7CCSMPRJ Final Year
Automating Forex Trade with Deep Reinforcement Learning

Final Project Report

Author: Jordan Truong

Supervisor: Peter McBurney

Student ID: 1638008

August 24, 2021

Executive Summary

This project report proposes a method in automating trade in the foreign exchange market utilising a Deep Reinforcement Learning algorithm; Deep Q-Network. This trading agent has attempted to tackle the problem with quantifying trades in which there exists an absence within most common and popular trading strategies. This is performed by implementing two streams of inputs and applying a linear activation function to quantify how many units to buy or sell against a specific market trend. The function approximator will consist of layers from different types of Neural Networks that have proven predictive power within the stock market domain. These Neural Networks consists of Artificial Neural Networks, Convolutional Neural Networks, and Recurrent Neural Networks. After conducting experimentation to design the structure of the model and select principal parameters for the function approximators, these trading agents were tested against popular trading strategies to assess its performance within the High Frequency Trading domain.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 15,000 words.

Jordan Truong

August 24, 2021

Acknowledgements

I would like to thank my supervisor, Peter McBurney, for his support and guidance in developing this project as well as overseeing my entire progress from beginning to end.

Contents

1 Introduction **3**
1.1 Aims and Objective . . . 4

2 Background **6**
2.1 Related Works . . . 8

3 Technical Background **9**
3.1 Deep Learning . . . 10
3.1.1 Biological Neurons . . . 10
3.1.2 Artificial Neurons . . . 10
3.1.3 Artificial Neural Networks . . . 11
Backpropagation . . . 13
3.1.4 Convolutional Neural Networks . . . 14
Convolutional Layers . . . 15
Pooling Layers . . . 16
Fully-connected Layer . . . 17
3.1.5 Recurrent Neural Networks . . . 17
LSTM . . . 18
3.2 Reinforcement Learning . . . 19
3.2.1 Markov Decision Process . . . 20
3.2.2 Continual and Episodic Tasks . . . 20
3.2.3 State-value function and Action-value function . . . 21
3.2.4 Q-Learning . . . 22
3.2.5 Exploration vs Exploitation . . . 22
3.2.6 Deep Q-Learning . . . 23
3.2.7 Experience Replay . . . 24
3.2.8 Target Network . . . 24

4 System Design **26**
4.1 Agent and Environment . . . 26
4.1.1 Function Approximator Design . . . 27
4.2 Methodology . . . 28
4.2.1 Exploration vs. Exploitation . . . 29
4.2.2 Reward Systems . . . 30
4.2.3 Q-Learning Parameters . . . 31

5 Experimental Results **34**
5.1 ANN Function Approximator 34
5.2 CNN Function Approximator 35
5.3 RNN Function Approximator 35
5.4 All Models 35

6 Conclusion **41**
6.1 Conclusion 41
6.2 Future Works 42
6.3 Project Evaluation 43
Bibliography 46

Chapter 1

Introduction

Reinforcement Learning has been an active area in research, enabling users to automate tasks by finding optimal solutions and revolutionize how intelligent systems are able to interact with humans in everyday life. There has been a huge number of applications in modern society ranging from robotics and the recent attention from autonomous vehicles [6, 18], but the purpose of this project is to assess whether Reinforcement Learning has the same successful application with automating trade in the foreign exchange market. Hedge Funds have implemented such applications in the past, but to maintain competitive advantage, many of the work has not been published and the advancements of this domain has remained undisclosed.

There has been a number of recent novel algorithms in forecasting the stock market with profitable margins utilising Deep Neural Networks [15], but this paper has utilised similar pattern recognition techniques as function approximators to determine executable trading actions in the Q-Learning model, also known as Deep Q-Learning. This project has tackled variations of the algorithm by first selecting the optimal parameters for Q-Learning then assessed the different function approximators utilising different areas of Deep Learning, such as: Artificial Neural Networks (ANNs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs). These types of Neural Networks were specifically chosen as function approximators due their dissimilar and unique properties in recognising patterns in timeseries data, more detail will be explained in Chapter 3.

The inspiration for using pattern recognition techniques in this domain was inspired by traders and their use of technical analysis. Modern day traders are able to make profitable predictions in the stock market by utilising technical indicators which are transformed and manipulated versions of existing raw data that provides informative insights and better judgement on the market as compared to the existing raw data. Sophisticated pattern recognition techniques in Deep Learning are utilised in hopes of finding similar or potentially dissimilar patterns to create an effective trading agent.

1.1 Aims and Objective

The aim of this project is to assess the efficacy and reliability of a profitable trading strategy utilising Deep Reinforcement Learning in the foreign exchange market. The parameters of these implementations are adjusted according to a single currency exchange dataset then re-trained onto different datasets with the same parameters. This is to simulate a real and practical implementation where training and validating a model can take an extensive amount of time such that the process for parameter selection cannot be easily reperformed onto different markets.

The performance of the implementations will be assessed on a variety of different metrics to measure the frequency of trades, time taken to execute a trade, reliability, and most importantly, the overall profitability of the strategy. The intention of this project is not to practically implement these algorithms into real life trades and out-perform the market, but to evaluate the feasibility and robustness of these strategies as compared to popular and conventional strategies in high frequency trading.

To create such a model, the following steps are required to gain a comprehensive understanding of how the model operates and provide a consistent methodology to conduct a fair evaluation on its performance and underlying properties:

1. Understand the technical background of Reinforcement Learning and Deep Learning, and how they overlap in the field of Deep Reinforcement Learning
2. Explore different variations of the Q-Learning model to initialise as the base model
3. Evaluate and select model parameters for Q-Learning
4. Evaluate hyperparameters of different Neural Networks to act as function approximators for the DQN model
5. Assess and test the performance between all variations of the DQN model

6. Re-train the model and assess its performance onto different currency exchange datasets

7. Compare the performance of the final models against popular trading algorithms

Chapter 2

Background

There exists an abundance of research in Deep Reinforcement Learning in a variety of different domains with Deep Q-Network (DQN) as one of the most popular algorithms. This is due to its off-policy properties allowing it to enable experience replay to reuse past experiences in the learning phase of the model. Google Deepmind has proven that DQN has been able to outperform world-renowned athletes in video games such as Atari games and Starcraft II [10, 11]. This shows that DQN agents are able to develop strategies and conduct educated decisions to optimise its performance in complex domains.

Generally within these research papers, Artificial Neural Networks [29, 32] and Long Short-Term Memory [20, 21] were implemented as function approximators with few that utilised Convolutional Neural Networks [23]. The paper that implemented CNNs required stock chart images as input to detect price patterns, whereas in this paper all inputs are consistently closing price data and will be fed into a 1-dimensional Convolutional layer in the case of a CNN function approximator. Function approximators are models that are employed to estimate the quality of an action given a state, these are called Q-values where the Q is an abbreviation for 'quality'. This represents how useful that action is in maximising its future rewards where more detail will be explained in section 3.2.4. Function approximators with ANN and LSTM has been proven to be successful in detecting underlying patterns in sequential data to execute trades as it has for forecasting stock prices [25]; CNNs has also proven to be successful in time series applications [7] with its property to realise relationships and information between data points in input data. There has been little research that compares the three variations of Neural Networks as function approximators in the DQN model, so this paper will evaluate these three variations under the same datasets.

In research, a sigmoid activation function, commonly tanh or softmax, is used for the final layer of the Neural Network [20, 32]. This enables the function approximator to decide an action by rounding the output to the nearest integer where one of three actions are chosen: buy, hold, or sell. Although this has proven to be effective, this is also a limitation to the model as this does not provide the system a quantity to trade and only processes a defined unit at a time. This project has implemented a linear activation function in the final layer within the function approximator to enable the agent to decide which of three actions to be processed along with its respective quantity. The intuition for this system is to provide the agent a level of confidence to each trade where if it believes an outbreak will emerge then a high quantity of trade will be executed. Conversely, smaller trades will be executed if the prediction confidence is low or it believes the input pattern is not worth trading. An output greater than zero will be the amount of units to sell at the given price and an output less than zero will be the units to buy at the given price, the function is below:

$$f(x) = \begin{cases} x < 0 & \text{sell x units from inventory} \\ x = 0 & \textit{hold} \\ x > 0 & \text{buy x units with bank} \end{cases}$$

Due the design of the system, the model output will have restrictions as a consequence to implementing a bank and inventory to the agent's design. These restrictions includes buying higher than the agent's bank or selling more than the capacity of the inventory (shorting is not available), which leads to requiring data from the agent's environment (bank and inventory) to decide an action and its respective quantity. This then leads to the function approximator having two streams of inputs, one for the 2-dimensional sequential closing price data and the environmental data; resulting in a concatenated Neural Network. This will be largely different from the Neural Networks in the previous research papers as they typically have a single streamed Neural Network where only the closing prices are the inputs and there is a single output dictating an action.

2.1 Related Works

Deep Reinforcement Learning has also been applied to other trading strategies such as Pair Trading [1], this has enabled it to simultaneously trade between two stocks which are highly correlated with a single agent. This has been a successful area of research where more sophisticated algorithms such as Deep Deterministic Policy Gradient (DDPG) and Duelling Q-Network were implemented [2, 13]. DDPG is similar to DQN such that it uses Neural Networks and experience replay, but applied to the Deterministic Policy Gradient algorithm instead of Q-Learning. Duelling Q-Network is similar to DQN, but it contains two streams in the Neural Network which separates the state-action pairs and state-dependant action advantages [12].

One of the problems of using Neural Networks as function approximators in Reinforcement Learning is that it has the tendency to overfit the data as especially within Q-Learning, the agent tends to give a systematic over-estimation of the utility values [30]. Google Deepmind overcame this by developing the algorithm Double Deep Q-Network (DDQN) that reduces overestimations by decomposing the max operation in the target into action selection and action evaluation [17]. Although all of this would be an interesting area of research to be implemented as a continuation of DQN in this domain, it is outside the scope of this paper.

Chapter 3

Technical Background

Machine Learning is an extensive area of research in Artificial Intelligence where the study of algorithms is utilised to learn underlying patterns from data and improve with experience [32]. Within Machine Learning, there is an abundance of different subfields each with their own unique applications and process of learning, but the two main areas that relates to this project is Deep Learning and Reinforcement Learning.

Deep Learning is a subfield of Supervised Learning, which is defined as a mapping from input data to labelled data [32]. These mapping functions are built using Neural Networks which is a single connected network in a layered structure where each edge corresponds to a weight and each node contains defined operations and functions. The predictions are then calculated by propagating inputs through the network which is a pass of the input values by multiplying it by the connection weights and computing the node operations until the output layer is reached.

Reinforcement Learning is the process of learning what to do without explicitly provided with the guidance of what to do. This is computed by mapping the agent's observed environment to actions by optimising a defined numerical reward system [29]. A simple example of this would be the arcade game 'Snake'. If we define the scoring system to gain +1 when a food item is eaten and -10 when the agent loses the game, this will intuitively lead to the agent aiming to eat as many food items as possible without losing the game; this will in turn maximise the score of the system.

3.1 Deep Learning

3.1.1 Biological Neurons

The basis of Artificial Neural Networks is based on biological sciences in components of how the brain operates. In neurobiology, a neuron is a nerve cell that serves as a unit of the brain where it communicates with electrical signals in the membrane. The connections between the neurons are connected by electrochemical junctions called synapses which are located at the branches of a cell called dendrites, this is shown in the Figure 3.1. [3].

