THE UNIVERSITY OF QUEENSLAND
AUSTRALIA

Deep Direct Reinforcement Learning for Financial Signal Representation and Trading

Authored by Joel Thomas

Bachelor of Commerce (Finance and Economics)

A thesis submitted for the degree of Master of Financial Mathematics at

The University of Queensland in November 2021

School of Mathematics and Physics

Declaration by Author

Acknowledgements

I would like to sincerely thank and acknowledge my supervisor Professor Duy-Minh Dang for encouraging me to pursue my research interests, supporting me through challenging hurdles as well as providing invaluable professional editorial feedback for my thesis.

Keywords

algorithmic trading, reinforcement learning, artificial neural networks, deep learning, recurrent neural networks, backpropagation through time, direct reinforcement learning, fuzzy learning, fuzzy deep recurrent neural network

Fields of Research (FoR) Classification

FoR code: 0103, Numerical and Computational Mathematics, 50%

FoR code: 0801, Artificial Intelligence and Image Processing, 50%

Table of Contents

1 Introduction 1

1.1 Algorithmic Trading 1

1.2 Selected Paper Summary 4

2 Background Theory and Preliminaries 6

2.1 Reinforcement Learning 6

2.2 Artificial Neural Networks and Deep Learning 8

2.3 Recurrent Neural Networks and Backpropagation Through Time 11

2.4 Direct Reinforcement Trading 13

2.5 Deep RNN for Deep Direct Reinforcement 14

2.6 Fuzzy Learning to Reduce Data Uncertainty 16

3 Implementation 18

3.1 Programming Language and Hardware 18

3.2 Dataset Retrieval and Preprocessing 19

3.3 Dataset Visualisation 20

3.4 Training and Testing Algorithms for the FDRNN Model 22

3.4.1 Pseudocode 22

3.4.2 Program Design 24

3.4.3 Verification Procedure 26

3.5 Experimentation Plan 27

3.6 Experimentation Methodology 29

4 Results and Discussion 30

4.1 Training Loss 30

4.2 Accumulated Rewards Over Training Period 32

4.3 Accumulated Rewards Over Testing Period 34

5 Conclusion 36

5.1 Summary 36

5.2 Limitations 37

5.3 Extensions 37

Bibliography 38

Appendix 40

Chapter 1
Introduction

1.1 Algorithmic Trading

Over the past decade, algorithmic trading has surged in popularity and use as a relatively new method to buy and sell financial assets via manmade computer-programmed algorithms. This field naturally emerged due to a combination of several factors:

- The increase in tertiary education accessibility allowing a greater number of individuals to pursue desired STEM degrees thereby lowering barriers to entry.
- Computerisation of financial markets enabling fully electronic markets – first introduced by the Designated Order Turnaround (DOT) and SuperDOT systems on the popular US NYSE in the early 1970s and 1984 respectively (Hasbrouck et al., 1996).
- Exponential growth in raw computing power driven by innovation and competition between major computer hardware corporations such as Intel, AMD, NVIDIA, etc.
- Increasing accessibility to said raw computing power driven by competition between major vendors such as HP, Dell, Lenovo, Amazon Web Services, Google Cloud, Microsoft Azure, etc.

The last two points roughly follow Moore's Law – the observation that the number of transistors on a microchip double every two years although the cost of these newer computers become halved (Moore, 1965).

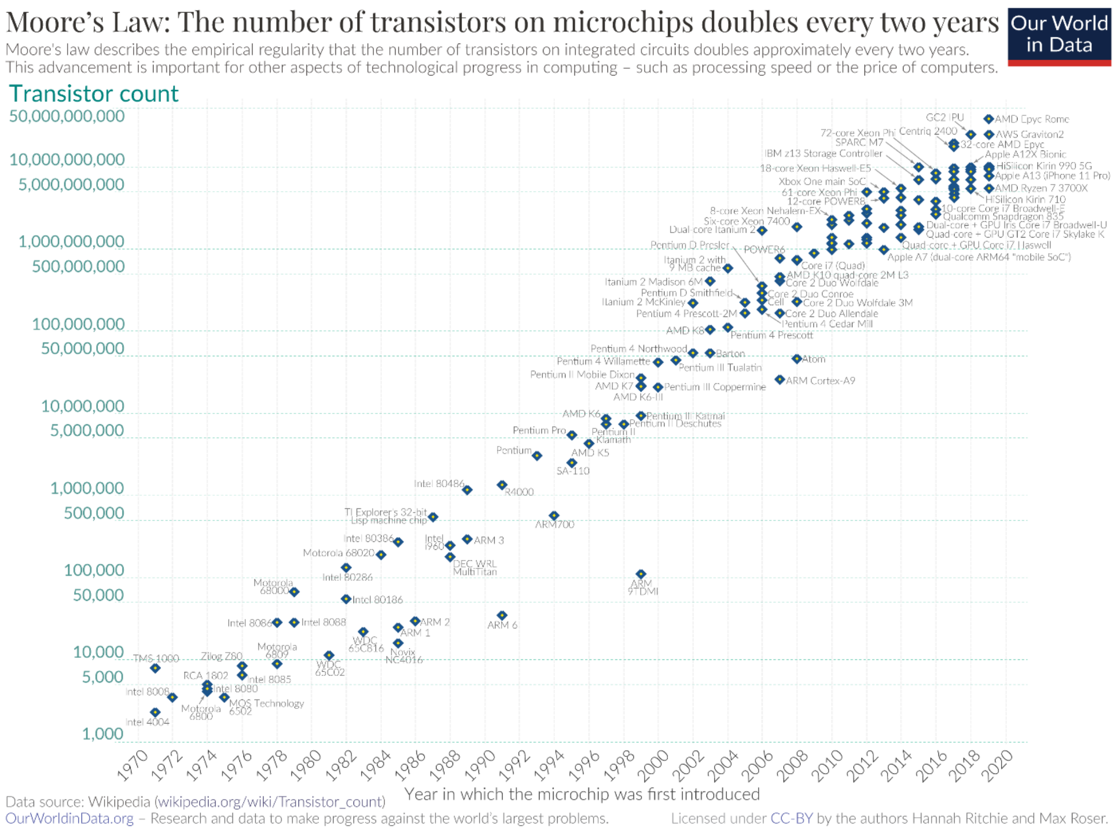


Figure 1: Visualisation of Moore's Law in practice. Observe that as time goes on, the number of transistors on newer processors increase implying higher performance through faster processing speeds (Ritchie et al., 2013).

Algorithmic trading had such a profound impact during the early years of its emergence because one of the biggest advantages it provided and continues to provide over manual trading is the ability to entirely eliminate human emotions out of any decision-making element. This is because an automated trading algorithm executes decisions solely based on the instructions it was initially programmed to perform. It does not experience joy, excitement, fear, frustration, or anxiety when the market moves favourably or unfavourably which could lead to irrational and clouded decision-making in humans. Furthermore, trade execution speed is another major benefit provided by algorithmic trading. Whilst dependent on several factors such as computer hardware, distance to brokerage server, strategy specifics, etc., general retail and institutional trading algorithms that used to take a few seconds in the past now take a mere 10 milliseconds on average to process new market information and execute orders using new information. High Frequency Trading (HFT) at the highest level is unbelievably fast –it takes just 5-10 *micro*seconds for strategies employed by HFT firms to do the same (Aquilina et al., 2021). Bear in mind that these algorithms are not limited to single financial assets – they are able to accurately scan large portions of multiple financial sectors at a time. This is all in comparison to humans that on average take roughly 300-400 milliseconds just to blink eyes. In fact, this gap became so severe that the US IEX implemented a 350-microsecond order delay to promote market fairness by protecting investors' stale orders from being scalped by HFT (Hu, 2018).

