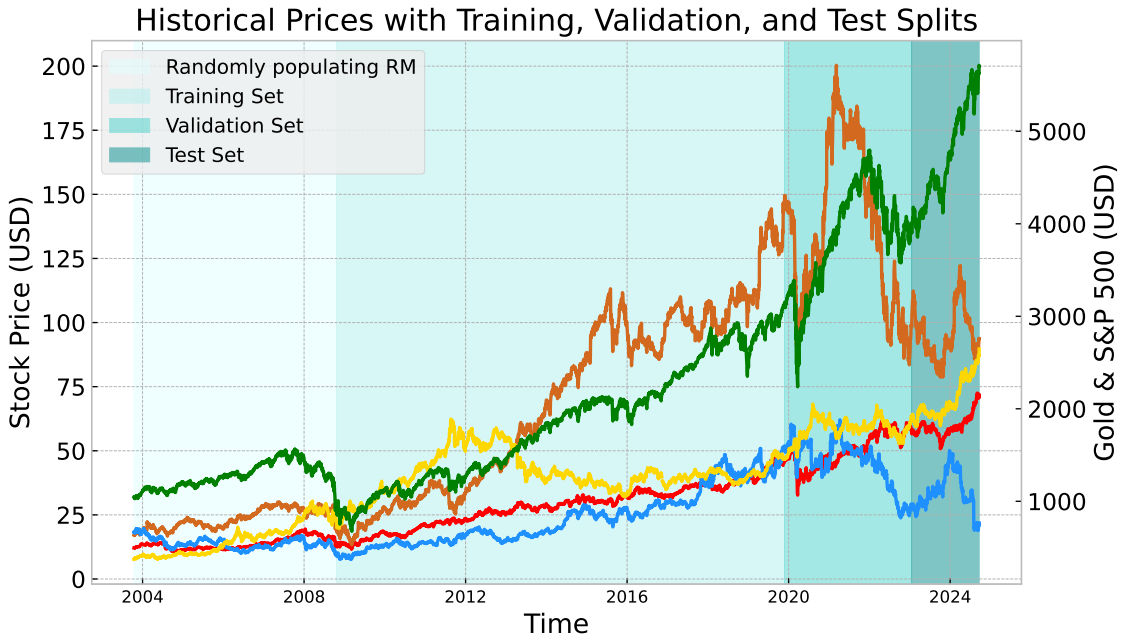

Figure 2.3: replay memory prepopulation, training, validation, and test splits

a state that reflects the most recent k prices. Using this data, it will execute one of the following actions:

$$a_t = \begin{cases} -1 & \text{short,} \\ 0 & \text{close,} \\ 1 & \text{long} \end{cases} \tag{2.19}$$

By -1, we mean opening or maintaining a short position if the preceding action was the same. By 0, we indicate the closure of a position if the agent is in the market or to abstain from opening any positions if the agent is out of the market. By 1, we refer to opening or holding a long position if the preceding action was 1.
Transaction fees have been considered in order to achieve more realistic results. After the

agent's action is executed, we calculate the reward using the formula:

$$r_{t\,simple} = \begin{cases} (1 - n \cdot sgn \cdot tc)\frac{p_{t+1}-p_t}{p_t} & \text{long,} \\ -(1 + n \cdot sgn \cdot tc)\frac{p_{t+1}-p_t}{p_t} & \text{short} \end{cases} \tag{2.20}$$

Where n represent the number of operations, sgn denote the direction of the price change, tc signify the transaction cost, with the remaining term representing the arithmetic return. When the agent maintains a long position, there are no transaction costs; by constrast, closing either a long or short position ($a_t = 0$) is considered a singular operation, so $n = 1$. The value of n is 2 when the agent transitions immediately from a long position to a short position (or viceversa). For example, firstly, the agent closes a long position, and then instantly shortens the stock, thus $n = 2$. To enhance the algorithm's resemblance to reality, the agent incurs transaction costs when maintaining a short position for more than one day.

This thesis will analyze two distinct reward functions: the first, which includes the returns previously reported, and the second, which incorporates the agent's risk aversion, with the same structure of transaction costs. We will denote the two reward functions as the function of simple returns, or returns, and the other as the function of risk-averse returns, or averse returns, interchangeably. The second reward function is inspired to the one proposed by Zeng et al. [11]. We compute those returns, namely the risk-averse returns, using the following formula:

$$r_{t\,risk-averse} = r_{t\,simple} - a_t \cdot \lambda \cdot downrisk \tag{2.21}$$

In this context, λ, representing the agent's risk aversion, is established at 0.05, while $downrisk$, defined as the standard error of negative returns for the designated window of value k, is calculated as follows:

$$downrisk = \frac{\sqrt{\frac{1}{k-1}\sum_{i \in I}(r_i - \bar{r})^2}}{\sqrt{k}}, I = \{i : r_i < 0, t-k+1 \leq i \leq t\} \tag{2.22}$$

The term a_t denotes the present action, the returns r_i and $\bar{r}$ represents respectively the i-th negative return and the mean negative return of the window of size k. In the case of a short position, the product between λ and $downrisk$ is added; in the case of a long position, the specific term is subtracted. In times of turmoil, the downrisk tends to increase, playing the role of a bonus for selecting the sell action; conversely, during periods of growth, the downrisk tends to decrease to zero, thereby reducing the penalty for the long action. Given that, we will have a reward function that is more focused on maximizing profits (the former), and another that aims to maximize reward while also mitigating risk (the latter).

In the Bellman equation, we chose 0.85 as a discount factor value for the future reward, as done by Gyeeun and Kim [38]. We trained the algorithm over a period of 50 episodes. Moreover, we have implemented the ε-greedy policy in the following manner:

$$\varepsilon_{t+1} = \max(\varepsilon_t * \varepsilon_{decay}, 0.05) \tag{2.23}$$

With $\varepsilon_0 = \varepsilon_{start} = 1$ and a decay value of 0.995. In this manner, ε will decay exponentially to 0.05 in around 600 iterations. This process will only occur in the first episode, not the subsequent ones. Indeed, during the initial stages, it's crucial to prioritize exploration. However, as the number of episodes increases, a reset of the ε decaying process could lead to instability in the agent's learning. Consequently, ε will stay at 0.05.
For hyperparameter tuning, we used Optuna [49]. In the table below, one can see the hyperparameters explored.