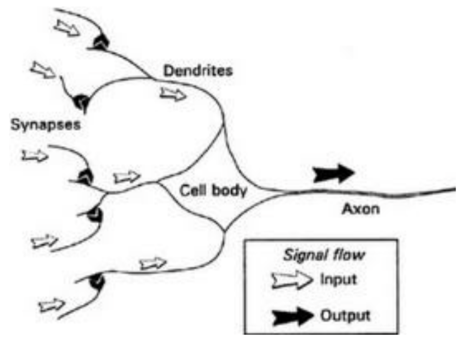


Figure 3.1: A diagram of a biological neuron [44].

Typically, neurons receive connections from thousands of other neurons. This large number of connections is filtered by theoretically applying a threshold to the summation of the connections and generating a voltage impulse in response depending on whether the threshold is met. The voltage impulse is sent through a branching fibre called an axon to be transmitted to the other neurons [3]. In determining whether an impulse is to be generated by the neuron, it is dependent on the type of the incoming signals to the biological neuron. The two types of incoming signals are inhibitory and exhibitory; inhibitory signals tends to have an effect of preventing firing whereas exhibitory signals promotes the generation of impulses [3].

3.1.2 Artificial Neurons

In Deep Learning, the architecture of a group of connected biological neurons is represented as a graph in a layered structure. Each neuron is represented by a node, also called an artificial neuron, and the synapses are the network edges where it contains a corresponding weight. Similar to a biological neuron, an artificial neuron receives incoming signals from other connected neurons and transmit a signal after calculating the summation of the signals and applying an activation function.

Below, we have a figure of a single artificial neuron. This neuron receives signals from 4 different sources, x_i, where each signal is multiplied by its corresponding connection weight, w_i. After calculating the summation of the weighted signals, an activation function $f(x)$ is applied. There are many activation functions known to be used in practice, but the most popular is the rectified linear unit (ReLu) activation function which takes the maximum between the current value and 0. The other activation function that is used in this project is the linear activation function which is also shown below.

$$y = f\Big(\sum_{n=i}^{N} w_i x_i\Big)$$

$$ReLu(x) = max(x, 0)$$

$$Linear(x) = x$$

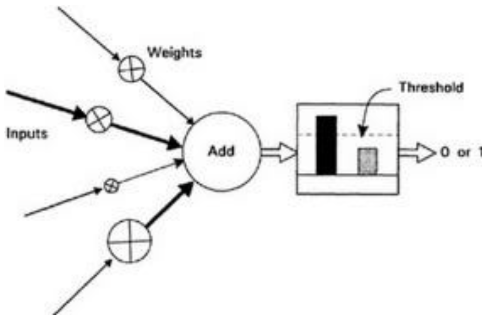


Figure 3.2: A diagram of an artificial neuron [44].

3.1.3 Artificial Neural Networks

Each Artificial Neural Network can be comprised of 3 types of layers: an input layer, a hidden layer, and an output layer. The figure below displays a Neural Network with each of the defined layers, this is also known as a 3-layered Neural Network as it comprises of a total of 3 layers. A Neural Network can contain more than one of the layers mentioned, but for the simplicity of this explanation, we will focus on the 3-layered Neural Network. Any Neural Network comprised with more than 1 hidden layer is considered as a Deep Neural Network (DNN).

Each node within the input layer, x_i, represents a data point within the input data of length

d. No connection is fed into the input layer as this is first layer for computation and only used to pass the weighted connections to the next layer. In each of the layers with the exception of the output layer, there exists a bias which is displayed as the final node in each layer with no previous connections. This is a constant to help shift the activation function and works similarly to how we would need an intercept in a linear line [26].

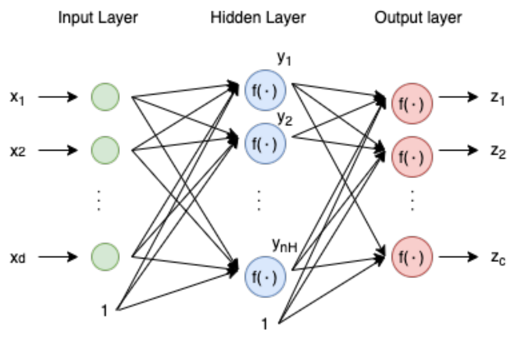


Figure 3.3: A diagram of 3-layered Neural Network.

Every neuron represented as y_j in the hidden layer of length n_H has the same properties as the artificial neuron. Each node within the hidden layer has a connection to all the nodes in the previous layer (which in this case is the input layer) including the bias and is multiplied by its corresponding weight. After calculating the summation and applying the activation function, an output is produced to be fed as a connection to the next hidden layer or in this case, the output layer. The weight corresponding to a connection from the i^{th} node in the previous layer to the j^{th} node in the current layer is represented as w_{ji} and the activation function is denoted as $f(\cdot)$. The net activation is as followed:

$$net_j = x_1 w_1 + ... + x_d w_{jd} + w_{j0} = \sum_{n=i}^{d} x_j w_{ji} + w_{j0} = \sum_{n=0}^{d} x_i w_{ji} = \boldsymbol{w}_j^T \boldsymbol{x}$$

$$where\ \boldsymbol{w}_j = \begin{bmatrix} w_{j1} \\ \vdots \\ w_{jd} \\ w_{jo} \end{bmatrix}\ and\ \boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_d \\ 1 \end{bmatrix}$$

$$y_j = f(net_j) \tag{1}$$

The output layer is the final layer of the ANN, it similar to the hidden layers where it produces an output z_k of length c to which an activation function is applied to the weighted summation of the connections from the previous layer. Dependant on the application, the output layer can be interpreted in many ways. In classification, each output node can represent the probabilistic representation of each class, but for the scope of this paper, the main focus will be on regression. For regression, we only seek for a single value, so only one node is required for the output layer and is mapped onto a continuous domain. The computation for the output layer is a similar computation to the hidden layer, the two computations can be combined to a single equation to compute the final output of the 3-layered Neural Network from only the input data and connection weights. This is shown in equation (3) where equation (1) has been substituted and expanded into equation (2).

$$net_k = y_1 w_{k1} + ... + y_{n_H} w_{kn_H} + w_{k0} = \sum_{j=1}^{n_H} y_j w_{kj} + w_{k0} = \sum_{j=0}^{n_H} y_j w_{kj} = \boldsymbol{w}_k^T \boldsymbol{y}$$

$$g_k(\boldsymbol{x}) = z_k = f(net_k) = f\Big(\sum_{j=0}^{n_H} y_j w_{kj}\Big) = f\Big(\sum_{i=0}^{n_H} w_{kj} f(net_j)\Big) = f\Big(\sum_{i=0}^{n_H} w_{kj} \sum_{i=1}^{d} w_{ji} x_i\Big) \tag{2}$$

$$\boldsymbol{Z} = \begin{bmatrix} z_1 \\ \vdots \\ z_c \end{bmatrix} = f(\boldsymbol{W}_{kj}\boldsymbol{Y} + \boldsymbol{W}_{k0}) = f(\boldsymbol{W}_{kj} f(\boldsymbol{W}_{ji}\boldsymbol{x} + \boldsymbol{W}_{j0}) + \boldsymbol{W}_{k0}) \tag{3}$$

Backpropagation

After computing the output with the given inputs, we can evaluate the performance of the predictions to its corresponding labelled output in the feed-forward network. Dependant on the model we are trying to achieve, there can be many ways to evaluate a model with different loss functions. Since the application is regression, Mean Squared Error can be used for evaluation where it is the squared difference between the prediction and the true output.

Backpropagation is one of simplest methods for training a feed-forward network, it is based on the minimisation algorithm gradient descent where it minimises the differences between the predicted and true output [4]. This enables the model to modify the weights of the Neural Network with the aim of achieving an improved accuracy in the next iteration. The algorithm is below, where $J(\cdot)$ denotes the loss function and Δw_{ji} denotes the change in weights which we use to update the weights of the network. The equation contains a derivative of the loss function with respect to the weights of the ANN, therefore, an essential constraint is that the loss function is also differentiable [4]. The proof is beyond the scope of this paper.

$$\delta_k = -\frac{\partial J}{\partial net_k}$$

$$\Delta w_{ji} = -\eta \frac{\partial J}{\partial w_{ji}} = -\eta \delta_j x_i = \eta f'(net_j)\Big[\sum_{k=1}^{c} w_{kj}\delta_k\Big] x_i$$

$$where\ i = 0, ..., d;\ j = 0, ..., n_H;\ k = 1, ..., c$$

$$w_{ji}(m+1) = w_{ji}(m) + \Delta w_{ji}(m)$$

$$where\ i = 0, ..., d;\ j = 0, ..., n_H;\ k = 1, ..., c;\ x_0 = 1,\ y_0 = 1$$

3.1.4 Convolutional Neural Networks

There exists a couple of problems implementing ANNs with dimensional data where there exists information between the data points such as images and signals. If the size of the dimensions of the input data is high, this then can lead to an exponential increase of the amount of connections required between the layers and therefore require high computational power and furthermore, longer training time. Another problem is that ANNs also tends to overfit with dimensional input data, because it is unable to detect generalised features from reduced forms of the input data when passed through the layers [24].

This problem is solved by including additional types of layers that reduces the size of the preceding data whilst generally maintaining the underlying patterns and furthermore, reducing the computational power to train the model. The main differentiator between CNNs and ANNs are the types of layers involved within the architecture. The additional layers within a CNN enables it detect common patterns within the dimensional input data; these layers are convolutional layers, pooling layers, and fully connected layers.

CNNs have been successful in its application in Computer Vision where it was first introduced to classify hand-written digit images [14] and now well-known for object detection where it is able to classify thousands of different objects in a variety of different domains. Within this paper, we are interested to explore its application in detecting generalised patterns in signals, more specifically, timeseries data in the financial markets. Traders use technical analysis to observe and predict movements in the market using transformed representations of the data as indicators; with proven research in successful predictions in signal processing and time series [7], we will explore its application in creating an automated trading agent which has the same intuition as technical indicators by detecting reoccurring patterns leading to similar outcomes.

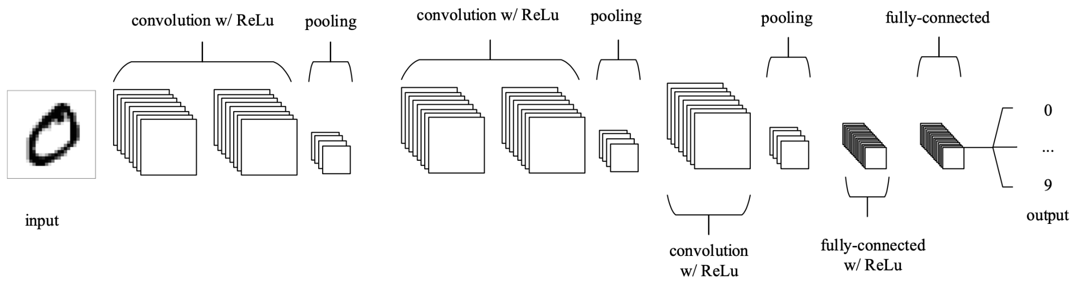


Figure 3.4: A diagram of a CNN with a convolutional layer, pooling layer and a fully-connected layer [24].

Convolutional Layers

Convolutional layers are focused on the properties of kernels; kernels are the generalised patterns of the incoming data which correlate to the output class. Kernels are generally smaller in size than the input data so that it can be compared to segments of the input data using the sliding windows technique. Dependant on the parameters of the convolutional layers, it can greatly reduce the overall complexity of the model and reduce training time. The parameters of the convolutional layers are depth, stride, and zero-padding.