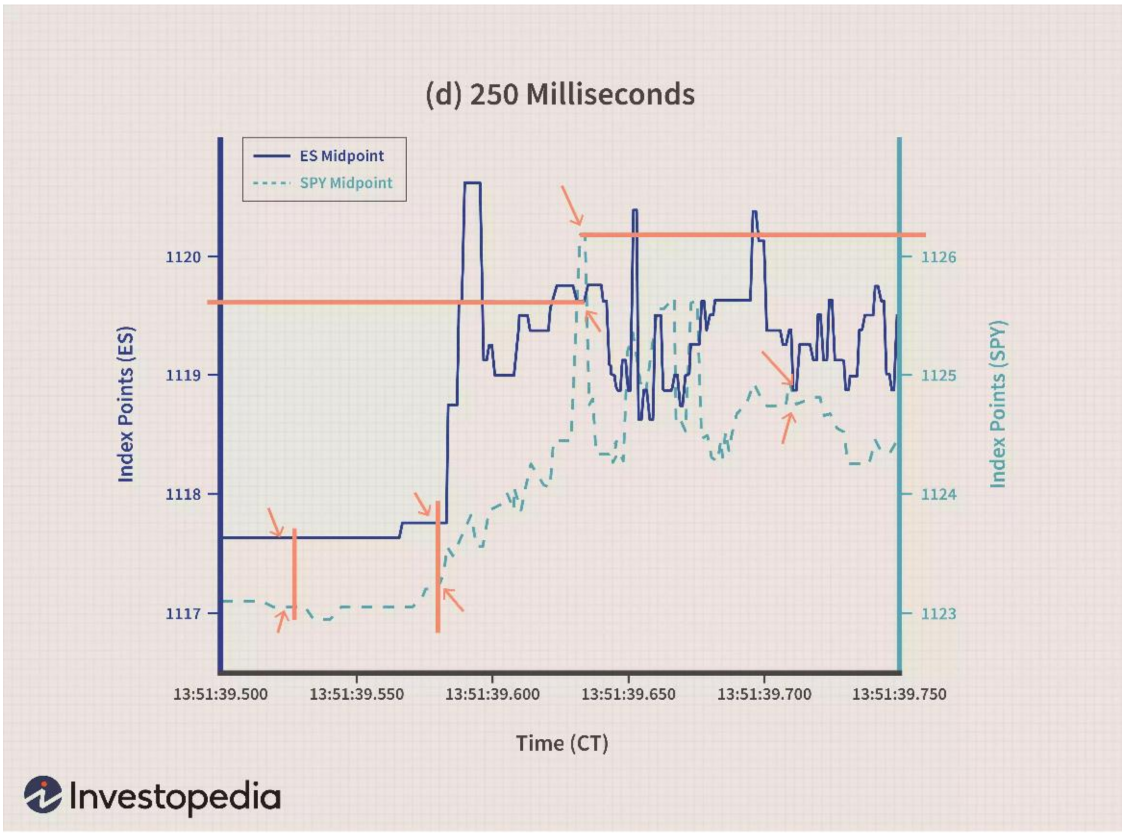
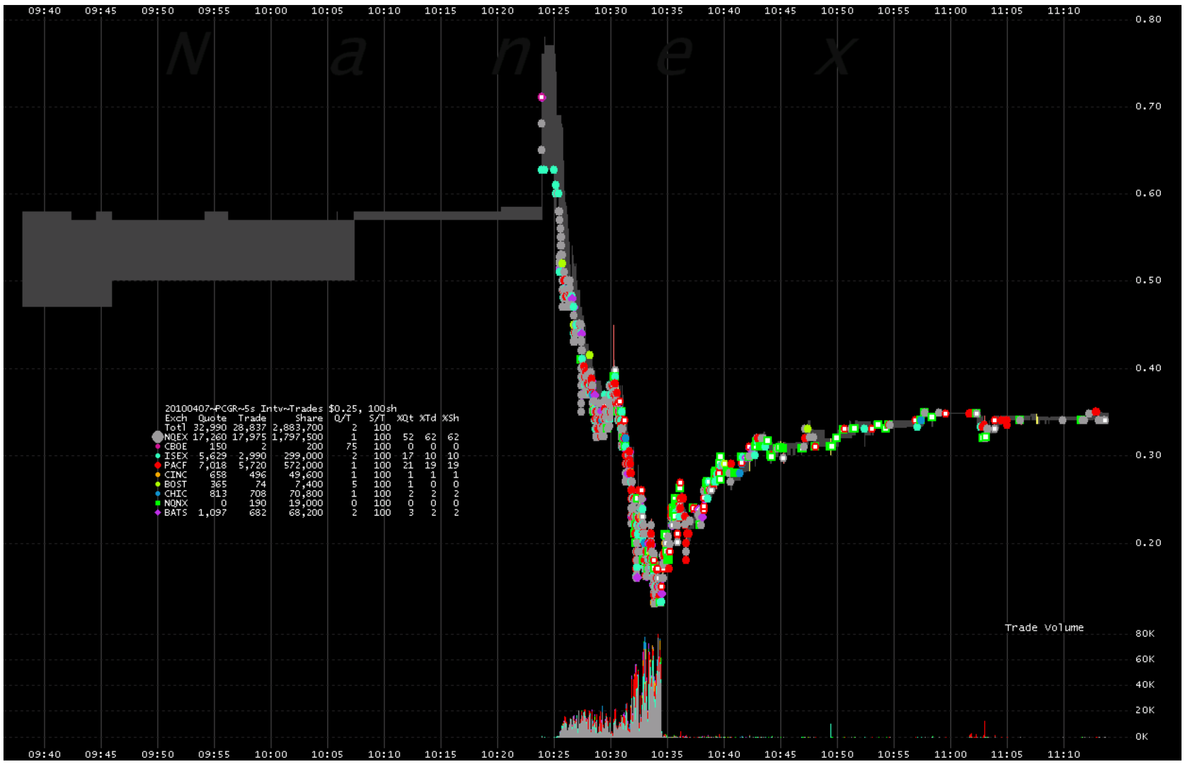


Figure 2: Tick movement activity of E-mini S&P 500 futures (ES) and SPDR S&P 500 ETFs (SPY) at the millisecond level (Seth, 2015).

Finally, algorithmic trading also enables investors to automate thorough backtesting of their strategies rather than tediously perform it manually. Backtesting is the process of assessing a trading strategy's viability by executing it on past historical data and generating summary/descriptive statistics based on its performance (Ni et al., 2007). However, note that a reliable automated backtest relies on the elimination of survivorship bias and data selection bias on the user side and elimination of look-ahead bias on the execution side. It must also consider realistic market factors such as market impact, transaction costs, order execution time, stop loss effects, adjustment in financial statements, stock splits, etc.

Despite the advantages algorithmic trading brings, it also carries several disadvantages (Kumar, 2016). It is quite difficult to trade algorithmically (let alone profitably) in practice due to the inherent steep learning curve associated with this field. Depending on the type of strategy, an investor could require advanced prerequisite knowledge in Mathematics, Probability, Statistics and Computer Science in order to successfully create and program their ideal sophisticated automated trading algorithm all on their own. Hiring another individual is possible but this is equivalent to disclosing the strategy. Furthermore, program debugging can be very challenging – unintended orders made can be caused by issues residing in the program code, program logic, loss of connection to broker server, inaccurate/inconsistent backtest engine, etc. These issues hence give rise to another disadvantage – the need for constant surveillance while the algorithm is running to prevent unintended orders. The importance of monitoring is showcased in a notable example displayed in Figure 3 below. On April 7th, 2010 at 10:07:23 am, Citadel's rogue algorithm took PC Group's stock down from \$0.71 to \$0.13 (82% loss) in roughly 10 minutes (Nanex Research, 2014). The trade rate accelerated in the final 3 minutes prior to Citadel manually shutting down its bot. Although the average investor does not have sufficient capital to cause this type of worst-case possible outcome, the example still showcases how challenging and dangerous algorithmic trading can be.



```
20100407~PCGR~5s Intv~Trades $0.25, 100sh
Exch   Quote   Trade     Share  Q/T   S/T  %Qt %Td %Sh
Totl  32,990  28,837 2,883,700    2   100
● NQEX 17,260  17,975 1,797,500    1   100   52  62  62
● CBOE    150       2       200   75   100    0   0   0
● ISEX  5,629   2,990   299,000    2   100   17  10  10
● PACF  7,018   5,720   572,000    1   100   21  19  19
● CINC    658     496    49,600    1   100    1   1   1
● BOST    365      74     7,400    5   100    1   0   0
● CHIC    813     708    70,800    1   100    2   2   2
■ NQNX      0     190    19,000    0   100    0   0   0
◆ BATS  1,097     682    68,200    2   100    3   2   2
```

Figure 3: Citadel's rogue algorithm on April 7th, 2010. Total of 2.75 million shares shorted (≈ 23707% average daily volume), majority across several trades in lots of 100 shares. The only apparent pre-trade risk was checking if the order size was under 1 million shares (Nanex Research, 2014).

In general, there are several popular classes under which algorithmic trading strategies fall (Mandes, 2016). The first of these contain highly common trend-based strategies that focus on the use of technical indicators to infer trends in moving averages, momentums as well as volumes traded for one or multiple financial assets. These are the easiest to understand and implement. Next, a class of mathematical analytical models exist that concentrate on utilising the concepts of market-microstructure and game theory. An example is the Almgren-Chriss model whose objective is to find an optimal execution sequence for (multiple) portfolio transactions (Almgren et al., 2000). It is quite sophisticated since it also considers endogenous price evolution brought about by the market impact from trading – other market participants are able to observe and take into account any rebalancing trades thereby adjusting their own buy and sell prices accordingly

(Almgren et al., 2000). Within the analytical model's class, there is also a subclass known as adverse selection models which attempt to approximate the correct bid-ask spread in a market containing a specialist trader (such as a market maker) along with "informed" trader/s (Mandes, 2016). Examples of these are the Glosten-Milgrom and Kyle models. Portfolio reallocation is another class of strategies – examples include alpha strategies (focussing on Sharpe ratio, dynamic weights, etc.) and delta-neutral strategies. The class of statistical arbitrage strategies focus on using mathematics and statistics to implement models that profit on short-term severe market inefficiencies – the historically most popular being the pairs trading strategy which employs several time series analysis concepts (Mandes, 2016). Finally, another major class of strategies is formed by the subsets of Artificial Intelligence known as Machine Learning and Reinforcement Learning (Ryś, 2018). Whilst a thorough description of both these fields is provided in Chapter 2 later on, for now, it is noted that these have many use cases for algorithmic trading. Examples include news and event based predictions, sentiment analysis, predicting likelihoods of structural breaks, optimal bet sizing, feature selection for trading and even portfolio (re)allocation (Ryś, 2018).

1.2 Selected Paper Summary

Following from the extensive discussion of algorithmic trading above, this project thesis aims to implement and test an innovative algorithmic trading strategy based on a novel paper that makes use of various elements of Artificial Intelligence (AI) in developing the strategy. The specific paper this strategy originates from is titled "Deep Direct Reinforcement Learning For Financial Signal Representation and Trading" and is authored by Yue Deng, Fend Bao, Youyong Kong, Zhiquan Ren and Qionghai Dai (Deng et al., 2017). It should be noted that the strategy developed in this paper is completely non-parametric or data-driven. In other words, it is trained completely using raw market data and does not involve calibrating parameters for an underlying model from the mathematical finance literature.