Hyperparameters	Values
k	{5, 10}
γ	0.85
D	{750, 1000, 1250}
α_{Adam}	0.001
Num Episodes	50
Batch Size	64
Target Update	{20, 40, 60, 120}
ε_{start}	1
ε_{decay}	0.995
ε_{end}	0.05

Table 2.1: hyperparameters explored

Therefore, the size of the replay memory, from which the algorithm will sample the experiences, will be explored as a hyperparameter with values 750, 1000, and 1250. This way, the agent will observe a variety of market conditions. Moreover, we will minimize the likelihood of correlated sequences and conduct a more comprehensive exploration of the state-action space. We will feed sequences of length 5 or 10 into the architecture, corresponding to one or two workweeks. Finally, we have set the target update to either 1, 2, 3, or 6 months. We have explored a total of 24 combinations per each asset-reward function pair, which has resulted in approximately 3.5 days of machine time using virtual machines in the Google Colab environment.
Afterwards, we preserved the optimal combination for each pair and conducted five trials, training the algorithm on both the training and validation sets, rather than solely on the training set as in hyperparameter validation. The results will be presented in the subsequent chapter.

Chapter 3

Results

In the following lines we will assess the algorithm's efficacy across several asset classes and reward functions utilizing many performance metrics. The Sharpe ratio, established by William F. Sharpe in 1966 [50], is one of the most famous metrics. The Sharpe ratio is calculated as:

$$S = \frac{R_p - R_f}{\sigma_p} \tag{3.1}$$

Where R_p is the return of the portfolio, which will be each single stock in our case; R_f is the risk free rate, which we assume equal to zero only for basing our discussion entirely on the risk ajusted component. The last term σ_p which is the standard deviation of the portoflio across the testing window. This commonly utilized financial statistic illustrates the excess returns obtained relatively to the volatility encountered, signifying if the returns justify the associated risk level.
The Sortino ratio, introduced by Frank A. Sortino and Robert van der Meer in 1994 [51], is another significant metric. The ratio is calculated as follows:

$$\text{Sortino Ratio } = \frac{R_p - R_f}{\sigma_D} \tag{3.2}$$

Where σ_D is computed as reported below:

$$\sigma_D = \sqrt{\frac{1}{n}\sum_{i=1}^{n} \min\left(0, R_i - R_f\right)^2} \tag{3.3}$$

The Sortino ratio is a variant of the Sharpe ratio that considers solely the standard deviation of negative returns. Consequently, this measure is more advantageous for risk-averse individuals. Indeed, for equivalent return performances, the ratio favors the more conservative strategies.
Another significant indicator to be evaluated is the maximum drawdown (MDD) intro-

duced by Magdon et al. in 2004 [52]. The MDD is calculated as follows:

$$\text{MDD} = \min\left(\frac{C(t) - \text{Max}(C(\tau))}{\text{Max}(C(\tau))}\right) \tag{3.4}$$

The MDD quantifies the most significant decline from a peak (local maximum) to a trough (local minimum) prior to the portfolio's recovery and subsequent achievement of a new peak. It not only evaluates the global minimum and maximum but also monitors all peak-to-trough fluctuations in the dataset to identify the most significant decline in value.This indicator measures the greatest decline in an investment's value, informing the trader of the potential loss in the portfolio value.

Subsequently, we have selected more pragmatic criteria, including the trading efficiency, the average holding period, and the proportion of successful trades. The percentage of succesful trades is calculated as follows:

$$\%\ \text{Correct trades} = \frac{n_p}{n} \tag{3.5}$$

With n_p equal to the number of profitable trades and n equal to the total number of trades. The trading efficiency is the calculated as shown below:

$$\text{Trade Efficiency} = \frac{\frac{1}{n_p}\sum_{i=1}^{n_p} P_i}{\frac{1}{n_l}\sum_{i=1}^{n_l} |L_i|} \tag{3.6}$$

Where n_l is the number of losing trades. Therefore, the trading efficiency is nothing but the average gain per trade divided by the average loss per trade.The connection between the percentage of successful trades and trading efficiency is often strong. An agent may exhibit a successful trade percentage below 50%, even while maintaining great trading efficiency and overall profitability. Conversely, high trading efficiency does mean profits if the agent makes few accurate judgments, resulting in a poor percentage of succesful trades.

Finally, the last metric is the average holding period, which computes how long the agent keeps open a long or a short position on a regular basis.

$$\text{Avg Holding Period} = \frac{1}{n}\sum_{i=1}^{n}\left(t_{\text{exit }i} - t_{\text{entry }i}\right) \tag{3.7}$$

As previously explained, we have explored 24 combinations of hyperparameters per each asset-reward function pair, obtaining the following best combinations.

Ticker	Reward Function	Trial	Reward	Capacity	Target Update	k
DIS	Averse returns	1	1.95	1000	60	5
DIS	Returns	9	1.98	1000	20	5
GC=F	Averse returns	22	0.83	1000	120	10
GC=F	Returns	20	0.37	750	120	5
INTC	Averse returns	15	0.74	1250	60	5
INTC	Returns	6	2.20	1000	40	10
KO	Averse returns	4	0.25	1250	20	5
KO	Returns	23	0.44	1000	40	10
^GSPC	adverseReturns	14	1.45	1250	20	10
^GSPC	returns	19	1.43	750	120	5

Table 3.1: performance metrics and parameters for different trials

For each asset, the algorithm was able to make profits with specific combinations of hyper-parameters in the validation test. DIS and the returns reward function of INTC obtained the best results reaching gains of 195% and more.The worst performances are in KO and GC=F, with gains "only" in the 25% to 50% range.
Of course some combinations performed poorly, losing more than half of the capital during the validation. We will base our discussion on the ones present in the table. We now present the performance metrics and the cumulative returns for each asset.