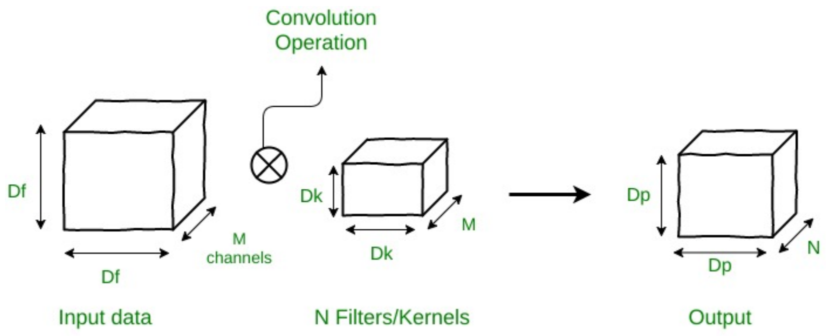


Figure 3.5: A diagram of a convolution operation in a Convolutional Neural Network [16].

The depth of the output volume of the convolutional layer and can be similarly seen as the amount of the neurons within a hidden layer. Large amounts of neurons within any layer of an ANN will lead to more connections between the layers, creating higher complexity. Similarly, decreasing the depth of a convolutional layer will reduce the amount the connections and lower the complexity, but will be a trade-off in reducing its pattern recognition capabilities [24].

Stride is stepping size for the sliding window technique. Setting a large stride avoids overlapping between the receptive field of the input, but too large of a stride can miss features and produce an output of lower dimensions.

Zero-padding is a technique to maintain the dimensionality of the input data, this places a border of 0 entries around the input data to ensure the dimensions of the convoluted output is the same as the input data.

Pooling Layers

The purpose of a pooling layer is to reduce the dimensionality of the preceding layer and furthermore, reduce the complexity and training time of the model. This works by applying a defined operation on non-overlapping segments of the data; popular pooling layers are the max pooling and average pooling where they take the maximum and average value of the segmented data respectively.

Due to the erratic movements of financial markets data, max pooling layers may not be useful as it will give false indications of movements of the data so only average pooling would be applicable for this project. An example is shown in Figure 3.6 where a max pooling and an average pooling of size 2x2 is applied.

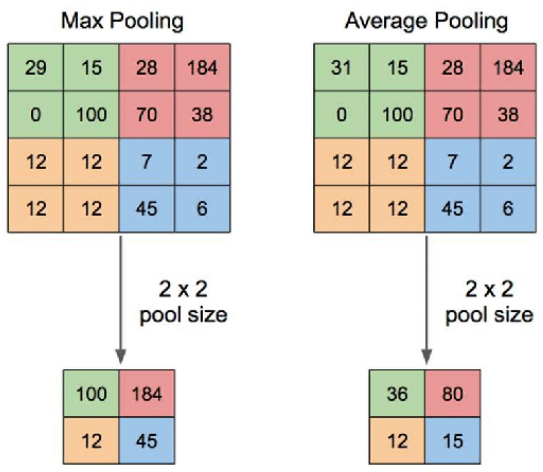


Figure 3.6: A diagram of appling a 2x2 Max and 2x2 Average pooling to a 4x4 matrix [8].

Fully-connected Layer

A fully-connected layer is similar to a hidden layer in an ANN where it exists as a single layer contained with only artificial neurons. This is typically used to transform complex structured layers, such as convolutional layers, by vectorising the data into a layer that is similar to a hidden layer in an ANN. This must be the final layer of the CNN to ensure that it is in the same dimension and format as the labelled output data.

3.1.5 Recurrent Neural Networks

A useful property of technical indicators are price patterns, traders often remember price patterns that will provide indications of specific movements in the market. ANNs do not have this property where it remembers patterns in sequential data that can lead to high probabilistic outputs, but Recurrent Neural Networks (RNNs) contains this property which is why it makes it advantageous in this application.

RNNs process temporal information by adding an additional connection from each node to the next iteration as an additional output. RNNs are trained by unfolding these recurrent connections for each timestep such that the architecture is similar to a DNN and backpropagation can be similarly applied for training [27].

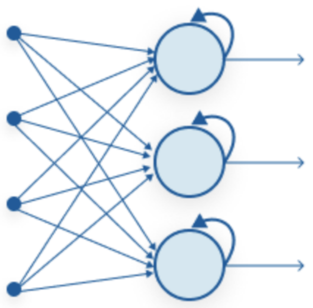


Figure 3.7: A diagram of a Recurrent Neural Network [31].

Problems with RNNs are the vanishing and exploding gradient problem. Vanishing gradient occurs when backpropagating and the error is consistently small, this will lead to the gradient becoming exponentially small though each iteration and furthermore, neurons in earlier layers learn more slowly and contribute little to the overall computation and reduce its ability in learning. Exploding gradient occurs when the errors are consistently large when backpropagating leading to the gradients becoming exponentially large. This can be problematic as this makes neurons in the earlier layers make random and erratic changes, moreover, making it difficult for the network to learn [19]. Long Short-Term Memory (LSTM) addresses both of these problems by having properties to maintain state information with components called memory cells and gates.

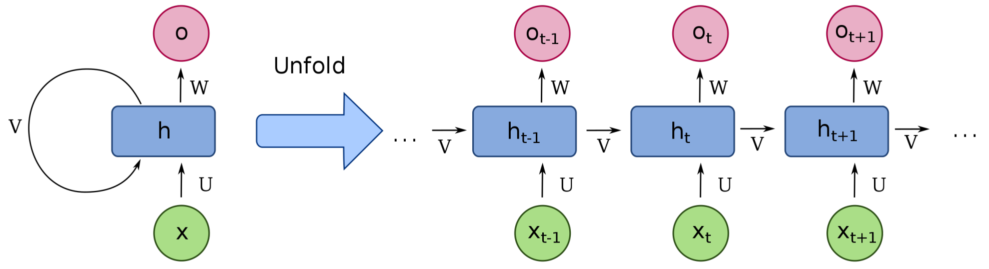


Figure 3.8: A diagram of a unfolded Recurrent Neural Network [5].

LSTM

The concept of LSTM is that it has a cell state that carries relevant information to each iteration where information is added or removed from the three gates: input gate, forget gate, and output gate. In Figure 3.9, we can see a memory cell within a LSTM layer, the black boxes represent a bit-wise operation and the circles represent a neural network layer with a sigmoid or tanh activation function.

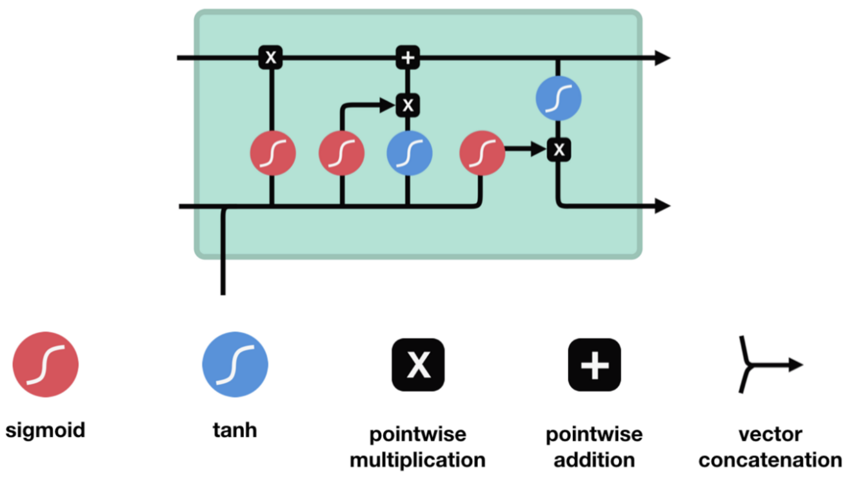


Figure 3.9: A diagram of a memory cell in a LSTM layer [22].

The forget gate is operated by a neural network layer with a sigmoid activation function. This is fed with information from the previous hidden state and the current timestep to decide which information is deemed important. The output boundary of the sigmoid function is between 0 and 1, this will be the weights to the information to decide which data is to be removed or kept in the cell state.

The input gate updates the cell state by the tanh and sigmoid activation functions based on the previous hidden state and the current timestep. Similarly, it first passes the data into the sigmoid function to decide which information is deemed important then the input data is passed through the tanh function to regulate the data by formatting it into a boundary between -1 and 1. Once multiplied together, this decides which output from the tanh function to keep or remove.

With the outputs of the forget gate and the input gate, the cell state can be calculated. By first multiplying the sigmoid outputs from the forget gate to the previous cell state, it 'forgets' or decrease the weight of importance of the data in the previous cell state. Then, it updates cell state by applying an addition operation with the outputs of the input gate with the new values that it finds relevant.

The output gate calculates what the next hidden state should be for the next timestep, the hidden state is used for predictions and provides information on the previous inputs. It first applies the sigmoid function to the previous hidden state and current timestep to which is similar to how the forget gate calculates information importance then multiplies the output of applying the tanh activation function to the new cell state. This output will be continued to the next timestep.

3.2 Reinforcement Learning

Reinforcement Learning, learning what to do without being explicitly being told what to do by mapping states to a defined actions under a numerical reward system. No doubt this has been an interesting area in research and provided us with steppingstones where machine intelligence has far exceeded human intelligence in numerous instances. Google Deepmind provided us with one of these steppingstones when their agents consistently defeated world champions in their own games, some of the games include Go and more recently, StarCraft II. In this project, an attempt has been made to simulate the same effects with a variation of the same algorithm, Deep Q-Learning, under the application of automating trade in the financial markets.

3.2.1 Markov Decision Process

The basis of reinforcement learning stems from Markov Decision Processes. A process is said to be Markovian if a given state is said to be probabilistically determinable from the previous state(s).

An agent is the learner of the system which in this case is the trading bot, and it has a perception of a closed-form world to which we call the environment. The aim of the agent is to maximise its rewards which is defined by the results of executing any given action at the current environment, this is similarly thought to be like a scoring system where the goal is to get the highest possible score. At any timestep t, the state, action, and rewards are denoted as S_t, A_t, R_t respectively and the sequence will begin like below. The action is determined after observing its environment and the reward is calculated after the action is executed to which then gives us the next state and the process is then repeated.

$$S_0, A_0, R_1, S_1, A_1, R_1, S_2, \dots$$

In a Finite Markov Decision Process, the model defines a probability distribution for each state S executing action A that will lead to state S'. The idea of reinforcement learning is not to maximise immediate rewards, but aims to maximise the cumulation of all future rewards [28].

3.2.2 Continual and Episodic Tasks

There are a couple of ways to represent a sequence of rewards. We can represent this as a never-ending sequence of rewards called continuing tasks, where there exist no limitations and the agent is continually learning. The other method would be Episodic tasks where the sequence is broken into subsequences called episodes. Each episode has a terminal state which is the last state of the episode followed by a reset which converts to a standard initial state. Dependant on the application, one can be favourable than the other. Episodic tasks can be useful for game-play where each episode is a single game and game strategies is improved in each episode, and continuous tasks are useful when there exists no terminal state such as a personal assistance bot.