Deng et al. begin their paper by acknowledging the popularity of automated financial assets trading from the perspective of the artificial intelligence community. They approach the problem of automated trading as an online decision-making task involving two major subproblems – market condition summarisation and optimal action execution. Popular algorithms arising from the field of Reinforcement Learning (RL) such as Q-learning and SARSA are introduced as one way to enable online (real-time) dynamic decision-making without human supervision/interference. This is by allowing an agent (trading bot) to explore an unknown environment (time series of a financial asset) whilst simultaneously learn to make correct decisions (profitable trades). These traditional algorithms suffer in this context because whilst they try to find optimal policies for optimal action execution, it will be extremely challenging to do so without using an efficient representation of market conditions i.e. tackling the first abovementioned subproblem. Moreover, optimal action execution on the basis of past actions does not only imply actions that generate profitable trades – it also means having to consider other factors such as transaction costs and slippage relevant to this context.

Following these realisations, Deng et al. then proceed to build a fully automated RL trading system incrementally. Initially, a Deep Artificial Neural Network (DNN) is employed to solve the first subproblem via feature learning as well as more robust feature representation. In addition, a Recurrent Neural Network (RNN) is also employed to solve the second subproblem. They further improve their system's market summarisation robustness by utilising fuzzy learning concepts to reduce input data uncertainty. The last step includes aiding the training process of their system by a few advanced (system-related) parameter initialisations as well as a very sophisticated extension of the general training algorithm used to train RNNs which they name as *Task-Aware* Backpropagation Through Time. At this point, their final model is named as the Fuzzy Deep Recurrent Neural Network (FDRNN). Pseudocode summarising the entire training algorithm for the FDRNN model is then provided.

After having implemented the pseudocode into code, they detail their experimental setup including the specific parameters, hyperparameters and datasets used in evaluating the performance of the FDRNN in terms of accumulated rewards i.e. profit and loss over time. Promising results are showcased along with a comparison of the FDRNN against prediction-based DNNs such as the Convolutional Neural Network (CNN), the RNN and the Long-Short Term Memory (LSTM) models. Next, the authors discuss the robustness of their system by summarising how the some of the FDRNN's hyperparameters were tuned before finally reviewing their findings in their conclusion.

The remainder of this project thesis aims to replicate parts of the abovementioned sections of the paper by Deng et al. in detail. The entirety of the paper could not be replicated due to time constraints. Chapter 2 aims to provide all the necessary prerequisite background knowledge required to understand the internal specifics of the FDRNN model well. If any terminology in this section unintendedly confused the reader, please refer to this Chapter for more detail. Chapter 3 showcases a closer look at the FDRNN in practice through a personal implementation of the model to the best of personal ability. This includes providing an overview of the chosen programming language and hardware used; retrieval, preprocessing and visualisation of datasets; pseudocode for training and testing the FDRNN, program design specifics and verification procedure; experimentation plan and experimentation methodology. Results following execution of the experimentation plan as per the experimentation methodology are discussed extensively in Chapter 4. Finally, Chapter 5 concludes the thesis by summarising the findings, limitations and extensions applicable to the model that were discovered over the course of MATH7013 this semester whilst undertaking this project.

Chapter 2

Background Theory and Preliminaries

2.1 Reinforcement Learning

Reinforcement Learning (RL) is defined as the branch of AI concerned with how intelligent agents decide on taking actions in an unknown environment that enable them to maximise some form of cumulative reward. In the context of algorithmic trading, there is a single agent which is the automated trading bot, the unknown environment is given by the time series of some financial asset(s), the actions available are generally $\{sell, neutral, buy\}$ and the cumulative reward is given by accumulated profit and loss since the bot's inception.

The dimensions of complexity for intelligent agent design were originally created to make progress in AI research via a series of simplifying assumptions, each of which gave rise to a dimension of complexity that ranged from simple to complex. Since these assumptions could be relaxed in various "combinations", different combinations lead to different points in space (Poole et al., 2010). Generally, traditional RL has the following dimensions of complexity:

Dimension	**Complexity**				**Explanation**
Modularity	Flat	Modular	Hierarchical	-	Model at one level of abstraction.
Representation	Explicit states	Features	-	-	States of the environment described explicitly or using features.
Planning horizon	Static	Finite stage	Indefinite stage	Infinite stage	Agent reasons about a finite (not predetermined) number of time steps or plans on going on forever.
Sensing uncertainty	Fully observable	Partially observable	-	-	Agent can observe the current state of the environment.
Effect uncertainty	Deterministic dynamics	Stochastic dynamics	-	-	Given agent knows the current state and action taken, there is uncertainty about the resulting state.
Preference	Goals	Complex preferences	-	-	Involves trade-offs between various desiderata at potentially differing times.
Number of agents	Single agent	Multiple agents	-	-	Single agent reasons in the environment, any other agents are part of the environment.
Learning	Knowledge is given	Knowledge is learned	-	-	Agent learns knowledge from data or past experience.
Computational limits	Perfect rationality	Bounded Rationality	-	-	Agent can determine best action without considering its limited computational resources.
Interaction	Offline	Online	-	-	Agent reasons while interacting with the environment.

Table 1: Dimensions of complexity for traditional RL (Poole et al., 2010).

Applying RL to the problem of algorithmic trading can be visualised as in Figure 4 below. Here, the agent (trading bot) takes actions ($\{sell, neutral, buy\}$) in the unknown environment (time series of some financial asset(s)) in the current state (current price) which leads to the realisation of a reward (profitable/unprofitable trade) and transition to the next state (future price). The reward and next state are both fed back into the agent.

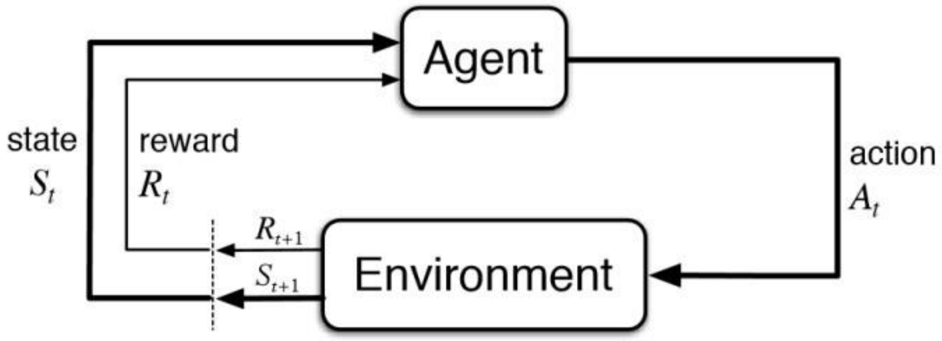


Figure 4: Typical formulation of an RL scenario (Bhatt, 2018).

A brief description of popular traditional RL algorithms such as Q-learning and SARSA shall now be discussed. Without diving into these algorithms' prerequisites such as Markov Decision Processes, Value Iteration and Policy Iteration, essentially, both Q-learning and SARSA fill up a 2D table called the Q-table which is denoted by $Q(s,a) \in \mathbb{R}^{|S|\times|A|}$ where $s =$ a state, $a =$ an action, $S =$ set of all possible states, $A =$ set of all possible actions defined for a specific scenario. In this Q-table, after an algorithm's convergence, each element represents the optimal value (expected discounted reward) of performing an action a given a state s of the environment. Hence, the best possible action then for a given state s is given by the $\arg\max_a Q(s,a)$ along the table's row. Whilst both algorithms can be solved efficiently using dynamic programming, the difference lies in the type of learning these algorithms perform. Q-learning does off-policy learning which means $Q(s,a)$ is learnt by an optimal policy no matter what the initially provided policy is (Hausknecht et al., 2016). In contrast, SARSA does on-policy learning which means $Q(s,a)$ is learnt by following the initially provided policy itself. Note that a policy here refers to one of the many actions $a \in A$ that the agent must take given some particular state in the environment. An example of a policy is acting greedily (choosing the action in a state with known highest reward based on past experience) 70% of the time and acting randomly the remaining 30% of the time.

The main reason why neither of these algorithms are well-suited to algorithmic trading is because the set of all possible states is discrete here i.e. $|S| < \infty$. In contrast, the set of all possible states given by the price of a financial asset is usually assumed to be continuous in nature. One possible solution is to discretise this set by rounding prices up or down a decimal, but this results in both lost information and large time and space computational complexity since the Q-table can become very large depending on the desired decimal accuracy. Moreover, using a discretised set, the largest state on the Q-table can always be surpassed by some future price, no matter how unlikely that event could be. Finally, since extreme values (large or small) might not occur during the training process, both algorithms will be forced to choose an action a completely at random when encountered with such values outside of model training. Essentially, this pitfall boils down to the market condition summarisation subproblem Deng et al. pointed out in the beginning of their paper (Deng et al., 2017).

2.2 Artificial Neural Networks and Deep Learning

Artificial Neural Networks (ANNs) or Multilayer Perceptrons (MLPs), with the terms used interchangeably, are a computational model created to simulate the way neurons within the human brain communicate with each other (IBM, 2020). They fall under the Machine Learning (ML) branch of AI that focusses on the usage of data and special algorithms to build models capable of make predictions or decisions without requiring explicit programming. Furthermore, these predictions and decisions tend to gradually improve over time via the concept of model training (IBM, 2020). Whilst the purpose of both ML and RL seem similar by their definitions, their approaches differ vastly making them very different subsets of AI. For example, the notion of agent and environment, maximising reward, optimal policy, online learning etc. in RL compared to training and testing, minimising loss, offline learning, etc. in ML (Ravichandiran, 2020). ANNs are the most basic class (subset) of Deep Neural Networks (DNNs). A graphical illustration of a typical ANN is provided in Figure 5 below.