3.1 S&P 500 (^GSPC)

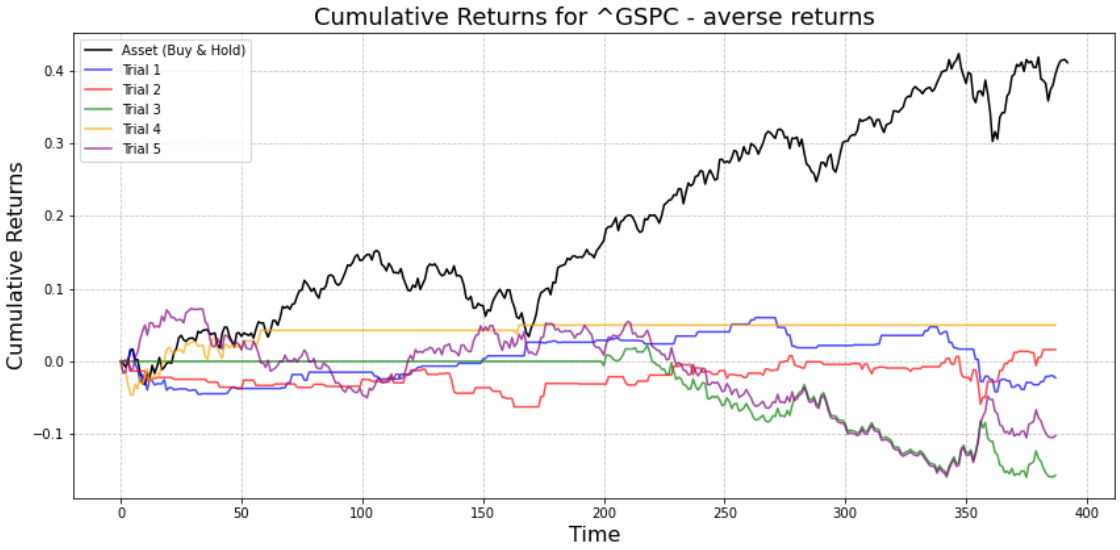

Figure 3.1: ^GSPC algorithm trials vs Buy & Hold. Averse returns.

As shown in figure 3.1 the algorithm is able to gain only in two of the three trials. Moreover, the agent sometimes struggles to open new positions, as seen in trials 3 and 4. The

Buy & Hold strategy outplays the algorithm. One of the explanations may be the reward function, which is most suited for turmoil periods, whereas the S&P observed a significant surge in the last year and a half.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	-0.02	-0.23	-0.18	-0.10	0.57	1.14	2.07
trial 2	0.02	0.14	0.13	-0.07	0.53	2.57	1.21
trial 3	-0.16	-1.26	-1.44	-0.18	0.42	1.03	61.33
trial 4	0.05	0.56	0.35	-0.05	0.53	1.28	19.00
trial 5	-0.10	-0.56	-0.85	-0.27	0.45	1.20	2.89

Table 3.2: performance metrics for ^GSPC Averse Returns trials

In most cases, the algorithm struggles to produce profits, but it manages to maintain a relatively low maximum drawdown. The efficiency is good, meaning that the average gain is higher than the average loss. However, the agent falls short in terms of the percentage of successful trades.

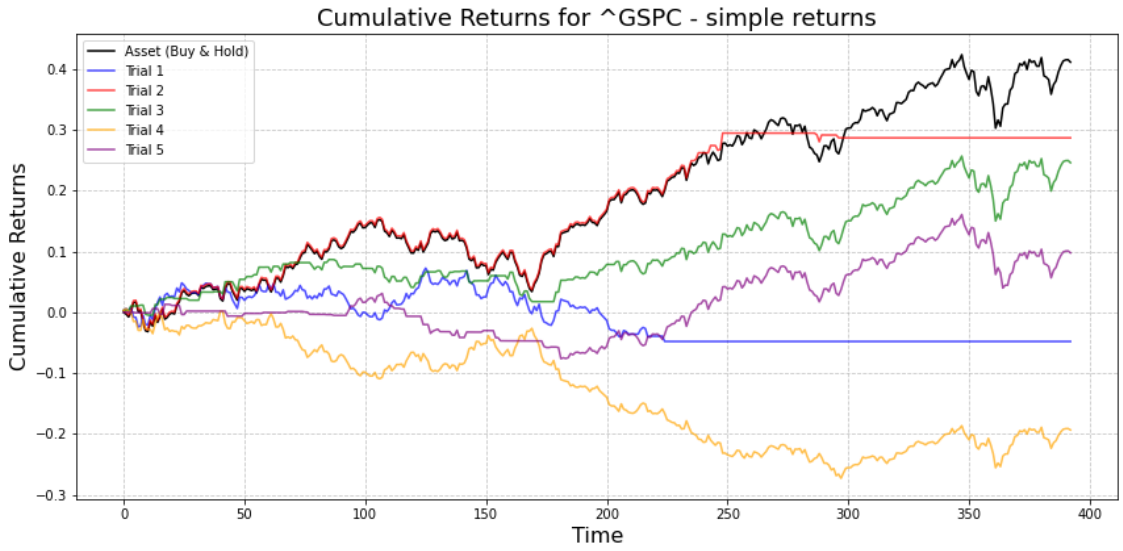

Figure 3.2: ^GSPC algorithm trials vs Buy & Hold. Simple returns.

The results for the simple returns reward function are more promising. The agent can generate profits more consistently, albeit not in a systematic manner. The agent has learned to act better, keeping the position closed less frequently. Even if the inactivity of the agent is smaller this still can be considered a major point for future research. Future solutions to this issue may be found in the structure of rewards when the agent closes the position.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	-0.05	-0.34	-0.42	-0.11	0.46	1.14	2.07
trial 2	0.29	1.64	2.13	-0.10	0.51	1.17	41.00
trial 3	0.25	1.34	1.69	-0.08	0.57	1.49	3.76
trial 4	-0.19	-1.12	-1.60	-0.28	0.48	0.96	4.89
trial 5	0.10	0.59	0.65	-0.10	0.55	1.19	4.94

Table 3.3: performance metrics for ^GSPC Returns trials

As previously observed in the graph, the holding period remains small for most trials and decreases significantly with respect to the most extreme cases observed in the averse returns. The MDD remains under 30% while we are starting to see more interesting Sharpe ratios. Given the nature of the reward function, the algortihm is not incentivized to have a caoutious behavior becoming more profitable.