In this application, we will be implementing episodic tasks for automating trade. This can also be implemented as a continuous task where our goal is to continually maximise the agent's portfolio, but I believe by limiting the timesteps and implementing episodes will encourage the agent to trade more frequently and profitably in a shorter amount of time. This method will enable the agent to focus more on recent rewards and disregard any actions that will have little impact due to the accumulated weight of the discount factor. Below displays the equation for expected returns, it is calculated by the accumulation of rewards in each episode multiplied by the discount factor, γ, which determines the present value of future rewards [28].

$$G_t := \sum_{k=t+1}^{T} \gamma^{k-t-1} R_k$$

$$\gamma \in [0, 1]$$

3.2.3 State-value function and Action-value function

In every reinforcement learning algorithm, there exists a state-value function or an action-value function which dictates how much the agent values a certain state or action at any given occurrence. A policy is thought of as a plan that maps each state to an action enabling the agent to decide in which action to execute in a specific state. If an agent is following policy π at time t, we can say that $\pi(a|s)$ is the probability of the agent executing action $a \in A$ given state $s \in S$ where A and S is the set of all actions and states respectively. The notation is the same as conditional probability. $v_\pi(s)$ denotes the state-value function, the equation below displays that the value of a given state following a policy π is equivalent to the expected cumulative returns (future rewards) given the state s. Similarly for the action-value function, $q_\pi(s, a)$, the expected cumulative returns is equivalent to the function requiring state s and action a.

$$v_\pi(s) := \mathbb{E}[G_t | S_t = s] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s\Big],\, for\, all\, s \in S$$

$$q_\pi(s, a) := \mathbb{E}[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s, A_t = a\Big],\, for\, all\, s \in S, a \in A$$

After defining what is a policy, we aim to optimise our reinforcement learning algorithm to find the optimal policy that will provide us with the best action to execute at any given state. The optimal policy is defined as a policy that is greater or equal to any other policy. Intuitively, this is thought as a plan that will provide the highest score out of the possible plans that can be created. The optimal state-value and action-value functions are denoted below.

$$v_*(s) := \underset{\pi}{max}\, v_\pi(s)$$

$$q_*(s,a) := \underset{\pi}{max}\, q_\pi(s,a)$$

3.2.4 Q-Learning

There any many different reinforcement learning algorithms that aims to find the optimal policy, such as policy gradient and SARSA, but the algorithm that will be implemented in this project is an off-policy temporal difference control algorithm called Q-Learning. Q-Learning is an off-policy algorithm, because it does not update a policy, but directly approximates q_* using the Bellman's equation by looking at future successor states [28]. The purpose of utilising the Bellman's equation in this way is so that it is able to converge to an optimal policy by minimising the margin between the expected and the current q-value of the state-action pair. Below displays the update for Q-Learning utilising the Bellman's equation and how it resides within the algorithm.

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \underset{a'}{max}\, Q(s_{t+1},a') - Q(s_t,a_t)]$$

Algorithm 1 Q-Learning [28]

```
1:  Initialise $Q(s,a)\ \forall s \in S, a \in A,$ and $Q(\text{terminal-state}, \cdot) = 0$
2:  for each episode do
3:      Initialise state, $s$
4:      for each step in episode do
5:          Choose $a$ from $s$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
6:          Take action $a$, observe $r, s'$
7:          $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma\, max_{a'}\, Q(s',a') - Q(s,a)]$
8:          $s \leftarrow s'$
9:      end for
10: end for
```

3.2.5 Exploration vs Exploitation

During the initial phases of learning in a reinforcement model, a sufficient amount of experience is required before making a good judgement of q-values from the state-action pairs. Often, the model makes random decisions to gather experience before determining the quality of an action given a state; this is called exploration. Exploitation is when the agent has sufficiently acquired enough experience and it is able to optimise on actions from similar experiences or from what it has already encountered.

The design of a system to balance the exploration and exploitation in the model is based on the creator. High exploration can lead to finding the optimal solution faster, but may require longer time to exploit and therefore, slower converge to the optimal policy. Too small of an exploration can lead to an early convergence to a local minima and may have difficulty in finding the optimal policy.

The design that will be implemented will be the ϵ-greedy algorithm where exploration will always have a chance of occurring, but will gradually decrease to a defined minimum. This is based on a parameter epsilon where it is initially 1 and denotes the likelihood of exploration. At every timestep, this is multiplied by an epsilon decay so that the chance of exploration is decreased and exploitation is increased simultaneously until a defined minimum is reached.

Algorithm 2 ϵ-greedy algorithm

```
1: Initialise $epsilon\_decay \in [0,1]$, $min\_epsilon \in [0,1]$, $\epsilon = 1$
2: procedure EPSILON_GREEDY($\epsilon$)
3:     if epsilon_decay$*\epsilon >$ min_epsilon then
4:         $\epsilon \leftarrow epsilon\_decay * \epsilon$
5:     else
6:         $\epsilon \leftarrow min\_epsilon$
7:     end if
8:     $rand =$ random(0,1)
9:     if $rand < \epsilon$ then
10:        return random $a \in A$ (exploration)
11:    else
12:        return best $a \in A$ from $S$ using policy derived from $Q$ (exploitation)
13:    end if
14: end procedure
```

3.2.6 Deep Q-Learning

The problem with Q-Learning is that it is an algorithm built for discrete states which then creates a simple dynamic programming implementation to estimate q-values for each state-action pair. In the real world, not all states can be represented discretely, but rather continuously. In the stock trading domain, the states are the closing prices so the environment in this domain is continuous. The general Q-Learning algorithm cannot be implemented and therefore, dynamic programming is not applicable. Another method would be to use Neural Networks as function approximators so that we can feed continuous and discrete data as inputs to estimate q-values for any given quantitative state.

3.2.7 Experience Replay

Reinforcement Learning is known to diverge or become unstable when a Neural Network is implemented as a function approximator, this is due to the high correlation between observations when fed with only the recent timestep for learning [30]. This will lead the agent having a skewed observations in learning and unable to make better decisions and generalise well for observations it has not encountered for a while.

A method to overcome this would be Experience Replay, this saves all the past experiences into a queue called Replay Memory and randomly selects a batch of these experiences for learning. An experience at timestep t, denoted as e_t, is a tuple of its current state, the action executed, the reward gained from that action, and the next state after executing the given action. This then removes the correlation of encountering recent observations by feeding random past experiences into the Neural Network which enables it to generalise better for seen and similar experiences [9]. The size of the Replay Memory will need to be chosen as saving all experiences may not be useful and can be memory intensive.

$$e_t = (s_t, a_t, r_t, s_{t+1})$$

3.2.8 Target Network

Q-Learning updates its state-action pairs by calculating the q-values of the current state-action pair and the maximum q-value from the next state-action pair, also known as the target value. By using Neural Networks as function approximators, this can be seen as propagating through the network twice for two different outputs, where intuitively, the current q value is attempting to approximate the target value. This can be problematic as we update the current q-values, because the target values will also be updated in the same direction and will lead to a never-ending optimisation problem and therefore, contain instability. This is solved by introducing a Target Network where only the target values are obtained from this network and is reset to the original Neural Network after a defined amount of timesteps. Along with the size of the Replay Memory, the amount of timesteps required to reset the target network weights to the original network also needs to be chosen.

Algorithm 3 Deep Q-Network algorithm [9, 11]

```
1: Initialise replay memory $D$ to capacity $N$
2: Initialise network $Q$ and target network $\hat{Q}$ with random weights
3: Initialise $min\_experience$, $max\_counter$, $counter = 0$
4: for each episode do
5:     Initialise state, $s$
6:     for each step $t$ in episode do
7:         Choose $a$ from $s$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
8:         Take action $a$, observe $r, s'$
9:         Store transition $(s_t, a_t, r_t, s'_t)$ in $D$
10:        if length of $D > min\_experience$ then
11:            Sample random minibatch of transitions $(s_j, a_j, r_j, s'_j)$ from $D$
12:            for each transition in minibatch do
13:                $counter \leftarrow counter + 1$
14:                if $s'_j$ is terminal then
15:                    $y_j = r_j$
16:                else
17:                    $y_j = r_j + \gamma \, max_{a'} \hat{Q}(s'_j, a')$
18:                end if
19:                Calculate mean squared error loss $\mathcal{L} = \frac{1}{N}\sum_{i=0}^{N-1}(Q(s_j, a_j) - y_j)^2$
20:                Update $Q$ by performing gradient descent and minimizing the loss $\mathcal{L}$
21:                if $counter > max\_counter$ then
22:                    $\hat{Q} \leftarrow Q$
23:                    $counter \leftarrow 0$
24:                end if
25:            end for
26:        end if
27:    end for
28: end for
```

Chapter 4

System Design

4.1 Agent and Environment

In reinforcement learning, the agent is the entity that performs actions and the environment is what the agent perceives its world to be. In this domain, the agent is the entity that decides and executes the trades, and the environment is the price of the market and the boundary information of the system. This boundary information includes the agent's bank and size of inventory.

At the start of each episode, the environment will reset itself to a standard initial state so that the agent is able to improve its strategy under the same initial conditions. The initial state of the episode includes resetting the agent's bank to the initial value of €200 and initialising a queue that will represent an empty inventory. How trades operate in this design is that when the agent wishes to buy a unit of currency, the price of the exchange will be appended to the inventory and the price will subtracted from the bank. Similarly, when an agent wishes to sell a unit of currency, the earliest unit of currency to be added to the inventory will be removed and the difference between the inventory price and the current price will be added onto the bank.

There exist some flaws within this design, if the agent intends to buy more than its bank size or sell more units than there exists in its inventory, this would not be executable. Experiments have taken place to assess what design would be most suitable in dealing with non-executable actions.

4.1.1 Function Approximator Design

A base model was originally created to evaluate different approaches in deciding a design for non-executable actions and selecting parameters for the reinforcement learning design. Figure 4.1 displays the concatenated Artificial Neural Network that will be used as a function approximator for these steps. It is designed this way to ensure 2 inputs of different dimensionality can be fed into a single function approximator to produce a single 1-dimensional output. The function approximator has two inputs for the sequential price inputs named 'price_input' and for the boundary information named 'env_input'. The 2-dimensional price input will be propagated into 2 hidden layers then flattened to form a one-dimensional layer before it is concatenated with the boundary information. This is then followed with further 2 more hidden layers before the output layer is reached.

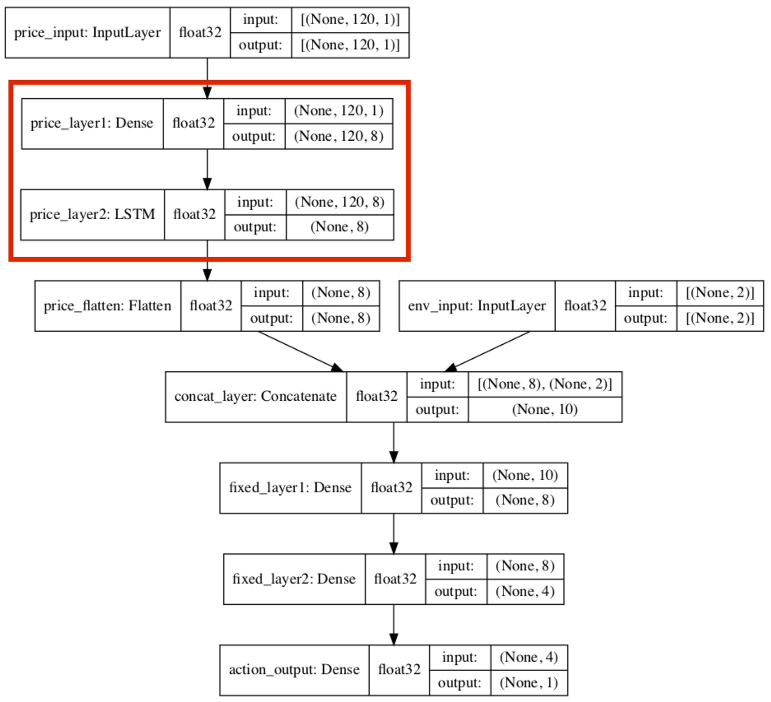

price_input: InputLayer | float32 | input: [(None, 120, 1)] | output: [(None, 120, 1)]
price_layer1: Dense | float32 | input: (None, 120, 1) | output: (None, 120, 8)
price_layer2: LSTM | float32 | input: (None, 120, 8) | output: (None, 8)
price_flatten: Flatten | float32 | input: (None, 8) | output: (None, 8)
env_input: InputLayer | float32 | input: [(None, 2)] | output: [(None, 2)]
concat_layer: Concatenate | float32 | input: [(None, 8), (None, 2)] | output: (None, 10)
fixed_layer1: Dense | float32 | input: (None, 10) | output: (None, 8)
fixed_layer2: Dense | float32 | input: (None, 8) | output: (None, 4)
action_output: Dense | float32 | input: (None, 4) | output: (None, 1)

Figure 4.1: Function approximator base model.