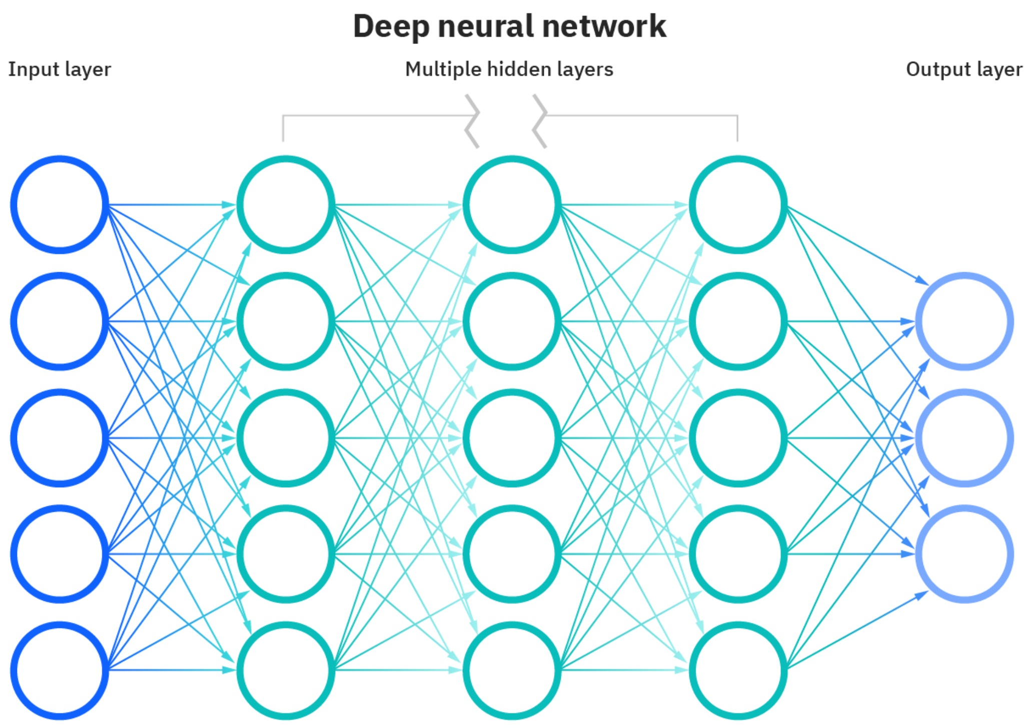


Figure 5: A typical ANN consisting of an input layer, multiple hidden layers and an output layer (IBM, 2020).

An ANN usually consists of one input layer containing input nodes (features), followed by a single or multiple hidden layers containing hidden nodes followed by the final output layer containing potentially several output nodes (outputs). Whilst the choice of how many input and output nodes to have in an ANN is problem-specific, the number of hidden layers as well as number of hidden nodes in each hidden layer are hyperparameters left completely up to the user providing an incredible amount of flexibility in model creation (Zhang et al., 2020). Parameters such as these and some others are labelled hyperparameters because they directly control how well a model trains thus affecting its prediction-/decision-making ability on new unseen data (Zhang et al., 2020).

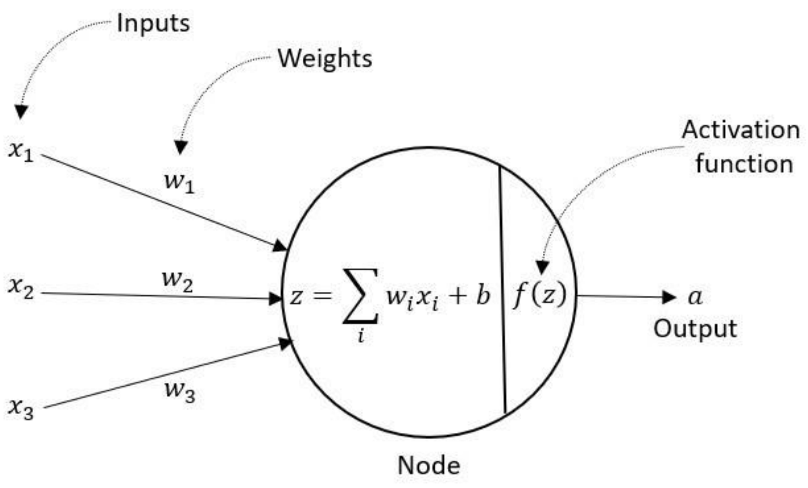


Figure 6: Close-up view of a single hidden/output node (Larson, 2020).

Figure 6 above provides a closer view of the internals of a single hidden/output node from Figure 5. If this is treated as a hidden node, then the inputs refer to input/hidden nodes belonging to the first input/previous hidden layer and the output is passed to the next hidden/final output layer. If instead treated as an output node, then the inputs refer to hidden nodes belonging to the previous hidden layer and output refers to the final prediction produced by the ANN. Generally, there is a separate set of weights w_i and bias term b used for every node in an ANN (hidden and output layers) except the input nodes which just take in as input the raw input features (Zhang et al., 2020). Observe that the expression $z = \sum_i w_i x_i + b$ is of the same form one might expect to see in multiple linear regression. Subsequently, the bias term b acts as an additional trainable parameter for every hidden and output node to better the model's ability to fit the given data. Finally, an activation function $f(z)$ is a function that "activates" the output of z if when passed into this function, z manages to exceed some fixed threshold (IBM, 2020). Its aim is to serve two purposes – to transform the output of the current node to input for the next node(s) and also "squash" the output into some range, usually $\in (0,1)$ or $\in (-1,1)$. Examples of commonly used activation functions are (Zhang et al., 2020):

- The sigmoid: $f(z) = \frac{1}{1+e^{-z}}: \mathbb{R} \to (0,1)$.
- The hyperbolic tangent: $f(z) = \frac{\sinh(z)}{\cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^{-z}}: \mathbb{R} \to (-1,1)$.
- The rectified linear unit (ReLU) $f(z) = \max(0, z) = (z)^+: \mathbb{R} \to [0, z]$.

Once the final set of outputs are obtained given a set of inputs, a loss function can be utilised to provide a sense of how accurate the model's final prediction is in contrast to the true output or answer (Zhang et al., 2020). It is at this point that it becomes necessary to differentiate between Supervised and Unsupervised Learning as well as between Classification and Regression problems. Supervised learning refers to an approach that uses labelled datasets designed to train algorithms into predicting outputs accurately whereas unsupervised learning is an alternative approach that instead uses unlabelled datasets designed to enable algorithms uncover hidden patterns in the data (Delua, 2021). Within supervised learning, classification problems refer to discrete prediction problems e.g. predicting a category whereas regression problems refer to continuous prediction problems e.g. predicting some quantity. While neural networks in general can be both supervised and unsupervised, ANNs fall under supervised learning. Loss functions only exist in supervised learning (since a true solution exists to compare to) and are logically different for classification versus regression problems (Zhang et al., 2020). An example for each problem is the sum of squared errors (from linear regression) and cross entropy respectively.

Once the loss between the final predicted output and true output for a given set of inputs is computed, error backpropagation can take place. Backpropagation is the usual algorithm used for training feedforward train neural networks such as an ANN. It works by computing the gradient of the loss function with respect to each of the trainable parameters i.e. all weights and biases (Zhang et al., 2020). Heavy use is made of the chain rule from ordinary calculus to compute each gradient one layer at a time from the output layer back towards the input layer (hence the term backpropagation) (Goodfellow et al., 2016). In cases where the activation function is directly differentiable, a dynamic programming approach can also be utilised to implement backpropagation massively increasing computational efficiency by avoiding recomputation of intermediate terms appearing in the chain rule (Goodfellow et al., 2016). The mathematics of backpropagation need not concern the reader as it can be quite challenging to follow due to notation (e.g. term belonging to specific node of a specific layer, etc.) and most deep learning programming libraries perform backpropagation and other more sophisticated training algorithms automatically. In fact a manual attempt at backpropagation should only ever be performed as a learning exercise! Figure 7 below provides a visual illustration of why training an ANN's parameters is important.

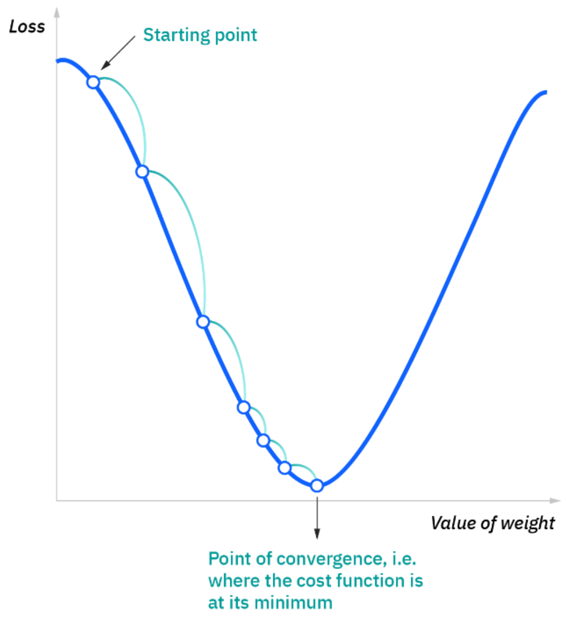


Figure 7: The goal of ANN training is to minimise the loss function to improve its prediction performance on the sample data (IBM, 2020).

Once the gradient of the loss function with respect to one of the weight parameters has been computed via backpropagation, the weight must be iteratively updated in such a way that decreases the loss until no further reduction is gained from additional updates i.e. loss is minimised at convergence. Optimisers are algorithms or mathematical functions designed specifically to achieve this task by taking "steps" in the right direction as per Figure 7 (Zhang et al., 2020). Gradient descent is the most common ML optimisation algorithm that uses first-order derivatives to iteratively find the local minimum of a differentiable function. It is similar to the Newton Raphson method except the latter has stronger constraints due to requiring the existence and use of the second-order derivative (hence, often converges faster). Gradient descent works by taking steps in the direction that is the negative of the gradient of the loss function at the current point. These steps are proportional since a learning rate is used to dampen the step size. Several other optimisers exist such as Momentum, Adagrad, RMSProm and Adam. Each optimiser attempts to improve on the previous one by introducing notions of momentum and adaptive learning rate (Zhang et al., 2020).