3.2 The Walt Disney Company (DIS)

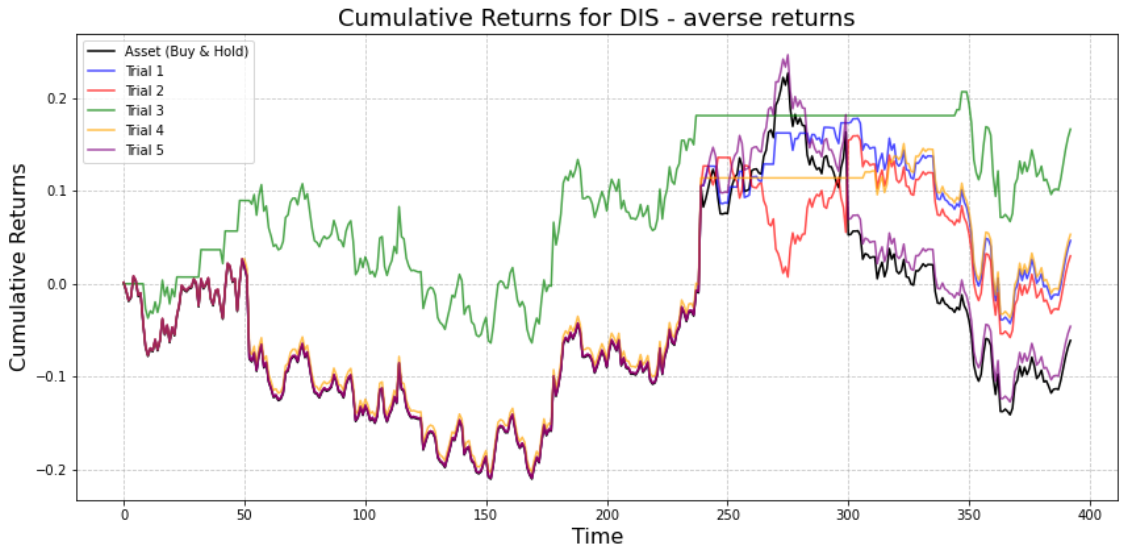

Figure 3.3: DIS algorithm trials vs Buy & Hold. Averse returns.

In the majority of the trials, the DIS algorithm consistently outperforms the Buy & Hold strategy. In a minimal part of the trial, the agent keeps the positions closed and is able to produce consistent profits while the stock loses money in the test set.
The great volatility observed by DIS in the last years of the training process may have elped the algorithm in exploring many different states, rendering it more robust.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	0.05	0.11	0.17	-0.23	0.50	1.12	25.93
trial 2	0.03	0.07	0.11	-0.23	0.48	1.09	64.17
trial 3	0.17	0.51	0.63	-0.15	0.52	1.12	24.60
trial 4	0.05	0.13	0.18	-0.22	0.50	1.08	53.50
trial 5	-0.04	-0.11	-0.15	-0.30	0.50	1.00	130.33

Table 3.4: performance metrics for DIS Averse Returns trials

We do not have very high Sharpe ratios due to the high variance in returns that is clear in the graph. However, the efficiency consistently exceeds or equals one. The average holding period increases significantly, allowing, as one can see from the graph, that in many cases, the agent follows the stock's trend but is then able to detach from it, generating profits.

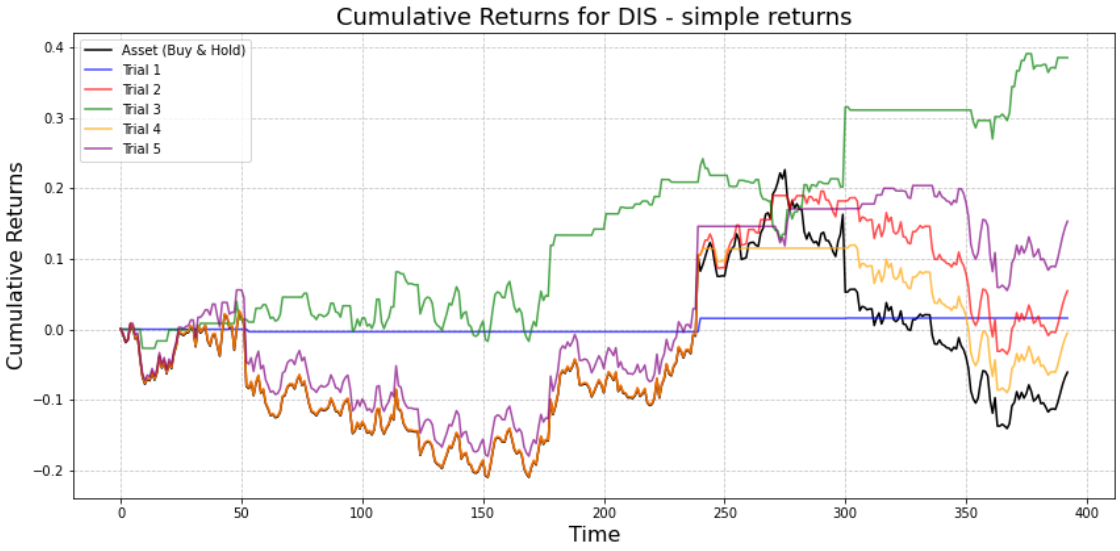

Figure 3.4: DIS algorithm trials vs Buy & Hold. Simple returns.

Also with the simple returns reward function the agent is able to beat the reward earned by the stock. Without the bonus given by going short in turmoil periods, the agent is less incentivized in performing actions, as one can see from Trial 1. Nevertheless, the agent outperforms the stock in all trials, generating profits in 4 out of 5.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	0.02	0.64	0.37	-0.00	0.67	10.38	1.00
trial 2	0.05	0.14	0.20	-0.23	0.50	1.11	30.83
trial 3	0.38	1.22	1.58	-0.09	0.52	2.29	52.57
trial 4	-0.01	-0.01	-0.01	-0.23	0.49	1.06	67.80
trial 5	0.15	0.39	0.51	-0.22	0.51	1.20	14.90

Table 3.5: performance metrics for DIS Returns trials

Trial 3 exhibits strong sharpe ratios and sortino ratios, gaining around 40% in the validation test. Trial 1's trading efficiency is biased due to the agent's inactivity, whereas trial 3 has a good trading efficiency and a decent percentage of successful trades.