In choosing the hyperparameters for the different function approximators, only the layers within the red box will be modified. This is to keep a fixed design for the rest of the function approximator and purely focusing on the types of layers within the red box that will influence the overall performance of the model. The types of layers that will be evaluated in the red box will be hidden layers, convolutional layers, and LSTM layers to assess how ANN, CNN, and RNNs perform as function approximators.

4.2 Methodology

The chosen foreign exchange datasets are EUR-GBP, EUR-USD, and EUR-CHF currency exchanges. The currency exchanges are conversions from one currency to another where EUR is Euros, GBP is Pound Sterling, USD is United State Dollar, and CHF is Swiss Franc. These datasets are 4-months in size with 1-second intervals. In evaluating the performance of a model, the data has been split into training, validation, and test sets with the ratio of 3:1:4. These datasets are displayed in Figure 4.2, where the blue line represent the training set, orange line for the validation set, and the green line for the test set. Models are trained with the training set then to be compared with other models with the validation set. Only the test set will be used to evaluate the final performance of the model with unseen data after the model has been selected in the validation phase. These three datasets were specifically chosen due to their dissimilar underlying trends in the 3 phases of learning, this will provide a more diverse evaluation under different environments. In Figure 4.2, EUR-USD has an increasing trend pattern for training whereas for EUR-CHF has a decreasing trend. Both of their test sets have high volatility with opportunities to make large gains or losses, so it would be an interesting evaluation to assess how the agents will optimise in these opportunities trained on contrasting price patterns. The training set of EUR-GBP has more of a consolidated trend which may provide less of a bias with unseen data, so this dataset was chosen for the parameter selection phase of the project.

Due to the large selection of parameters to be selected, evaluating all combinations of models by brute-force is not practical due to the extensive training time required. A solution to this would be to choose parameters consecutively where a large range of a single parameter will be assessed to which then the optimal parameter will be continued for further parameter selection. Although this will reduce the number of combinations to evaluated, this can also lead to finding a local optimum and will not guarantee in finding the optimal model for this application.

The design and parameter selection will be based on the EUR-GBP dataset. After finding the optimal model for the different function approximators, the same optimal models will be trained and tested onto the EUR-USD and EUR-CHF datasets to assess its performance on other foreign exchange markets.

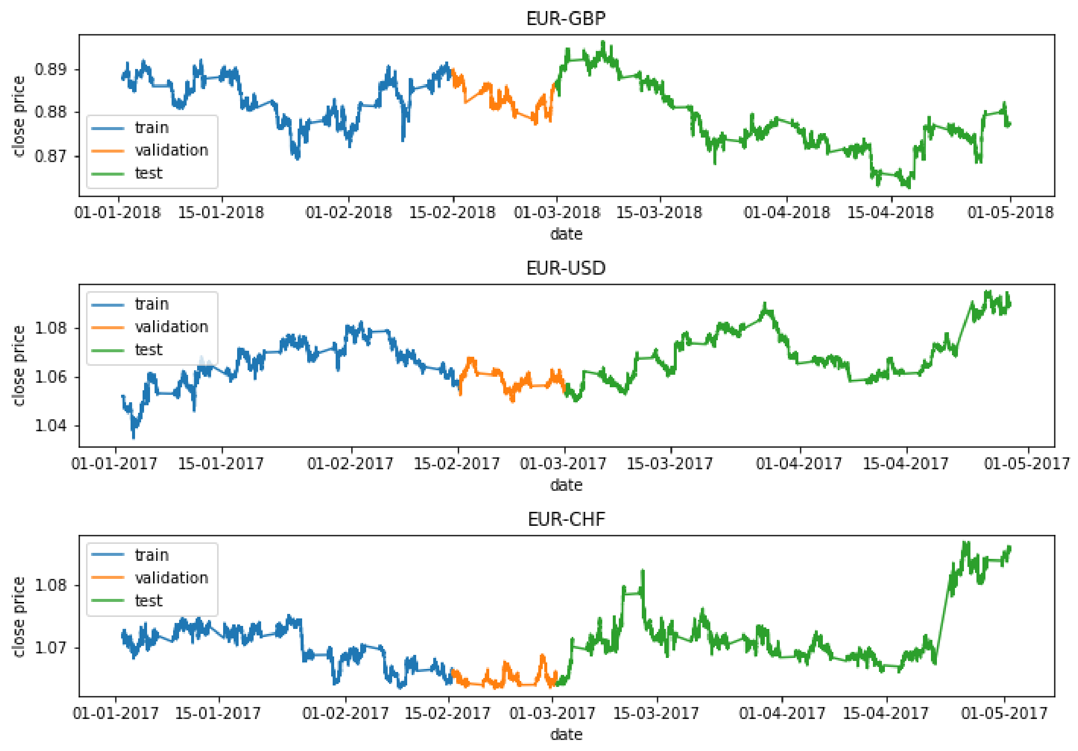


Figure 4.2: Train, validation, and test split of the 3 datasets: EUR-GBP, EUR-USD, EUR-CHF.

4.2.1 Exploration vs. Exploitation

When an action has been decided, it may not be executable due to exceeding the agent's bank when a buying too many units or exceeding the inventory when selling too many units. Two methods have been designed to deal with these non-executable actions. The first method is to not execute the given action, but still save the transition into experience replay with a reward of 0. The intuition behind this is to inform the agent that no benefit can be gained from this action, but can be misleading as it provides learning examples of failed actions and may be difficult for the agent to differentiate to what is considered as a 'good' or 'bad' action. The second design is to force exploration when an action is not executable. This will guarantee that the action is feasible as exploration will always be within the boundary limits of the agent and provide the agent with more informative examples in the experience replay.

Figure 4.3 displays the performance of the model that forced exploration (blue line) and did not forced exploration (orange line). With the explore design performing the best out of the two, this design will be continued with the next stage of the design process. With clear observation, forcing exploration when an action was deemed non-executable was proven to be consistently better than saving the transition to the replay memory with a reward of 0. This is due to the agent provided with better experiences to learn from and therefore, enabled it to execute better and more educated actions.

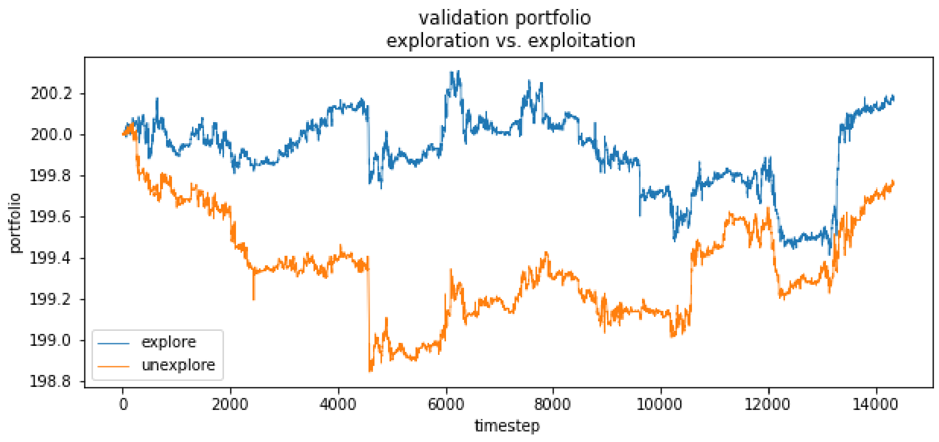


Figure 4.3: Portfolio values of two methods implemented for non-executable actions.

4.2.2 Reward Systems

Another two designs have been created to assess which reward system performs best in this domain. An agent has 3 decisions to make, to either: buy, hold, or sell. When an agent decides to hold, it does not gain or lose anything from this action, so it has a reward of 0. When an agent buys a unit of currency, it loses bank but gains it back as inventory to neutralise the action, so it also has a reward of 0. Different from the other two actions, when an agent sells a unit of currency, it can either make a profit or a loss dependant on the price of the current unit and the unit removed from the inventory. The reward system will be entirely based from the profit or loss of the sell action. To have the reward as the true gain or loss, it can provide a realistic impact on the agent's portfolio and furthermore, penalise for unprofitable actions. This reward design is displayed as the blue line in Figure 4.4 labelled as 'face-value reward'. Another design would be to only have positive rewards where the rewards are only the gains and losses will have a reward of 0, this is shown as the orange line. A negative reward may discourage the agent in trading so setting negative rewards to 0 may encourage the agent to trade more frequently.

Both of these designs ended with similar final portfolio value, but the overall performance of having reward as true gain or loss performed better overall. The true gain/loss reward system and forced exploration design are taken for the further experimentation.

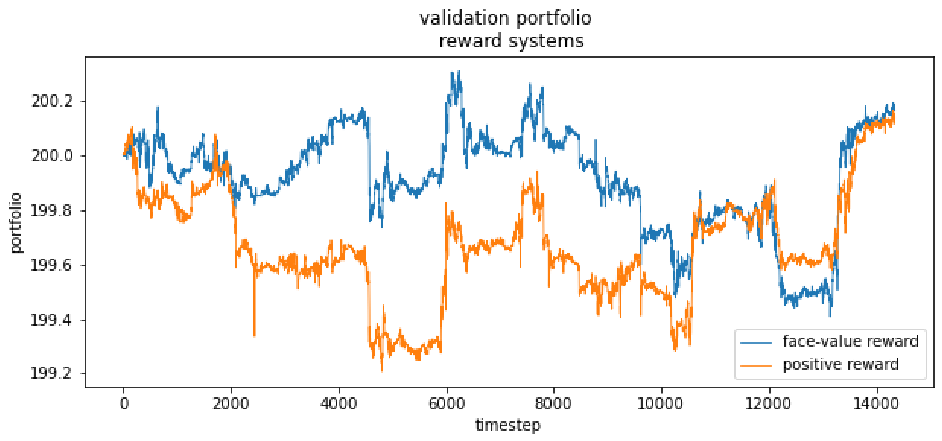


Figure 4.4: Portfolio values of the two reward systems.

4.2.3 Q-Learning Parameters

Q-Learning contains many parameters to be selected where often all combinations of a small range is tested by brute-force and the best performing is chosen as the optimal combination of parameters. Some of these parameters have been fixed from the beginning to reduce complexity in parameter selection, but the parameters that will most likely have a large impact on the performance of the model will be evaluated. These Q-Learning parameters are epsilon decay and the size of the replay memory.