Finally, model training can be summarised as follows. Given a training set which is the set of all samples used to train an ANN, the loss function is used to compute the loss/error for each of the predicted outputs from the corresponding training samples. For each input sample, backpropagation together with some optimiser is used to update weights and biases in the direction that decreases the loss. Finally, this process repeats indefinitely until convergence of the loss function at which point one can conclude that the model's training has converged to some local (hopefully global) minima. The model's ability to generalise on new unseen data can then be evaluated using a separate testing set different to the original training set.

2.3 Recurrent Neural Networks and Backpropagation Through Time

Recurrent Neural Networks (RNNs) are another subclass of DNNs derived from ANNs that are specifically designed to handle sequentially dependent data such as time series data (Zhang et al., 2020). In contrast, ANNs are usually used for cross-sectional (non-time dependent) data. RNNs are able to handle sequential data by generally allowing self-loops of the hidden nodes in a hidden layer which implicitly translates to existence of an internal memory (Elman, 1990). Hence, variable length sequences of inputs can be provided because the RNN will consider the current input in addition to the learning gained from previous inputs. This is as opposed to ANNs which do not have any notion of "order" or time. A graphical illustration of a typical RNN is provided in Figure 8 below.

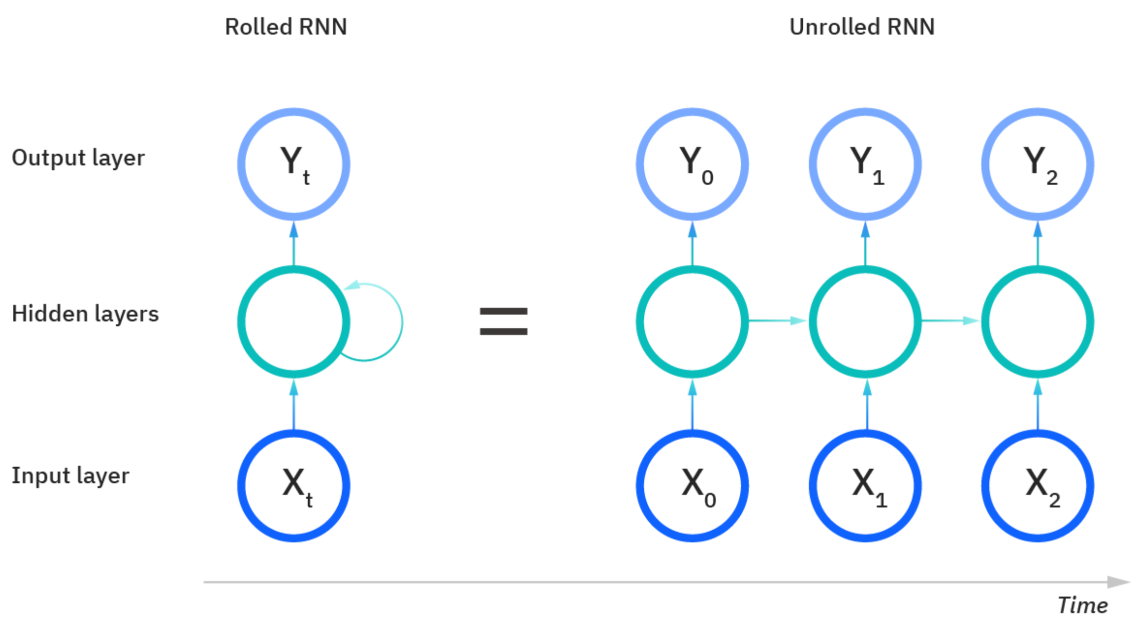


Figure 8: A typical RNN in its rolled and unrolled form (IBM, 2020).

The bulk of the discussion in Section 2.2 above is also applicable to RNNs with a minor exception on the model training algorithm. Backpropagation Through Time (BPTT) is a generalisation of regular backpropagation specifically designed to train special types of RNNs, an example being an Elman Neural Network (Zhang et al., 2020). This is the same type of RNN showcased in Figure 8 and used for the remainder of the project in implementing the FDRNN model in Chapter 3. For these types of special RNNs, BPTT involves unrolling or unfolding the network through time prior to conducting regular backpropagation to update all of the unrolled network's trainable parameters (Cao, 2018). The network must be unrolled because strictly speaking, the hidden nodes in the hidden layer are not exact self-loops back to themselves but rather to additional nodes in a "context" layer that then connect back to the hidden nodes (Cao, 2018). This is what enables the existence of internal memory as the current time's context layer's nodes store the outputs from the previous time's hidden nodes. A detailed visualisation is provided in Figure 9 below for further reference.

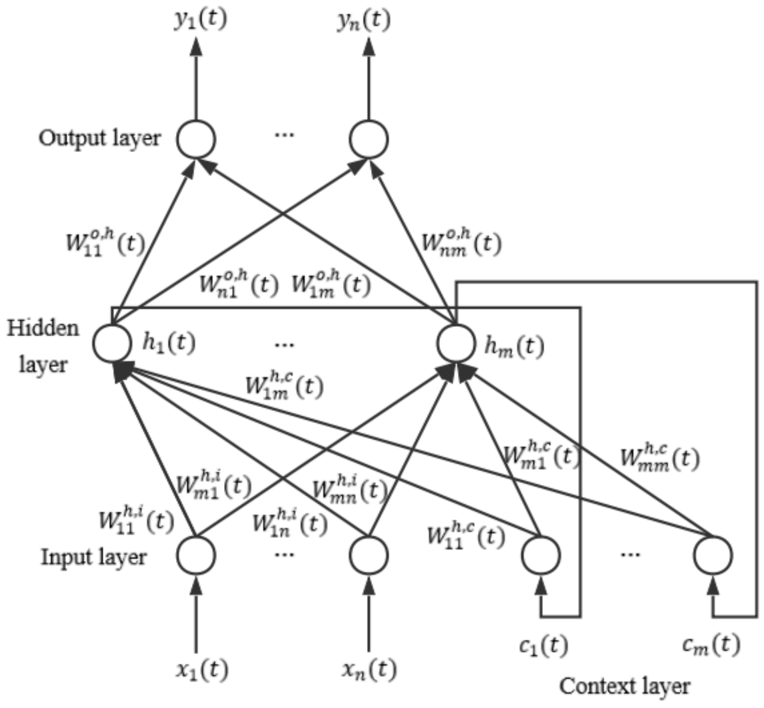


Figure 9: The Structure of the Elman Neural Network (Cao, 2018).

Finally, it must be noted that RNNs are generally much more difficult to train than typical ANNs. This is because unrolling an RNN in BPTT can cause it to become very deep – each timestep that an RNN is unrolled for is proportionally equivalent to making a copy of the original network but without any self-loops (Zhang et al., 2020). The output from one network at the current timestep is passed directly as input into the next network at the next timestep (see Figure 9). The end result of unrolling can potentially be a very long feedforward network with a very large number of trainable parameters. Furthermore, the products derived by the chain rule of ordinary calculus during backpropagation to calculate the gradient of the loss function with respect to the trainable parameters become much longer leading to two major issues – vanishing gradients and exploding gradients (Zhang et al., 2020). Each of these can occur when the terms being multiplied in the products are either extremely small or extremely large leading to the final gradient to either disappear (hence the term "vanish") or completely blow up (hence the term "explode") (Nielsen, 2015). This is disruptive for the training process because the incremental steps observed in Figure 7 either yield no change at all or completely step over the local/global minimum (perhaps even jump out of the current "valley" and enter another one). In other words, convergence of the loss function to a desirable minimum can become impossible. Some methods to solve the vanishing and exploding gradients problems include using gradient clipping, weight regularisation or alternative activation functions (Zhang et al., 2020). Weight regularisation (specifically L2 regularisation) is used for this thesis which works by penalising (adding to) a network's loss function using squared weights. Logically, this attempts to tackle the issue of exploding gradients by discouraging the use of larger weights during model training since these would naturally result in a proportionally higher loss.

2.4 Direct Reinforcement Trading

Now that the reader is equipped with the required background theory in RL, ANNs and RNNs, the incremental choices and decisions made by Deng et al. in developing their novel FDRNN model start to become clearer. The initial model produced by them involves using John Moody's and Matthew Saffell's constructed Deep Reinforcement Learning (DRL) framework. This is from their own paper titled "Learning to Trade via Direct Reinforcement" back from 2001 (Moody et al., 2001).

The mathematical notation and reasoning behind various terms defined by Deng et al. shall now follow (Deng et al., 2017). Given a sequence of prices $p_1, p_2, \dots, p_{t-1}, p_t, p_{t+1}, \dots, p_{T-1}, p_T$ of some financial asset arriving from the exchange centre, the return on this financial asset at time t is defined as $z_t = p_t - p_{t-1}$ for $t = 2, \dots, T$. Typically, sophisticated mathematical finance models for modelling the price dynamics of financial assets allow for continuous t but in practice, there is always some latency between the next price p_{t+1} and the current price p_t in the sequence of prices. Consequently, t is also discrete here ($t = 2, \dots, T$). Now using RL terminology, the policy (recall set of all possible actions) is given by $\delta_t \in \{sell, neutral, buy\} = \{-1,0,1\}$ that must be used at each time t (state of the environment) in order to make trading decisions in real-time. Defining the transaction cost of making a single trade as a constant c, the profit/loss made at time t is then naturally given by $R_t = \delta_{t-1} z_t - c|\delta_t - \delta_{t-1}|$. The first expression evaluates to the profit/loss made from natural price movement whereas the latter term is used as penalisation – observe that in this form, switching positions regularly is discouraged in the model because these costs can add up over time diminishing profitability. The transaction cost c only applies when $\delta_t \neq \delta_{t-1}$, otherwise it disappears when no position change is made. Usual DRL frameworks use the same expression for R_t above as the value function. Hence, the accumulated value for the entire training period is then defined as $\max_{\Theta} U_T\{R_1, \dots, R_T|\Theta\}$.