3.3 GOLD (GC=F)

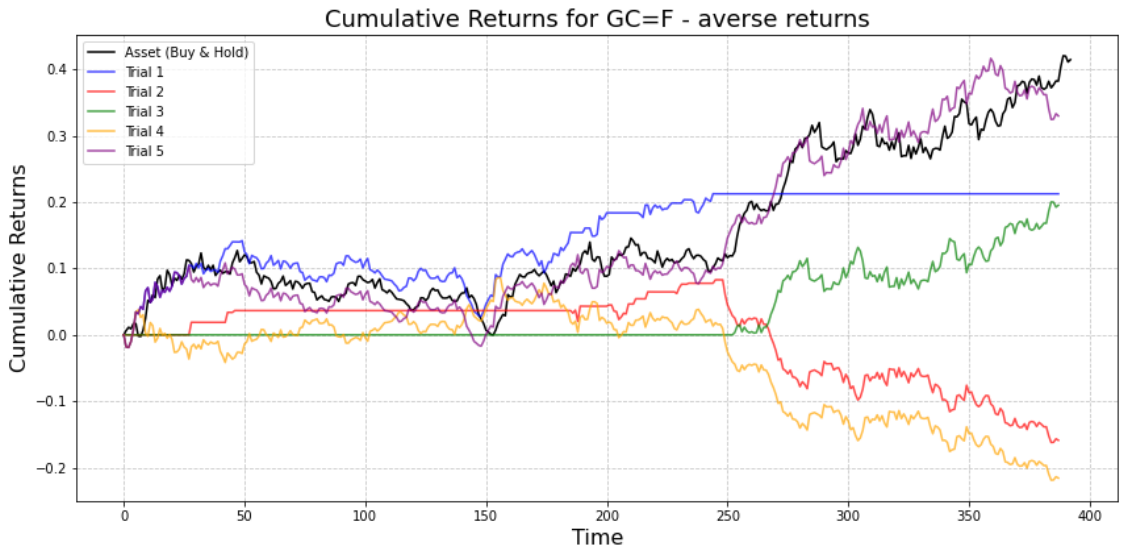

Figure 3.5: GC=F algorithm trials vs Buy & Hold. Averse returns.

As far as GC=F is concerned, the agent is not able to consistently create profits. However, in one trial, the agent was able to outperform the Buy & Hold strategy in certain instances. Given the constant growth of the commodity price, there is a risk that the agent may not receive bonuses for going short during turbulent periods and may face small penalties for going long. This could be a contributing factor to the agent's struggles to act in certain trials.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	0.21	1.31	1.98	-0.10	0.52	1.38	21.30
trial 2	-0.16	-1.11	-1.29	-0.22	0.43	1.10	13.50
trial 3	0.19	1.27	0.96	-0.06	0.59	2.02	135.00
trial 4	-0.21	-1.12	-1.87	-0.28	0.47	0.96	97.00
trial 5	0.32	1.32	1.96	-0.11	0.53	1.10	48.50

Table 3.6: performance metrics for GC=F Averse Returns trials

The MDD is still reasonable, whereas in trial 3, the high sharpe ratio is justified by the inactivity of the agent, which is then able to consistently gain in a very short time span with optimal trading efficiency and a fraction of correct trades.

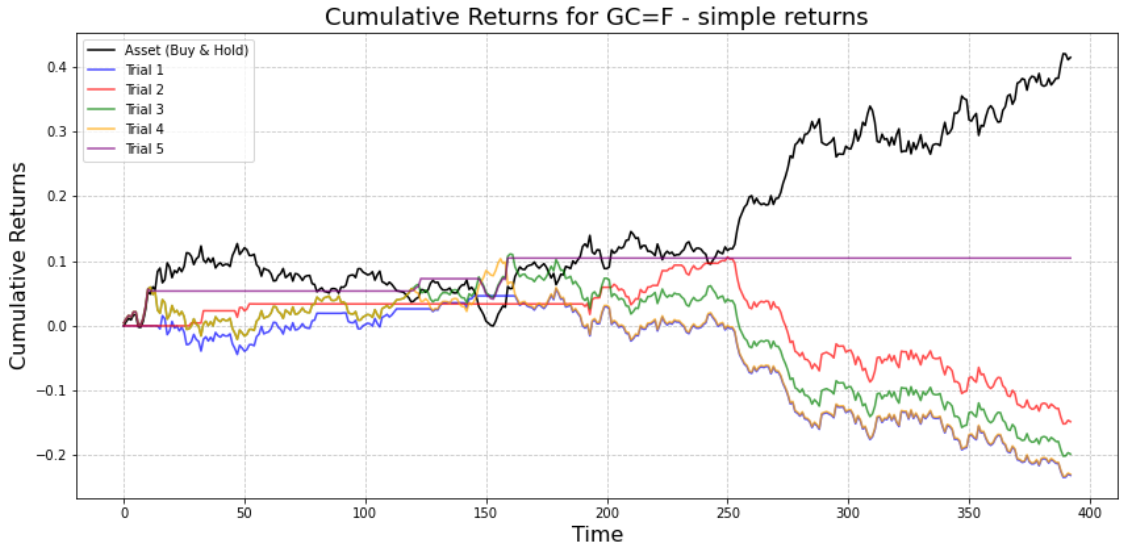

Figure 3.6: GC=F algorithm trials vs Buy & Hold. Simple returns.

In this case, the algorithm is not able to generate consistently high profits and shows similar behavior among each trial. The only profitable trial is given by a few actions with a higher trading efficiency.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	-0.23	-1.32	-2.03	-0.28	0.46	0.97	46.71
trial 2	-0.15	-0.99	-1.22	-0.23	0.46	1.06	14.62
trial 3	-0.20	-1.01	-1.68	-0.28	0.48	0.94	65.50
trial 4	-0.23	-1.19	-1.89	-0.30	0.47	0.92	98.25
trial 5	0.10	1.41	0.81	-0.03	0.61	2.31	9.33

Table 3.7: performance metrics for GC=F Returns trials

The table indicates one of the agent's worst performances. As one can see, in the majority of the cases the agent combines poor trading efficiency with an unsucessful percentage of correct trades. In all the cases, the MDD is bigger than 20%.

3.4 Intel Corporation (INTC)

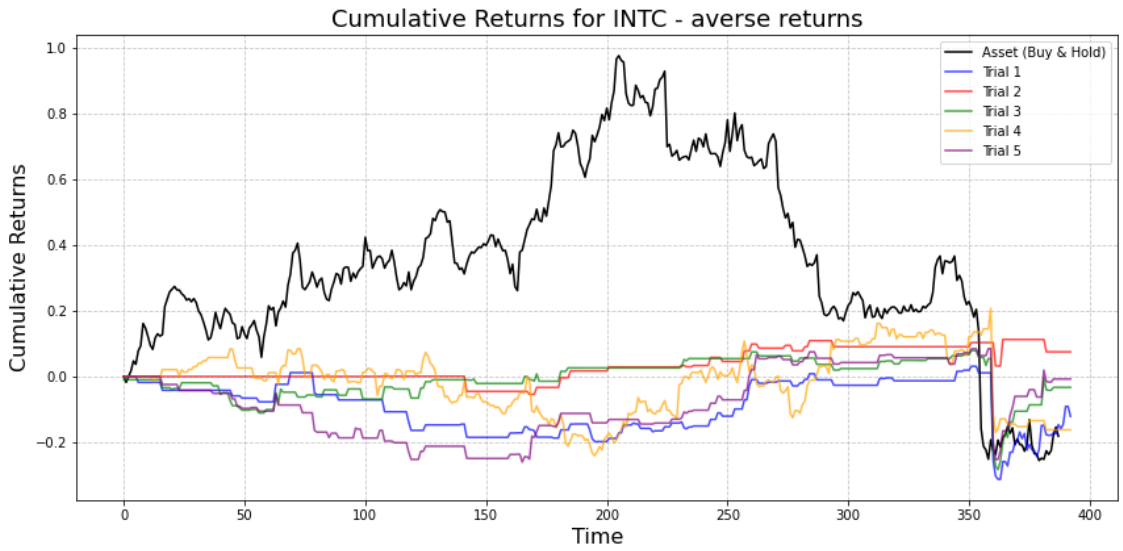

Figure 3.7: INTC algorithm trials vs Buy & Hold. Averse returns.