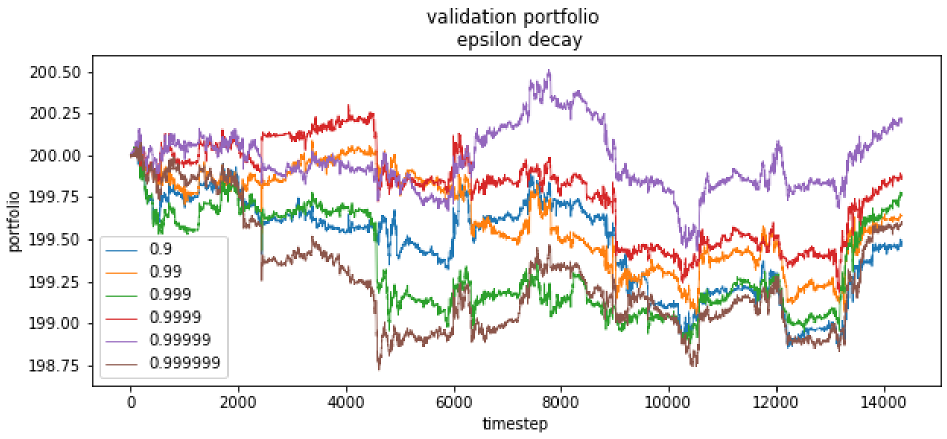
The reason why epsilon decay was chosen for evaluation is because this decides how much exploration to conduct onto the data before large focus is conducted onto exploitation. If epsilon decay is too low, the agent may not conduct sufficient exploration and can begin to focus on exploitation too early and thus, optimise in a local optimum. Conversely if epsilon decay is too high, an agent will explore too much and may not be able to correctly estimate the q-values of its states as it was unable to exploit those actions early on. This will also increase the overall learning time before it reaches to a global optima.

As mentioned, replay memory is stored as queue data structure where a unit of memory is appended and removed at every timestep. In experience replay, a batch is randomly chosen from this memory to train the function approximator. The importance of size is that you can define how many previous timesteps you wish to keep in memory where a larger memory size will include memory further in the past. With the goal of executing better actions, the more the agent is trained the better the quality of the memories. It is indefinite that more recent memories will be of better quality, but too large of a memory size can lead to variability in memory quality and too small can lead to observing a too similar market trends and thus, less variability between memories.

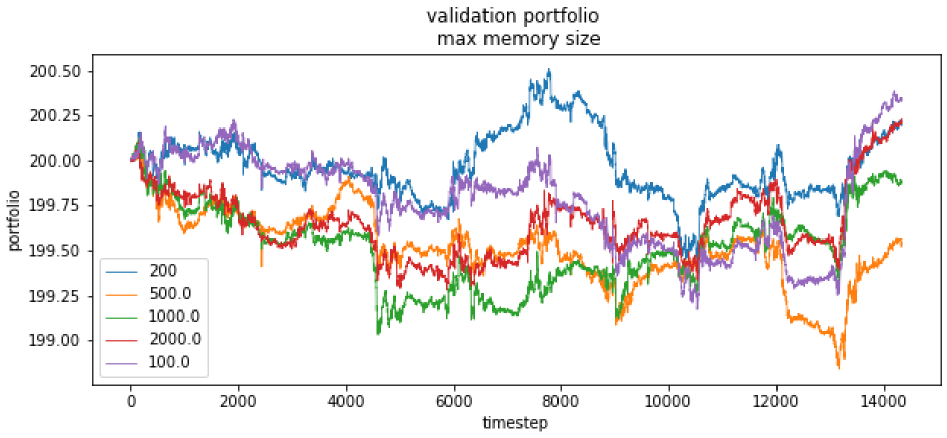
Another parameter that was evaluated was the window size or the length of the price input. It is difficult to say how many previous timesteps is required to sufficiently predict the movement of the given market, so this has also been experimented to evaluate the optimal window size.

With some of the figures shown, it is difficult to assess by speculation which is the optimal parameter due to the clustered portfolio values. A metric of average portfolio was implemented to decide the optimal parameters which lead to an epsilon decay of 0.99999, replay memory size of 200, and a window size of 5. This then implies that the agent only requires recent experiences to evaluate the optimal actions and it is able to effectively estimate q-values by only repeatedly experiencing the memory for a short duration. Epsilon decay is expectedly high due to the large sequential states there are within the domain leading to a high demand of exploration before it is able to estimate q-values effectively.

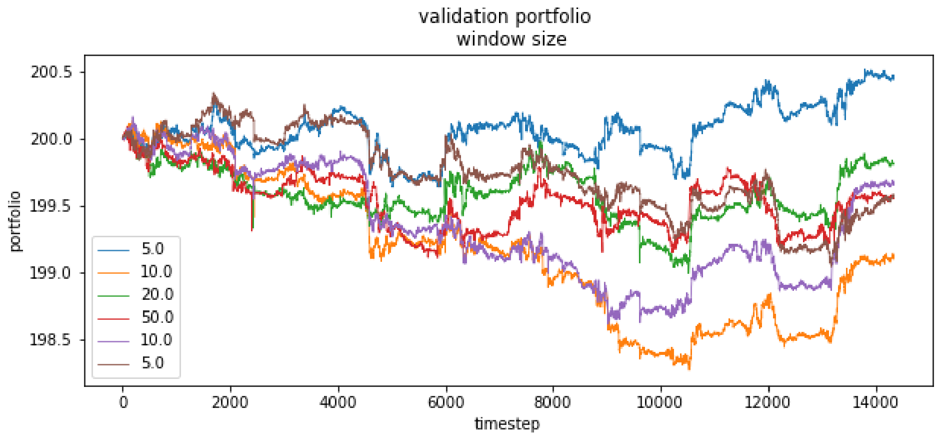
With the results seen so far, the model is able to become profitable as in all three instances as it was able to finalise in a profitable portfolio value, although, it does have its dips below the initial bank of €200 and appears unreliable due to its erratic variability in portfolio. This could be deemed as a positive milestone for the agent, because as shown in Figure 4.2, the validation phase of the EUR-GBP dataset is a falling market so the agent is already at a disadvantage at this stage where shorting is unavailable, but it was able to finalise in a profitable portfolio.



(a) Portfolio values for the different epsilon decay values.



(b) Portfolio values for the different replay memory sizes.



(c) Portfolio values for different window sizes.

Figure 4.5: Portfolios for different experiments.

Chapter 5

Experimental Results

5.1 ANN Function Approximator

Of the three variations of the function approximator, ANN was first assessed. The experiment contained a maximum of 2 hidden layers of either 8 or 32 nodes, and an optimizer of stochastic gradient descent or Adam. The method of assessing the optimal model was by grid search where all hyper-parameter combinations within the defined ranges are exhaustively trained and the validation performances are assessed afterwards. The best performing model was a 2 hidden layer function approximator with both nodes of 32 and an optimizer of Adam. For the validation phase, this model was able to become profitable with a final portfolio of €201.09 leading to an average monthly return of investment (ROI) of 0.54%.

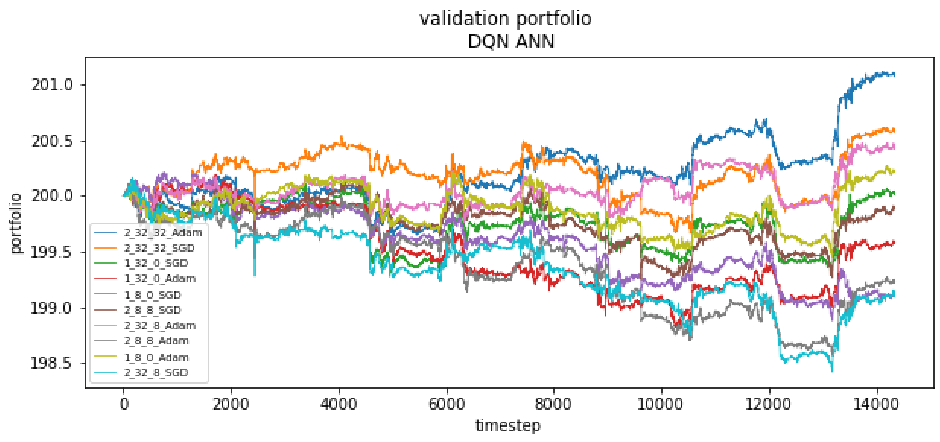


Figure 5.1: Portfolio values for DQN_ANN models.

5.2 CNN Function Approximator

The same grid search method was used in assessing the function approximator with convolutional layers. The best performing model was with 2 convolutional layer of 32 filters and an optimizer of Adam. This had a final portfolio value of €200.97 leading to a monthly ROI of 0.49% which is lower than the DQN_ANN model.

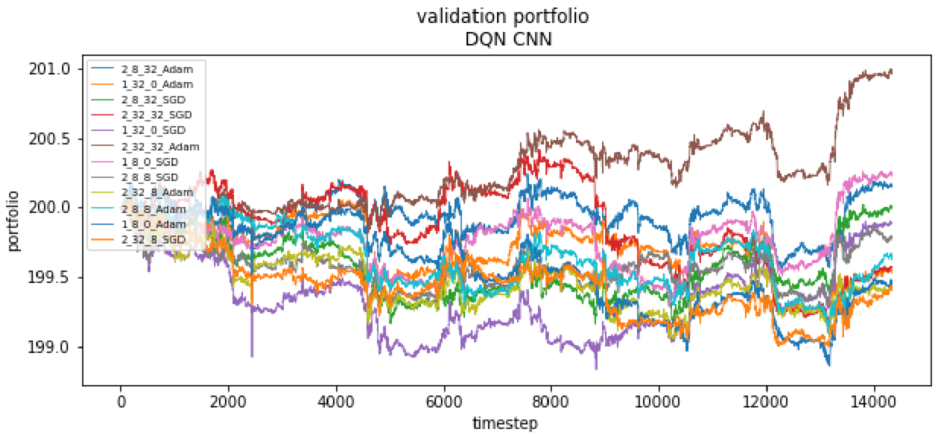


Figure 5.2: Portfolio values for DQN_CNN models.

5.3 RNN Function Approximator

The method was again repeated for the function approximator with LSTM layers. The results are more clustered and less obvious in this experiment and thus, the optimal model was harder to differentiate. An average portfolio per timestep was calculated with the highest being the 2-layered LSTM with nodes of 32 and 8 respectively, and an SGD optimizer. All averages were below the initial bank meaning that none of these models were profitable on average, although, some were able to achieve a final profitable portfolio value. This model acquired a final portfolio value of €200.35 and a monthly ROI of €0.17%.