In other words, the goal here is to maximise the accumulated value over the entire training period by maximising some function U_T with respect to a set of trading parameters Θ. A logical function to use to represent U_T appropriately is the total profit/loss made from $t = 1, \dots, T$ given accordingly by $U_T = \sum_{t=1}^{T} R_t$. The final question remains as how to perform this maximisation task. Traditional RL algorithms such as Q-learning and SARSA, whilst they both do utilise dynamic programming providing large computational efficiencies, will not be able to solve the dynamic trading task. This is directly due to the pitfalls mentioned towards the end of Section 2.1 above – using a set of discrete states to represent the entire state space will not work due to a lack of effective market condition summarisation. Moody et al.'s DRL framework proposes an alternative innovative method to directly learn the policy δ_t (Moody et al., 2001):

$$\delta_t = \tanh(<\boldsymbol{w}, \boldsymbol{f}_t> + b + u\delta_{t-1})$$

In this expression, $\boldsymbol{w}$ and b refer to the weights and bias for feature regression and $\boldsymbol{f}_t = [z_{t-m+1}, \dots, z_t] \in \mathbb{R}^m$ is the feature vector that represents the m most recent asset returns. Accordingly, $\boldsymbol{w} \in \mathbb{R}^m$ as well. The extra term $u\delta_{t-1}$ is utilised to induce the effect of requiring the model to consider past trading decisions as well e.g. from the build-up of transaction costs from regular position switching that can diminish profitability. Moreover, recall from Section 2.2 that the hyperbolic tangent always maps an input to an output $\in (-1,1)$ which contains the different actions available from the policy $\delta_t \in \{-1,0,1\}$.

The final optimisation problem is hence given by:

$$\max_{\Theta} U_T\{R_1, \dots, R_T|\Theta\} = \max_{\Theta}\{\textstyle\sum_{t=1}^{T} R_t\,|\Theta\}$$

$$s.t.\ R_t = \delta_{t-1}z_t - c|\delta_t - \delta_{t-1}|$$

$$\delta_t = \tanh(<\boldsymbol{w}, \boldsymbol{f}_t> + b + u\delta_{t-1})$$

$$\Theta = \{\boldsymbol{w}, b, u\}$$

The abovementioned framework can be implemented and visualised as a single hidden layer RNN with the recurrent link forming at the hidden node that outputs δ_t back to the input layer. An illustration is provided in Figure 10 below and is straightforward to follow after reviewing the background theory on RNNs in Section 2.3. The BPTT training algorithm can be used to train the DRL as it is an example of the Elman Neural Network which was described earlier. However, based on previous discussion, there will also be an increased possibility of vanishing or exploding gradients occurring due to RNN unrolling yielding a very deep feedforward neural network.

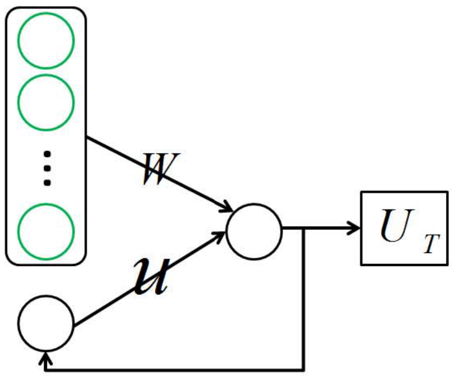


Figure 10: The DRL model for automated financial assets trading (Deng et al., 2017).

2.5 Deep RNN for Deep Direct Reinforcement

The abovementioned DRL model is intuitive but limited because while its purpose is to learn the optimal policy for profitable trading decision-making, it will not be able to do so without an effective summarisation of market conditions (first subproblem mentioned in Deng et al., 2017). However, the general framework is still better suited than traditional RL to the problem of algorithmic trading since now there is no need to store/discretise all of the possible states of the environment e.g. via a Q-table. For effective market condition summarisation, it is necessary that either the raw inputs to the DRL model are changed from $\boldsymbol{f}_t$ to some other feature vector that reflects this condition or each $\boldsymbol{f}_t$ must be transformed into an alternative representation. This is precisely where ANNs and all the relevant preliminary theory discussed in Section 2.2 come in. Recall that the deep transformations that take place from layer to layer in an ANN can yield a much more informative feature representation that can then be used to better the learning for a given task.

Deng et al. modify Moody et al.'s original DRL model to incorporate an ANN into the framework in an attempt to target both the market condition summarisation and optimal action execution subproblems (Deng et al., 2017). Utilising their notation, the deep representation of $\boldsymbol{f}_t$ is given by $\boldsymbol{F}_t = g_d(\boldsymbol{f}_t)$ where $g_d(.)$ is the nonlinear mapping implemented by the ANN component of the revised model. The deep transformation layers consist of multiple fully-connected (dense) hidden layers where every node on the previous layer is connected to every node in the next layer. To make this mathematically explicit, defining a_i^l as the input for the i-th node on the l-th layer and o_i^l as the corresponding output, then o_i^l is given by:

$$a_i^l = < \boldsymbol{w}_i^l, \boldsymbol{o}^{l-1} > + b_i^l$$

$$\rightarrow o_i^l = \frac{1}{1 + e^{-a_i^l}}$$

The $\boldsymbol{w}_i^l$ and b_i^l refer to the set of weights and bias respectively that connect to the i-th input node on the l-th layer and $\boldsymbol{o}^{l-1}$ refers to all of the outputs of the nodes on the previous $(l-1)$-th layer. Observe that the sigmoid activation function is used to compute each o_i^l but this can be easily swapped out for an alternative such as hyperbolic tangent or ReLU. The complete set of weights and biases $\{\boldsymbol{w}_i^l, b_i^l\}$ for every layer l and node i in the deep representation part are trainable and hence must also be considered as part of the overall optimisation problem since they directly affect U_T. This is achieved by requiring maximisation of U_T with respect to the nonlinear mapping $g_d(.)$ in addition to the set of trading parameters $\Theta = \{\boldsymbol{w}, b, u\}$.

Therefore, the modified optimisation problem along with the modified policy δ_t for executing trading decisions is now given by:

$$\max_{\{\Theta, g_d(.)\}} U_T\{R_1, \ldots, R_T|\Theta\} = \max_{\Theta}\{\textstyle\sum_{t=1}^{T} R_t \,|\Theta\}$$

$$s.t.\ R_t = \delta_{t-1} z_t - c|\delta_t - \delta_{t-1}|$$

$$\delta_t = \tanh(< \boldsymbol{w}, \boldsymbol{F}_t > + b + u\delta_{t-1})$$

$$\boldsymbol{F}_t = g_d(\boldsymbol{f}_t)$$

$$\Theta = \{\boldsymbol{w}, b, u\}$$

The revised model is termed as the Deep Recurrent Neural Network (DRNN) by the authors and it is indeed equivalent to a literal deep RNN. It can be implemented and visualised as a complex neural network with built-in ANN and RNN components. An illustration is provided in figure 11 below. As before, BPTT can used for parameter training again subject to increased possibility of vanishing or exploding gradients occurring (see Section 2.4 end explanation).

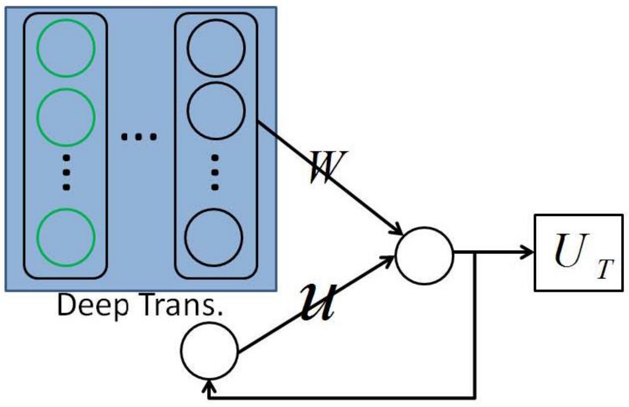


Figure 11: The DRNN model for automated financial assets trading (Deng et al., 2017).

2.6 Fuzzy Learning to Reduce Data Uncertainty

The final revision Deng et al. make to arrive at the FDRNN model is to introduce fuzzy learning concepts to reduce the underlying uncertainty associated with each $\boldsymbol{f}_t$ that is fed as input. They argue that the sequence of prices $p_1, p_2, \dots, p_{t-1}, p_t, p_{t+1}, \dots, p_{T-1}, p_T$ of any financial asset arriving from the exchange centre naturally contain a high level of unpredictable uncertainty that can essentially be attributed to the price discovery process. The price discovery process, which is what determines market prices at which financial assets are bought and sold, can evidently be affected by many unpredictable and uncertain factors such as level of market information available, global macroeconomic state, market participants' attitudes to risk (e.g. risk-neutral vs. risk-seeking vs. risk-averse), etc.