With Intel Corporation, we have observed the most extreme differences with respect to the two reward functions. In the case of averse returns, the agent struggles to generate profits in many trials. The difference between the two performances may be given by the fact that at the beginning of the test, INTC underwent a dramatic surge. In this type of market condition, the risk-avere returns function may lose its potential. However, during the recent turbulent period, the agent managed to outperform the buy strategy, minimizing losses and even generating a profit in one instance.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	-0.12	-0.27	-0.18	-0.33	0.53	2.02	1.45
trial 2	0.07	0.42	0.20	-0.07	0.63	3.04	1.00
trial 3	-0.03	-0.08	-0.05	-0.34	0.52	1.18	1.18
trial 4	-0.16	-0.29	-0.29	-0.31	0.51	1.43	1.65
trial 5	-0.01	-0.02	-0.01	-0.31	0.54	2.30	1.15

Table 3.8: performance metrics for INTC Averse Returns trials

The agent has an extremely short holding period, deciding carefully when to enter the market. However, the majority of the cases show high MDD. The trading efficiency and the percentage of correct choices are misleading because of the outliers present in the negative returns especially in the last period.

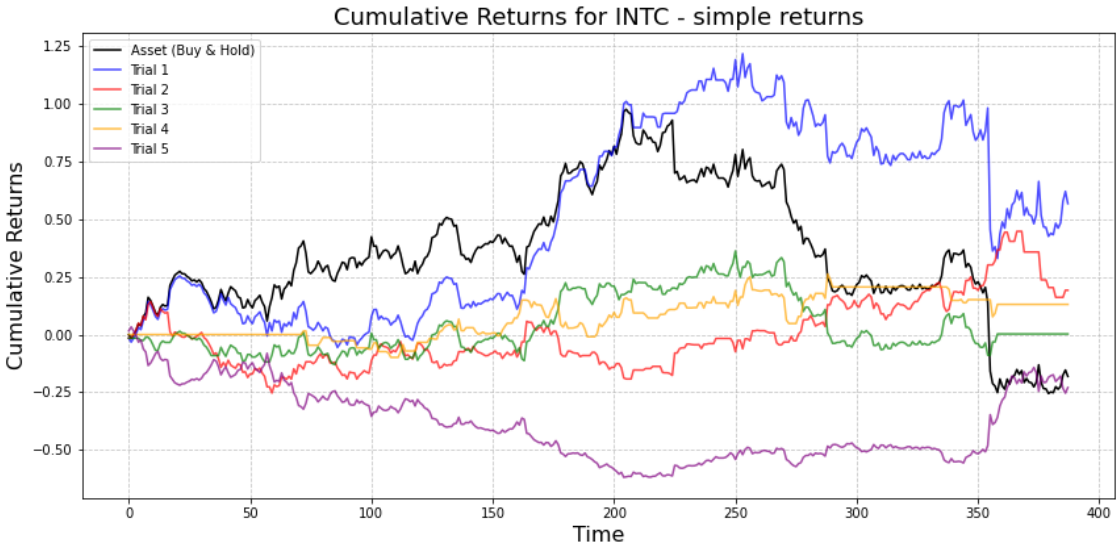

Figure 3.8: INTC algorithm trials vs Buy & Hold. Simple returns.

As confirmed by the results obtained during the hyperparameter tuning, the agent offers one of the best performances with the simple return reward function applied to INTC. While the agent does not consistently generate profits, it outperforms the stock in 4 out of 5 cases. In this case, the agent has learned to act fast, remaining out of the market in very few instances.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	0.57	0.64	0.72	-0.40	0.55	1.08	4.17
trial 2	0.19	0.32	0.46	-0.35	0.49	1.57	1.63
trial 3	0.01	0.00	0.01	-0.33	0.52	1.22	4.54
trial 4	0.13	0.33	0.29	-0.16	0.56	1.58	1.65
trial 5	-0.36	-0.61	-0.23	-0.63	0.48	1.05	3.77

Table 3.9: performance metrics for INTC Returns trials

In the majority of cases, the agent combines good efficiency with successful decisions. Even if in this case the agent showed the highest results, it also showed the maximum drawdown. The average holding period is consistent among the trials. The best trial does not exhibit an excellent sharpe ratio because of the leaks that were observed in the last part of the test.

Similar to DIS, the price volatility of INTC may have facilitated the algorithm's training on a broader range of experiences, thereby enhancing its performance.