5.4 All Models

After selecting the hyperparameters in the validation phase for the three variations of DQN, the models were re-trained and tested onto the 3 datasets: EUR-GBP, EUR-CHF and EUR-USD. Along with evaluating these models onto new unseen data, the trading model was compared against two popular trading strategies: Mean Reversion and Moving Average Trading Strategy. Unlike the DQN trading model, these two trading strategies enabled the trader to know

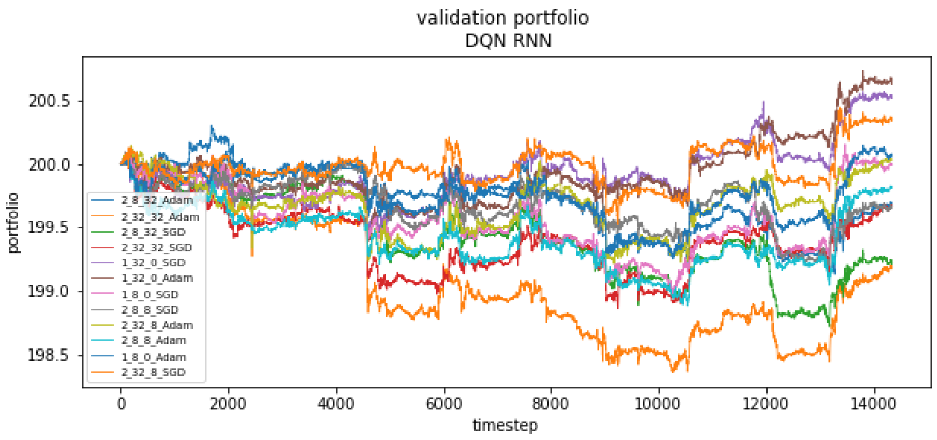


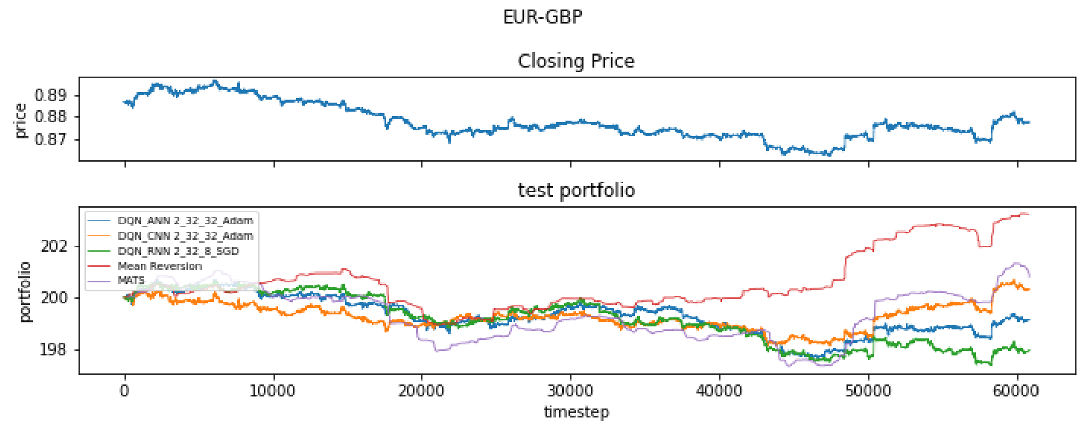
Figure 5.3: Portfolio values for DQN_RNN models.

whether to trade at a particular price pattern, but does not provide a trade quantity to match the level of confidence of the trade. Therefore, every trade executed will be traded with 50 units chosen arbitrarily.

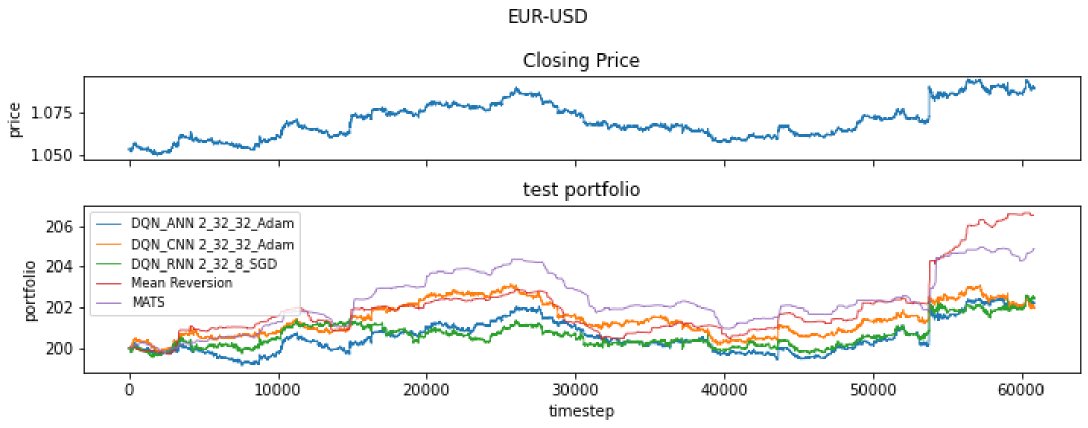
Mean Reversion is based on fluctuations around an underlying curve. The idea is that the movement of the prices follow a long-term trend such that whenever there exists a spike, it will eventually revert to the other direction of the curve so that the variance around the underlying curve is consistent. The underlying curve is calculated with a moving average of 50 timesteps and indications of significant spikes are detected using Bollinger Bands and RSI.

Moving Average Trading Strategy consists of 3 moving average curves with periods of 10, 20, and 50. These 3 curves represent the short-term, mid-term, and long-term movement of the market. Outbreaks are detected when these moving averages crossover from one another and the curves are ordered in a specific way.

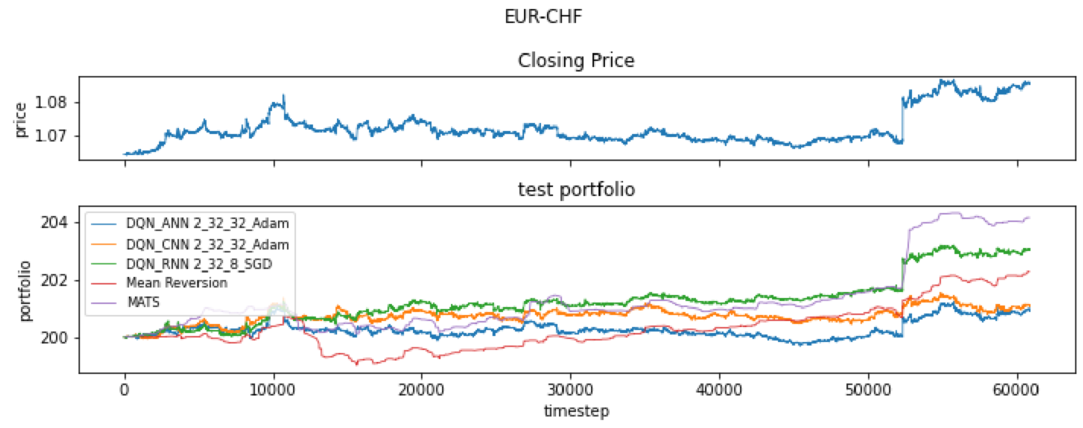
With the graphs shown in Figure 5.4, all DQN models were able to become profitable in almost every foreign exchange market as they were able to achieve a monthly ROI between 0.55% - 1.74%. Some models were unprofitable in the EUR-GBP dataset which may be due to the falling nature of the market. Between the three different Neural Networks as function approximators, there was no clear model that out-performed the other two models. The portfolio values between the models were very clustered together and only displayed variability in performance towards the end of the testing phase. Although the datasets were chosen specifically due to their dissimilar underlying trends to test how the models would behave in different



(a) Model portfolio for EUR-GBP

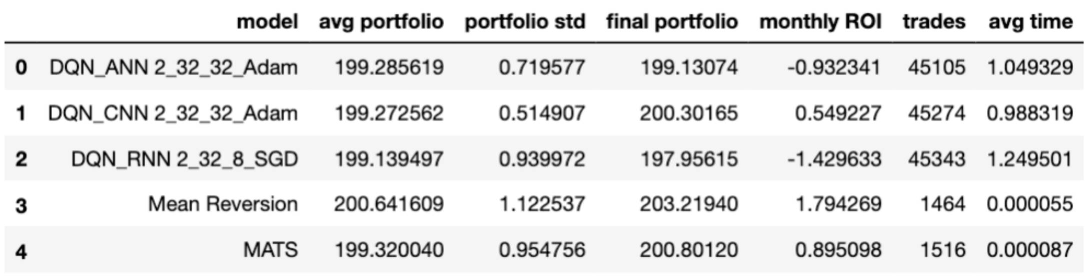


(b) Model portfolio for EUR-USD



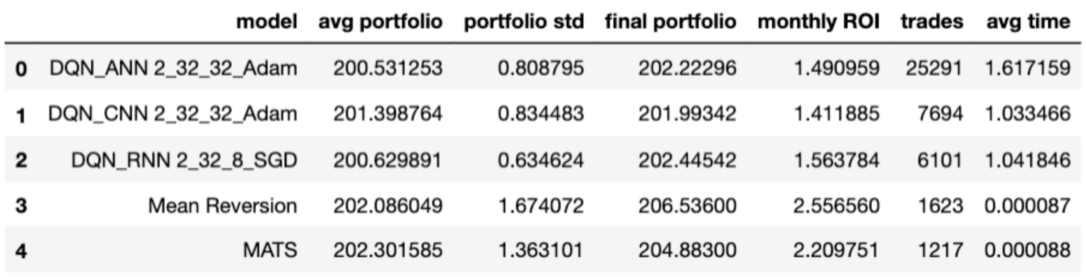
(c) Model portfolio for EUR-CHF

Figure 5.4: Model statistics for all test datasets.



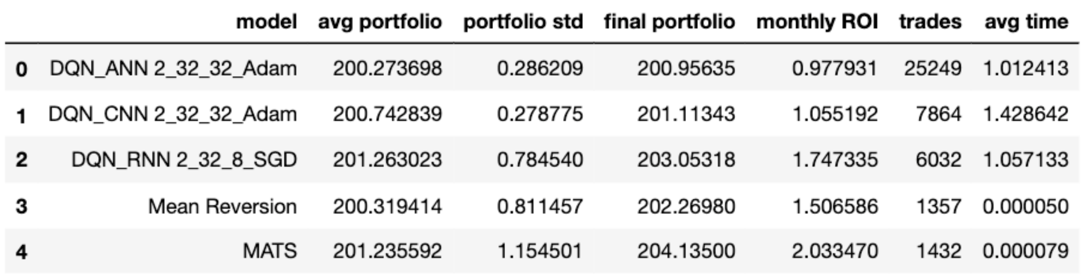
	model	avg portfolio	portfolio std	final portfolio	monthly ROI	trades	avg time
0	DQN_ANN 2_32_32_Adam	199.285619	0.719577	199.13074	-0.932341	45105	1.049329
1	DQN_CNN 2_32_32_Adam	199.272562	0.514907	200.30165	0.549227	45274	0.988319
2	DQN_RNN 2_32_8_SGD	199.139497	0.939972	197.95615	-1.429633	45343	1.249501
3	Mean Reversion	200.641609	1.122537	203.21940	1.794269	1464	0.000055
4	MATS	199.320040	0.954756	200.80120	0.895098	1516	0.000087

(a) Model statistics for EUR-GBP



	model	avg portfolio	portfolio std	final portfolio	monthly ROI	trades	avg time
0	DQN_ANN 2_32_32_Adam	200.531253	0.808795	202.22296	1.490959	25291	1.617159
1	DQN_CNN 2_32_32_Adam	201.398764	0.834483	201.99342	1.411885	7694	1.033466
2	DQN_RNN 2_32_8_SGD	200.629891	0.634624	202.44542	1.563784	6101	1.041846
3	Mean Reversion	202.086049	1.674072	206.53600	2.556560	1623	0.000087
4	MATS	202.301585	1.363101	204.88300	2.209751	1217	0.000088

(b) Model statistics for EUR-USD



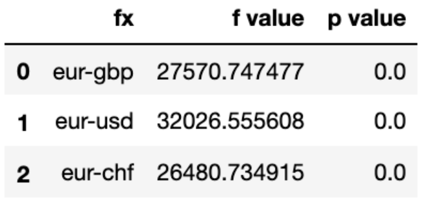
	model	avg portfolio	portfolio std	final portfolio	monthly ROI	trades	avg time
0	DQN_ANN 2_32_32_Adam	200.273698	0.286209	200.95635	0.977931	25249	1.012413
1	DQN_CNN 2_32_32_Adam	200.742839	0.278775	201.11343	1.055192	7864	1.428642
2	DQN_RNN 2_32_8_SGD	201.263023	0.784540	203.05318	1.747335	6032	1.057133
3	Mean Reversion	200.319414	0.811457	202.26980	1.506586	1357	0.000050
4	MATS	201.235592	1.154501	204.13500	2.033470	1432	0.000079

(c) Model statistics for EUR-CHF

Figure 5.5: Model statistics for all test datasets.

environments, the component that became more noticeable was the small positive outbreak during the last stage of the test set. The main differentiator between the trading strategies and the DQN models was how they were able to optimise on positive outbreaks as when the closing prices were generally decreasing or in consolidation; all strategies were unable to make large profits until the end of the test phase. This should be expected because shorting was unavailable to all strategies, but if enabled, different results may have been achieved.