Essentially, the notion of fuzzy learning is applied to this context by assigning fuzzy values $\in (0,1)$ (as opposed to binary 0/1) to the inputs of the model rather than directly using the raw data arriving from the exchange centre (Deng et al., 2017). To do this, the raw data must be compared against a number of rough fuzzy sets before deriving the corresponding fuzzy membership degrees. The authors establish the rough fuzzy sets for the problem of algorithmic trading according to the different types of basic price movements that are possible – decreasing, increasing and not trending. Notice that this conveniently matches up with the trade decisions available in the policy δ_t at each time t. Now that the rough fuzzy sets have been built, the corresponding fuzzy membership degrees can be learnt directly through raw data and this process is described in detail in Section 3.4.1 later on.

Using fuzzy learning ideas in neural networks naturally translates to the concept of fuzzy neural networks. Given the DRNN model from Section 2.5, the logical way to create a fuzzified representation of the input feature vector $\boldsymbol{f}_t$ is to place a "fuzzy layer" immediately after the input layer. Using the contributions of Lin et al., in this fuzzy layer, each element of $\boldsymbol{f}_t = [z_{t-m+1}, \dots, z_t]$ is assigned to k different fuzzy degrees which is set to 3 in practice to incorporate the decreasing, increasing and not trending conditions. Using the same mathematical notation from Section 2.5 to refer to input and output nodes and layers, the i-th fuzzy membership function $v_i(.) \colon \mathbb{R} \to (0,1]$ maps the i-th input as a fuzzy degree:

$$o_i^2 = v_i(a_i^2) = e^{-\frac{(a_i^2 - m_i)^2}{\sigma_i^2}} \quad \forall i$$

The "2" superscript on o_i^2 and a_i^2 does not imply squaring but rather belonging to the second layer since the fuzzy layer gets placed immediately after the initial input layer. There are two key subtleties to observe here. Firstly, the Gaussian membership function with mean m and variance σ^2 is used in computing each o_i^2 instead of an activation function like the sigmoid that is used for the deep representation part. Secondly, given that k different fuzzy degrees are assigned to each of the m elements of $\boldsymbol{f}_t \in \mathbb{R}^m$, the fuzzy layer then naturally has a total of $m \times k$ fuzzy nodes. Thus, the range of i is given by $i = 1, \dots, m \times k$. Since the choice of m_i and σ_i^2 in each fuzzy node has a direct impact on U_T, once again, they must also be considered as part of the overall optimisation problem because they are trainable. This is achieved by requiring maximisation of U_T with respect to the fuzzy membership functions $v(.)$ In addition to the nonlinear mapping $g_d(.)$ and the set of trading parameters $\Theta = \{\boldsymbol{w}, b, u\}$.

Thus, the final revised optimisation problem is given by:

$$\max_{\{\Theta, g_d(.), v(.)\}} U_T\{R_1, \dots, R_T|\Theta\} = \max_{\Theta}\{\textstyle\sum_{t=1}^{T} R_t\,|\Theta\}$$

$$s.t.\ R_t = \delta_{t-1} z_t - c|\delta_t - \delta_{t-1}|$$

$$\delta_t = \tanh(<\boldsymbol{w}, \boldsymbol{F}_t> + b + u\delta_{t-1})$$

$$\boldsymbol{F}_t = g_d\big(v(\boldsymbol{f}_t)\big)$$

$$\Theta = \{\boldsymbol{w}, b, u\}$$

This is the final Fuzzy Deep Recurrent Neural Network (FDRNN) model whose derivation has been sought after since the commencement of this thesis. It can be implemented and visualised as a complex neural network architecture that includes an initial fuzzy layer in addition to built-in ANN and RNN components. An illustration of the rolled and unrolled variants is provided in Figures 12 and 13 below. As before, BPTT can used for parameter training again subject to increased possibility of vanishing or exploding gradients occurring (see Section 2.4 end explanation).

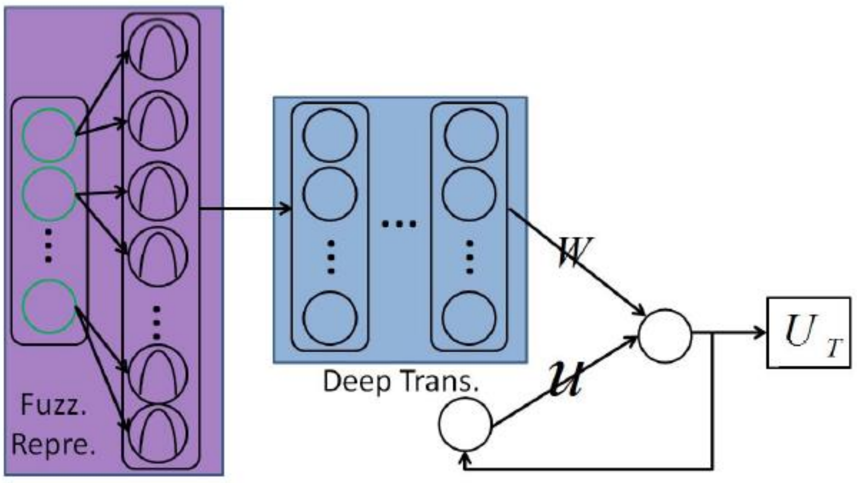


Figure 12: The rolled FDRNN model for automated financial assets trading with $k = 2$ (Deng et al., 2017).

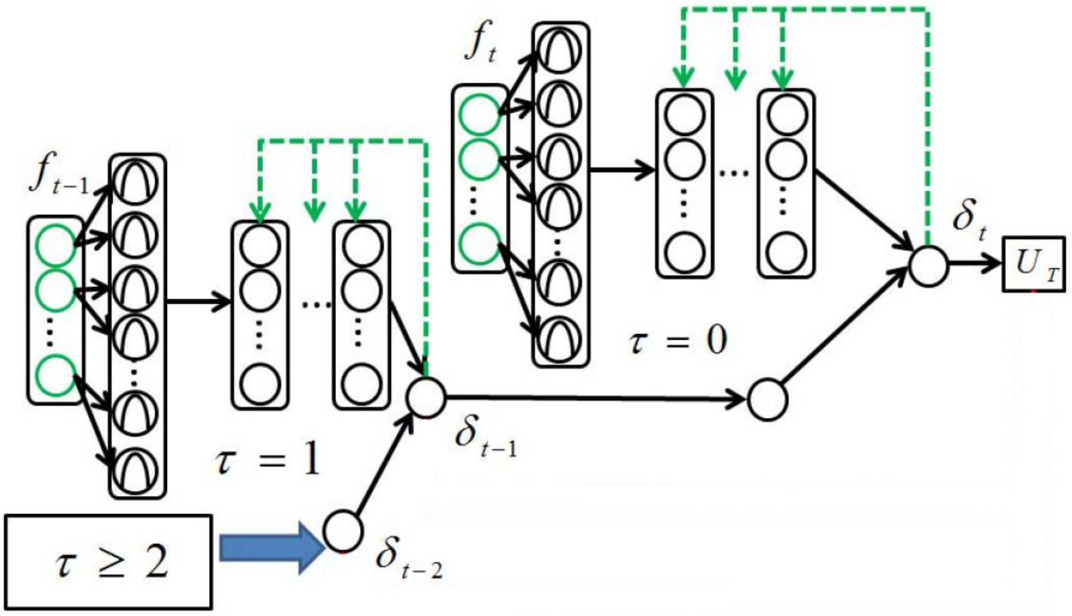


Figure 13: The unrolled FDRNN model for automated financial assets trading with $k = 2$ (Deng et al., 2017).

Chapter 3
Implementation Details

3.1 Programming Language and Hardware

The popular programming language Python was chosen to implement the FDRNN model by Deng et al in code.

There are several reasons why Python was chosen over other languages such as C or C++. Python is a high-level programming language that enables developing programs using human natural language elements making it much more human readable, understandable as well as easier to program (and debug) in. In addition to automated memory management, it is also an object-oriented language that allows designing software around "objects" in addition to pure functions and logic. Dynamic typing is supported which avoids the need to declare every variable and function type (integer, float, etc.) which is compulsory in C, C++ or Java. There is extensive library and community support available that can aid developers in programming a wide range of applications.

The largest disadvantage to using Python in this context is certainly its program execution speed. In contrast to a compiled language such as C or C++ that uses a compiler to compile source code directly into machine code prior to program execution, Python is an interpreted language that utilises an interpreter to convert source code into bytecode rather than machine code. The corresponding bytecode instructions then proceed to be executed on a virtual machine. The benefit is operating system independency at the cost of execution speed. Logically, this is because the bytecode is not the "native language" understood by computers and hence additional work must be done by the interpreter to retranslate it into another form that can be understood by the machine.

All the incremental programs used in implementing the FDRNN model were developed on a 16 GB RAM, Intel i7 11700 CPU and NVIDIA RTX 3060 Ti GPU (8 GB) personal machine. As mentioned above, the FDRNN becomes a very deep neural network once the RNN component is unrolled for parameter training in BPTT. Hence, using a CPU alone can result in very slow training times for the model. GPUs, on average, possess many more cores than CPUs (1000-10000 compared to 8-32 usually) and are known to better handle multiple computations simultaneously by performing them in parallel. Hence, the FDRNN training and testing was parallelised using the above NVIDIA GPU – Section 3.4.2 provides further detail on this. Furthermore, one of UQ's several high-performance computing clusters was used to perform the experimentation plan outlined in Section 3.5 below in obtain experimentation results faster. Additional details of exactly how the cluster was used is provided in Section 3.5.