3.5 The Coca-Cola Company (KO)

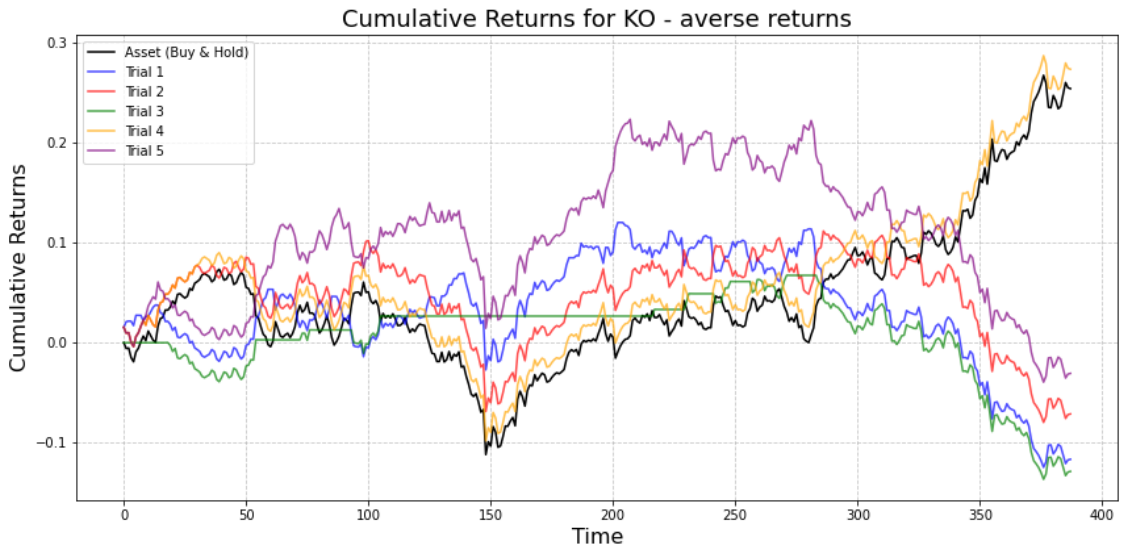

Figure 3.9: KO algorithm trials vs Buy & Hold. Averse returns

We observed that the agent outperformed the stock in 4 out of 5 cases when testing the algorithm on KO, but it failed to respond to the sudden surge in the final months of the test, leading to the loss of all gained profits. From the figure, it is clear that the agent is able to interact constantly with the environment, producing trading decisions that are satisfactory only in the first 300 days of the test.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	-0.11	-0.65	-0.86	-0.22	0.48	1.06	15.17
trial 2	-0.07	-0.38	-0.55	-0.17	0.49	0.96	48.50
trial 3	-0.13	-1.15	-1.14	-0.43	0.44	1.07	16.27
trial 4	0.27	1.24	1.83	-0.17	0.55	1.01	388.00
trial 5	-0.03	-0.16	-0.22	-0.21	0.49	0.99	21.56

Table 3.10: performance metrics for KO Averse Returns trials

The average holding position tells us that the only positive trial is the one in which the algorithms went consistently long during all the tests. In all other instances, the algorithm exhibited a percentage of successful trades below 50%, that was frequently associated with suboptimal efficiency.

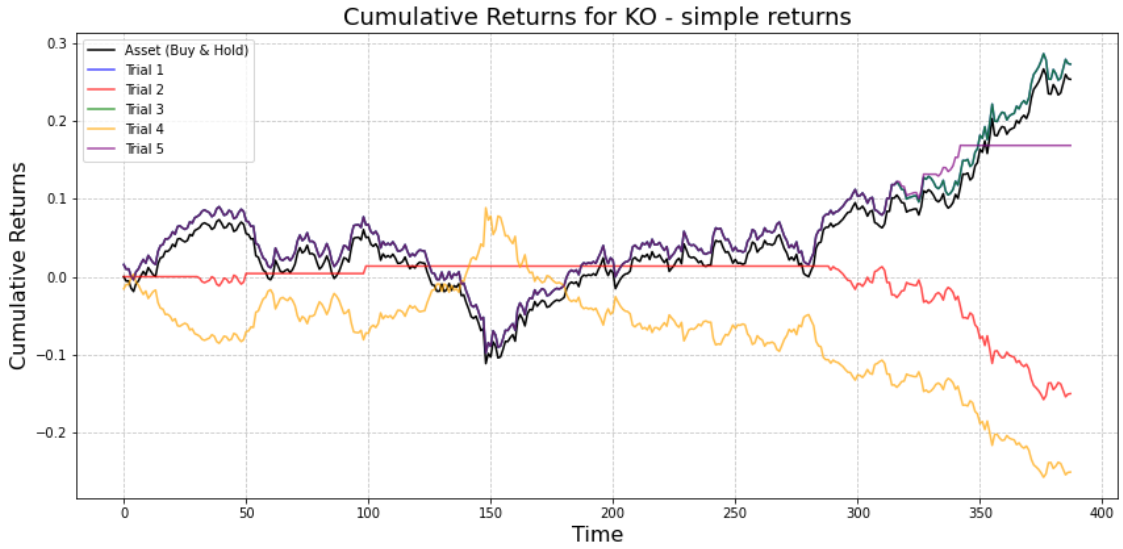

Figure 3.10: KO algorithm trials vs Buy & Hold. Simple returns.

With GC=F, this is one of the most unsatisfactory tests in terms of the algorithm for detaching from the path of the stock. In 3 cases, the algorithm follows the paths of the stock for the majority, or even all the trials. Given that, the algorithm still generates profits in the majority of the cases.

Trial	Return	Sharpe	Sortino	MDD	% C.T.	Efficiency	Avg H.P.
trial 1	0.27	1.24	1.83	-0.17	0.55	1.01	388.00
trial 2	-0.15	-1.62	-1.32	-0.18	0.40	0.91	40.33
trial 3	0.27	1.24	1.83	-0.17	0.55	1.01	388.00
trial 4	-0.27	-1.24	-2.24	-0.32	0.44	0.97	388.00
trial 5	0.17	0.87	1.17	-0.17	0.55	0.99	55.67

Table 3.11: performance metrics for KO Returns trials

The average holding period reveals how the algorithm selects a position and adheres to it in three cases. In the last trial, it still follows the path of the stock for the majority of the case. The remaining trial in which it does not do so show poor trading efficiency and an unsuccessful percentage of correct trades.

Conclusions

The test of the DQN trading algorithm with a LSTM feature extractor on the 5 stocks leads to the following conclusions.
In the majority of the trials, the algorithm was able to generate profits, even though not consistently. We observed the best results in the case of DIS, where the algorithm consistently outperformed the stock returns. The same outcome was observed when we implemented a reward function with simple returns for INTC. In this case, the algorithm also shows potential.
In other cases, like ˆGSPC, using the simple return function, the algorithm was able to generate profits consistently even if not beating the Buy & Hold strategy. In the case of GC=F, the averse returns reward function demonstrated superior performance compared to the simple returns reward function.
In the case of KO, the algorithm faced significant challenges. One of the possible explanations is the steady growth offered by KO, which does not allow the algorithm to explore different states and therefore cannot find answers when market conditions change.
One of the primary issues affecting the algorithm's performance is the computational expense of the learning process. So even if in future work we can increase the number of episodes to 100 or set a threshold for convergence in the Adam optimizer, this will come at a cost.
Another solution could involve conducting a more comprehensive search of the hyperparameters, albeit at a high computational price.
The reward functions could potentially benefit from some future improvements. We can incorporate the opportunity cost of not acting. For example, the agent may receive a small penalty for not acting during a period of growth or for closing positions that later resulted in a gain.
Another improvement that may be applied is in the replay memory, where the agent gives the same importance to all the past experiences, whereas it may be beneficial to give more importance to the ones that produced the higher rewards.
We only used one feature to aid the agent's decision-making; future research should consider adding more features like MACD, RSI, and similar indicators.
Considering all factors, the DQN architecture with an LSTM demonstrated promising results, generating profits in the majority of cases. On the other hand, the need to be exposed

to a lot of different past states and the potential inactivity of the agent are some caveats that can be resolved with improvements in the management of the replay memory and with a better reward assignment when the agent is out of the market.