In comparison to the two popular trading strategies, they were able to outperform the DQN models almost entirely. When a positive trend was emerging, the popular trading strategies were able to take advantage of these moments and optimise the portfolio. The DQN models were able to notice these opportunities, but were slow in committing to larger trades in generating greater profits. To evaluate this more thoroughly, a one-way ANOVA test was conducted to assess how statistically different the mean of the portfolios differed from one another. An ANOVA test evaluates whether statistically there exists a relationship between two or more groups and tests the idea that some strategies were able to out-perform another strategy. As shown in Figure 5.6, the results of the ANOVA test contains significantly high f-values and significantly small p-values for each currency exchange. This provides us with evidence that we can reject the hypothesis of all the strategies having the same mean and conclude that at least one strategy differs from another in each currency exchange.



	fx	f value	p value
0	eur-gbp	27570.747477	0.0
1	eur-usd	32026.555608	0.0
2	eur-chf	26480.734915	0.0

Figure 5.6: ANOVA statistics for each currency exchange.

The length of time in deciding an action for the DQN models was about 1.17 seconds, this duration would be too long for this application as a necessary constraint in trading in 1-second intervals is to decide actions much faster than 1 second to quickly process data and to execute actions immediately. A more suitable timeframe for this model would be duration greater than 10 seconds, this makes room for adjustments such as increasing the overall complexity of the model which will likely increase runtime in deciding an action, but may lead to more positive results. Not only the average time was greater than the popular trading strategies, the number of actions executed were also greater. This then shows that the popular trading strategies were able to make better trades utilising less time and actions implying that the popular strategies were able to generally execute better trades.

Chapter 6

Conclusion

6.1 Conclusion

The goal of this project was to create a trading agent using Deep Q-Learning to not only automate what is the optimal trade execution given a state, but to quantify the trade by having a linear activation function in the final layer of the function approximator. The final results proved that this was profitable for most markets, but in comparison to the popular trading strategies, this failed to beat simple existing algorithms commonly used by traders. This may be due to the complexity of the function approximator where concatenating multiple inputs and deciding a linear output can over-complicate the simple and prime job of the function approximator, which is to predict the q-values of the state-action pairs. Not only did it fail to perform better than these popular strategies, but the required time for computing was also far too long for practical use for the second-interval domain. If this were to be implemented in a real forex trading domain, the popular trading strategies would be far more advantageous with the simple calculations to provide quick and immediate trading decisions along with generating greater profits.

An important factor to note is that transactions fees were not included in the final development of the model. This was tested during the phase of experimenting with different reward systems, but any transactions fees included had a negative impact on the learning of the model. The reason for this would stem from each buy action having an immediate loss and become a negative incentive in buy trades. This would then discourage the agent in buying units and furthermore, discourage overall trade. To analyse the existing results without transactional fees, the trading agents conducted a significant amount of trades, but the profits in the final portfolio would not have compensated for the total transactional fees in a real-life application and would have resulted in an overall loss.

6.2 Future Works

Within the domain of trading in 1-second intervals, the architecture of the function approximator could be simplified to provide faster decision making. A large number of layers and nodes sizes can lead to a longer training time, but the structure of having multiple inputs and a concatenated layer may also have a large impact on execution time. With proven results with a tanh final activation function [20, 32], an interesting assessment could be to compare the performance of the two final activation functions where one provides the action and quantity of trade, and the other provides only which action should be executed.

If the domain were to be changed to a larger trading interval, a more complex or deeper model requiring longer training time may be more suitable for this domain. In evaluating parameters for the function approximators, a more thorough evaluation could be performed by experimenting with all combinations rather than choosing parameters after another, along with having a larger sample size to optimise the model further in hope of converging to the global optima. When selecting the best model in each DQN variation, the best performing models in the validation phase were usually the models that had the maximum amount of hidden layers and node sizes in the specified range. This could potentially mean that the global optima has not been achieved, but limited to the range of the parameters available. Additional layers could also be included such as pooling layers in the CNN function approximator or including regularisation, such as dropout, to minimise the effect of overestimation.

Feature engineering could also be an interesting area to build on from this model. As proven with the popular trading strategies, they both use technical indicators that provides a more informative observation of the trends in the data. If technical indicators were fed instead or alongside the raw closing price data, this may provide the model in a better judgement of the market.

6.3 Project Evaluation

There are many adjustments and considerations I would have factored if I were to restart this project. I started this project with an immediate bias that the architecture of the base model was a suitable template in forming the function approximators. A simpler base model with one or no hidden layers after the concatenated layer may have been more suitable to reduce the runtime for training these models and provide more insight whether additional hidden layers should be considered.

The entire project was run on a MacBook Pro with 2 GHz Quad-Core Intel Core i5 processor. Training each model took around 13 hours where a whole experiment took 4-5 days. A more suitable alternative would be to train all these models simultaneously with cloud computing with better specifications to decrease the runtime to train these models and reduce delays before any data analysis could be conducted.

To summarise the learnings from this project, it was an interesting area of research to learn how machine learning is applied in the foreign exchange market and to then implement this as a project to develop a trading agent to automate trades. Although the models were out-performed by simple and popular trading strategies, it shows that it is difficult to develop a strategy to out-perform commonly used strategies which are popular due to their simplicity and effectiveness in this domain.

References

[1] Andrew Brim. Deep reinforcement learning pair trading. *All Graduate Plan B and other Reports*, 1425, Decemeber 2019. doi: 10.1109/CCWC47524.2020.9031159.

[2] Andrew Brim. Deep reinforcement learning pairs trading with a double deep q-network. *Annual Computing and Communication Workshop and Conference (CCWC)*, 10, January 6-8 2020. doi: 10.1109/CCWC47524.2020.9031159.

[3] N. Buduma. Fundamentals of deep learning. designing next-generation machine intelligence algorithms, first edition. *O'Reilly Media*, pages 5–6, May 25 2017.

[4] Massimo Buscema. Back propagation neural networks. *Semeion Research Center of Sciences of Communication, pg. 239. Rome, Italy*, February 1998. doi: 10.3109/10826089809115863.

[5] WikiMedia Commons.
available: https://commons.wikimedia.org/wiki/file:recurrent_neural_network_unfold.svg. [Accessed August 23 2021], June 19 2017.

[6] A. Kendall et al. Learning to drive in a day. *Wayve.AI*, September 11 2018. doi: arXiv: 1807.00412.

[7] B. Zhao et al. Convolutional neural networks for time series classification. *Journal of Systems Engineering and Electronics*, Vol. 28(No. 1):162 − 169, February 2017. doi: 10.21629/JSEE.2017.01.18.

[8] M. Yani et al. Application of transfer learning using convolutional neural network method for early detection of terry's nail. *International Conference on Electronics Representation and Algorithm (ICERA 2019)*, 1201, 2019. doi: 10.1088/1742-6596/1201/1/012052.

[9] Mnjih et al. Human-level control through deep reinforcement learning. *Nature*, 518, February 26 2015. doi: 10.1038/nature14236.

[10] O. Vinyals et al. Starcraft ii: A new challenge for reinforcement learning. *Collaboration between DeepMind & Blizzard*, August 16 2017. doi: arXiv:1708.04782.

[11] V. Mnih et al. Playing atari with deep reinforcement learning. *NIPS Deep Learning Workshop 2013*, December 19 2013. doi: arXiv:1312.5602.

[12] Z. Wang et al. Dueling network architectures for deep reinforcement learning. *ICML'16: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, 48:1995 – 2003, June 2016. doi: arXiv:1511.06581v3.

[13] Z. Xiong et al. Practical deep reinforcement learning approach for stock trading. *NIPS 2018 Workshop on Challenges and Opportunities for AI in Financial Services*, December 2018. doi: arXiv:1811.07522.

[14] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics*, 36, 1980. doi: 10.1007/BF00344251.

[15] K. Valavanis G. Atsalakis. Surveying stock market forecasting techniques – part ii: Soft computing methods. *Expert Systems with Applications*, 36, 2009. doi: https://doi.org/10.1016/j.eswa.2008.07.006.

[16] GeeksforGeeks. Available: https://www.geeksforgeeks.org/depth-wise-separable-convolutional-neural-networks/. *[Accessed August 23, 2021]*, August 28 2019.

[17] D. Silver H. Hasselt, A. Guez. Deep reinforcement learning with double q-learning. *AAAI'16: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, page 2094–2100, February 2016. doi: arXiv:1509.06461.

[18] T. Horiuchi H. Sasaki and S. Kato. A study on vision-based mobile robot learning by deep q-network. *Proceedings of the SICE Annual Conference 2017*, Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE)(56), September 19-22 2017. doi: 10.23919/SICE.2017.8105597.

[19] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, Vol. 9 (No. 8):3–5, Novemeber 15 1997. doi: https://doi.org/10.1162/neco.1997.9.8.1735.

[20] L. Xiong J. Wu, C. Wang and H. Sun. Quantitative trading on stock market based on deep reinforcement learning. *International Joint Conference on Neural Networks*, 2019, July 14-19 2019. doi: 10.1109/IJCNN.2019.8851831.

[21] L.Chen and Q.Gao. Application of deep reinforcement learning on automated stock trading. *IEEE 10th International Conference on Software Engineering and Service Science (ICSESS)*, 2019, October 18-20 2019. doi: 10.1109/ICSESS47205.2019.9040728.

[22] Phi Michael. Towardsdatascience. available: https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21. *[Accessed August 23 2021]*, September 24 2018.

[23] Global Stock Market Prediction Based on Stock Chart Images Using Deep Q-Network. J. lee, r. kim, y. koh, and j. kang,. *Korea University*, February 28 2019. doi: 1902.10948v1.

[24] K. O'Shea1 and R. Nash. An introduction to convolutional neural networks. *Aberystwyth University, Lancaster University*, December 2015. doi: arXiv:1511.08458.

[25] J.Sen S. Mehtab and S. Dasgupta. Robust analysis of stock price time series using cnn and lstm-based deep learning models. *EEE Int. Conf. on Electronics, Communication and Aerospace Technology (ICECA'20), Coimbatore, INDIA*, 4th, Nov 5–7, 2020. doi: 10.1109/ICECA49313.2020.9297652.

[26] T. Zhou S. Wang and J. Bilmes. Bias also matters: Bias attribution for deep neural network explanation. *Proceedings of the 36th International Conference on Machine Learning*, 2019.

[27] Alex Sherstinsky. Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, Vol. 404:9, March 2020. doi: arXiv:1808.03314v9.

[28] R. Sutton and A. Barto. Reinforcement learning: An introduction. *MIT Press. Cambridge, MA*, November 5 2017.

[29] D. Ernsta T. Theate. An application of deep reinforcement learning to algorithmic trading. *Montefiore Institute, University of Li'ege*, October 9 2020. doi: arXiv:2004.06627.

[30] S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. *Conference Paper, Proceedings of the 1993 Connectionist Models Summer School. Erlbaum Associates*, June 1993.

[31] AI Wiki. Available: https://docs.paperspace.com/machine-learning/wiki/recurrent-neural-network-rnn. *[Accessed August 23 2021]*, 2019.

[32] I. Wang Y. Dai, C. Wang and Y Xu. Reinforcement learning for fx trading. *Stanford University.*