Bibliography

[1] Frank Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[2] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[4] Christopher JCH Watkins. *Learning from delayed rewards*. Phd thesis, King's College, University of Cambridge, 1989.

[5] John Moody and Matthew Saffell. Reinforcement learning for trading. *Advances in Neural Information Processing Systems*, 12:917–923, 1999.

[6] Yuriy Nevmyvaka, Yi Feng, and Michael Kearns. Reinforcement learning for optimized trade execution. In *Proceedings of the 23rd international conference on Machine learning*, pages 673–680. ACM, 2006.

[7] Yiting Deng, Feng Bao, You Kong, Zhiwei Ren, and Qionghai Dai. Deep direct reinforcement learning for financial signal representation and trading. *IEEE Transactions on Neural Networks and Learning Systems*, 28(3):653–664, 2016.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[9] Anastasia Borovykh, Sander Bohte, and Cornelis W Oosterlee. Dow jones trading with deep learning: The unreasonable effectiveness of recurrent neural networks. *arXiv preprint arXiv:1706.04013*, 2017.

[10] Zhuoran Jiang, Dong Xu, and Jinjun Liang. Improving financial trading decisions using deep q-learning: Predicting the number of shares, action strategies, and transfer learning. *Expert Systems with Applications*, 108:181–199, 2017.

[11] Xiangyu Chang Zhiyi Zeng, Cong Ma. Multi-step reward ensemble methods for adaptive stock trading. *Expert Systems With Applications*, 230(120547), 2023.

[12] Wanshan Zheng Yang Li and Zibin Zheng. Deep robust reinforcement learning for practical algorithmic trading. *IEEE Access*, 7:108014–108022, 2019.

[13] Ji Cao, Jun Chen, and Jianjun Liang. Deep reinforcement learning pairs trading with a double deep q-network. *Expert Systems with Applications*, 165:113700, 2021.

[14] Thibaut Théate and Damien Ernst. An application of deep reinforcement learning to algorithmic trading. *arXiv preprint arXiv:2004.06627*, 2020.

[15] Mehran Taghian, Ahmad Asadi, and Reza Safabakhsh. Learning financial asset-specific trading rules via deep reinforcement learning. *Expert Systems with Applications*, 195:116523, 2022.

[16] Ethem Alpaydin. *Introduction to Machine Learning*. MIT Press, Cambridge, MA, 4th edition, 2020.

[17] Charu C. Aggarwal. *Neural Networks and Deep Learning*. Springer, Cham, Switzerland, 2023.

[18] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, New York, 2011.

[19] Herbert Robbins and Sutton Monro. A stochastic approximation method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951.

[20] Jack Kiefer and Jacob Wolfowitz. Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[21] Frank Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. Spartan Books, Washington, D.C., 1962.

[22] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989.

[23] Kunihiko Fukushima. Visual feature extraction by a multilayered network of analog threshold elements. *IEEE Transactions on Systems Science and Cybernetics*, 5(4): 322–333, 1969.

[24] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). In *4th International Conference on Learning Representations (ICLR)*, 2016.

[25] Michael I. Jordan. Serial order: A parallel distributed processing approach. Technical Report Report No. 8604, University of California, Institute for Cognitive Science, San Diego, CA, 1986.

[26] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.

[27] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. In *Proceedings of the 9th International Conference on Artificial Neural Networks (ICANN 99)*, pages 850–855. IET, 2000.

[28] Ronald J Williams and David Zipser. A learning algorithm for continually running fully recurrent neural networks. *Neural computation*, 1(2):270–280, 1989.

[29] Ronald J Williams and Jing Peng. An efficient gradient-based algorithm for on-line training of recurrent network trajectories. *Neural Computation*, 2(4):490–501, 1990.

[30] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2 edition, 2018.

[31] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[32] Ronald A. Howard. *Dynamic Programming and Markov Processes*. MIT Press, Cambridge, MA, 1960.

[33] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, 1988.

[34] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3-4): 279–292, 1992.

[35] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[36] William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3-4):285–294, 1933.

[37] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[38] Ha Young Kim Jeong Gyeeun. Improving financial trading decisions using deep q-learning: Predicting the number of shares, action strategies, and transfer learning. *Expert Systems with Applications*, 117:125–138, 2019.

[39] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, 2016.

[40] PyTorch Documentation. Broadcasting semantics, 2023. URL `https://pytorch.org/docs/stable/notes/broadcasting.html`. Accessed: 2024-09-26.

[41] PyTorch Documentation. torch.nn.layernorm, 2023. URL `https://pytorch.org/docs/stable/generated/torch.nn.LayerNorm.html`. Accessed: 2024-09-26.

[42] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

[43] Shaoqing Ren Jian Sun Kaiming He, Xiangyu Zhang. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, pages 1026–1034, 2015.

[44] PyTorch Documentation. torch.nn.init — pytorch 2.0.1 documentation, 2023. URL `https://pytorch.org/docs/stable/nn.init.html`. Accessed: 2024-09-27.

[45] Andrew L. Maas, Awni Y. Hannun, and Andrew Y. Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, 2013.

[46] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[47] PyTorch Documentation. torch.nn.smoothl1loss, 2023. URL `https://pytorch.org/docs/stable/generated/torch.nn.SmoothL1Loss.html`. Accessed: 2024-09-28.

[48] Peter J. Huber. Robust estimation of a location parameter. *The Annals of Mathematical Statistics*, 35(1):73–101, 1964.

[49] Optuna. Optuna: A hyperparameter optimization framework, 2023. URL `https://optuna.org/`. Accessed: 2024-09-28.

[50] William F. Sharpe. Mutual fund performance. *The Journal of Business*, 39(1):119–138, 1966.

[51] Frank A. Sortino and Robert van der Meer. Performance measurement in a downside risk framework. *The Journal of Investing*, 3(3):59–64, 1994.

[52] Malik Magdon-Ismail, Amir F. Atiya, Ananda Pratap, and Yaser S. Abu-Mostafa. *Maximum Drawdown*, volume 17. 2004.