3.2 Dataset Retrieval and Preprocessing

When conducting experimentation, the authors used the raw price changes (i.e. returns) of the last 45 mins in addition to the previous 3-hours, 5-hours, 1-day, 3-days and 10-days momentum changes as input into each feature vector $\boldsymbol{f}_t$ for the FDRNN model. Whilst they do not make it clear what momentum implies, it is assumed to imply price change/return but for a much longer time period. To make this mathematically explicit for the model, m_{t-k} is used to denote the k-minutes momentum change. Then $m_{t-k} = p_t - p_{t-k}$. For example, the 3-hours and 10-days momentum changes are given by:

$$m_{t-3\times60} = m_{t-180} = p_t - p_{t-180}$$

$$m_{t-10\times60\times24} = m_{t-14400} = p_t - p_{t-14400}$$

On a side note, since $m = 50$ here (<u>not to be confused with</u> m_t), each $\boldsymbol{f}_t$ is then $\in \mathbb{R}^{50}$ and combined with the number of fuzzy degrees $k = 3$ per dimension of $\boldsymbol{f}_t$, the fuzzy layer that follows will contain $m \times k = 50 \times 3 = 150$ total fuzzy nodes.

However, prior to moving straight to model training, data is required to be able to train the FDRNN. The writers' decision to use minutely financial asset data was followed. Generally, the higher the frequency of financial data, the more expensive accessing it becomes since a finer overview of true market dynamics is gained. High frequency data can easily be converted to lower frequency via some reduction statistic such as averaging but the reverse cannot work because that would entail artificially generating data to "fill in the gaps".

GitHub user Max Williams provides a large collection of financial datasets in minutely price frequency amassed from a collection of financial data vendors, brokers and exchanges through his $financial\text{-}data$ Git repository (Williams, 2020). He additionally provides Python installation instructions to install his custom Python library $pyfinancialdata$ using Python's PIP software installation manager. From the available datasets, all minutely financial data on futures for the S&P 500, NIKKEI 225 and DAX 30 stock market indices for the year of 2017 were retrieved after importing $pyfinancialdata$. These index futures track some of the largest and most recognisable stocks in the United States, Japan and Germany respectively. These are the datasets on which the FDRNN model will eventually be trained and tested on. Complications relating to trading futures in real life such as margin, leverage, taxes, sufficient liquidity, etc. are digressed from since the FDRNN model does not account for these.

To help preprocess these datasets, the popular $numpy, matplotlib$ and $pandas$ Python libraries were employed. The largest flaw observed in the quality of these datasets was missing close prices p_t for some t. There are several techniques available to deal with missing time series data ranging from simplistic such as forward filling and linearly-spacing missing data between samples all the way to sophisticated such as Bayesian methods and imputation via ML algorithms. Due to time constraints, the most simplistic technique – forward filling was applied acknowledging the fact that a portion of the missing data in every dataset would be attributed to daily, weekend and public holiday market close hours. Finally, the last step is to create the feature vector $\boldsymbol{f}_t$ for each time t. As mentioned in the first paragraph in this section, the raw price changes (i.e. returns) of the last 45 mins in addition to the previous 3-hours, 5-hours, 1-day, 3-days and 10-days momentum changes form the dimensions of each $\boldsymbol{f}_t \in \mathbb{R}^{50}$. Mathematically, this is equivalent to:

$$\boldsymbol{f}_t = [z_t, z_{t-1}, \dots, z_{t-45+2}, z_{t-45+1}, m_{t-3\times60}, m_{t-5\times60}, m_{t-1\times60\times24}, m_{t-5\times60\times24}, m_{t-10\times60\times24}]$$

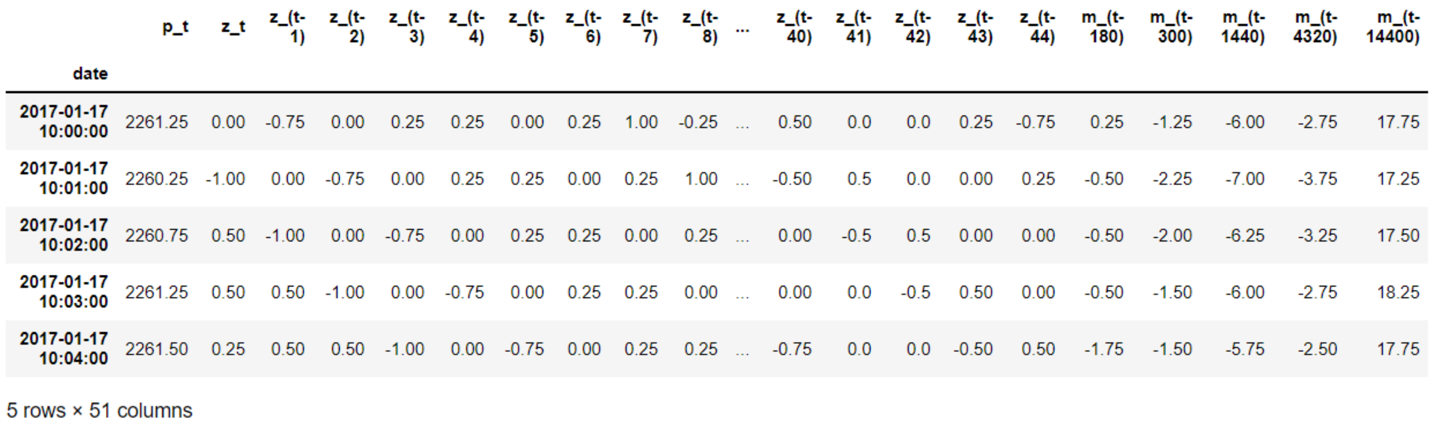
$$\rightarrow \boldsymbol{f}_t = [z_t, z_{t-1}, \dots, z_{t-43}, z_{t-44}, m_{t-180}, m_{t-300}, m_{t-1440}, m_{t-7200}, m_{t-14400}]$$

In the three datasets, each sample (row) then contains a single $\boldsymbol{f}_t$ for the corresponding time t. Note that at time $t = 1$, the first row for each of the datasets must be removed because the first dimension of $\boldsymbol{f}_t$ given by $z_t = z_1 = p_1 - p_0$ is undefined since p_0 is undefine (time only starts from $t = 1$). Similarly, at time $t = 2$, the second row for each of the datasets must also be removed because the second dimension of $\boldsymbol{f}_t$ given by $z_{t-1} = z_1$ is undefined as shown above. Continuing this process, a total of 14401 original rows are deleted from the beginning of every dataset to ensure well-defined $\boldsymbol{f}_t$ at each time t. This is because the smallest initial value possible for $m_{t-14400}$ is when:

$$t = 14401$$

$$\rightarrow m_{t-14400} = p_{14401} - p_{14401-14400} = p_{14401} - p_1$$

A tabular illustration of one of the complete preprocessed datasets with p_t in column 1 and the corresponding $\boldsymbol{f}_t$ in columns 2-51 is provided in Figure 14 below.



date	p_t	z_t	z_(t-1)	z_(t-2)	z_(t-3)	z_(t-4)	z_(t-5)	z_(t-6)	z_(t-7)	z_(t-8)	...	z_(t-40)	z_(t-41)	z_(t-42)	z_(t-43)	z_(t-44)	m_(t-180)	m_(t-300)	m_(t-1440)	m_(t-4320)	m_(t-14400)
2017-01-17 10:00:00	2261.25	0.00	-0.75	0.00	0.25	0.25	0.00	0.25	1.00	-0.25	...	0.50	0.0	0.0	0.25	-0.75	0.25	-1.25	-6.00	-2.75	17.75
2017-01-17 10:01:00	2260.25	-1.00	0.00	-0.75	0.00	0.25	0.25	0.00	0.25	1.00	...	-0.50	0.5	0.0	0.00	0.25	-0.50	-2.25	-7.00	-3.75	17.25
2017-01-17 10:02:00	2260.75	0.50	-1.00	0.00	-0.75	0.00	0.25	0.25	0.00	0.25	...	0.00	-0.5	0.5	0.00	0.00	-0.50	-2.00	-6.25	-3.25	17.50
2017-01-17 10:03:00	2261.25	0.50	0.50	-1.00	0.00	-0.75	0.00	0.25	0.25	0.00	...	0.00	0.0	-0.5	0.50	0.00	-0.50	-1.50	-6.00	-2.75	18.25
2017-01-17 10:04:00	2261.50	0.25	0.50	0.50	-1.00	0.00	-0.75	0.00	0.25	0.25	...	-0.75	0.0	0.0	-0.50	0.50	-1.75	-1.50	-5.75	-2.50	17.75

5 rows × 51 columns

Figure 14: Fully preprocessed $SPXUSD\_2017$ dataset – minutely data for S&P 500 futures in 2017 (Williams, 2020).

3.3 Dataset Visualisation

Some of the Exploratory Data Analysis (EDA) performed on each of the three selected datasets is showcased in Figure 15 on the next page. Python libraries such as $matplotlib$, $pandas$ and $jupyter\ notebook$ were used to produce these diagrams. In order to better describe the different aspects observed during EDA, time series concepts are used. Mean-stationarity refers to a time series varying around some fixed mean over the entire given time period (Wei et al., 1999). Variance-stationarity refers to a time series having constant variance – the "jumps" in the time series are always of fixed size (Wei et al., 1999).

Commencing with p_t along the first row of graphs in Figure 15, the price time series for the S&P 500 futures appears to be stationary in both mean and variance and is trending upwards over time. However, the NIKKEI 225 and DAX 30 futures both appear to have non-stationary mean and variance but also generally trend upwards over time. The DAX 30 appears to be the most volatile index of the three in 2017.

Next, it is first observed that z_t, in time-series nomenclature, is equivalent to the first difference of p_t since it is given by $p_t - p_{t-1}$. Traditionally, first-differencing is performed to induce stationarity in the mean in the transformed time-series. Indeed, this is evident for z_t along the second row of graphs in Figure 15 since these price change series are evidently varying around a mean of 0 over the entire time period. However, there are several outliers in these series made evident by rare extreme increases or decreases in z_t. A plausible explanation is market participants' adverse reaction to positive/negative news/events from firms that hold sizeable proportions in these indices.

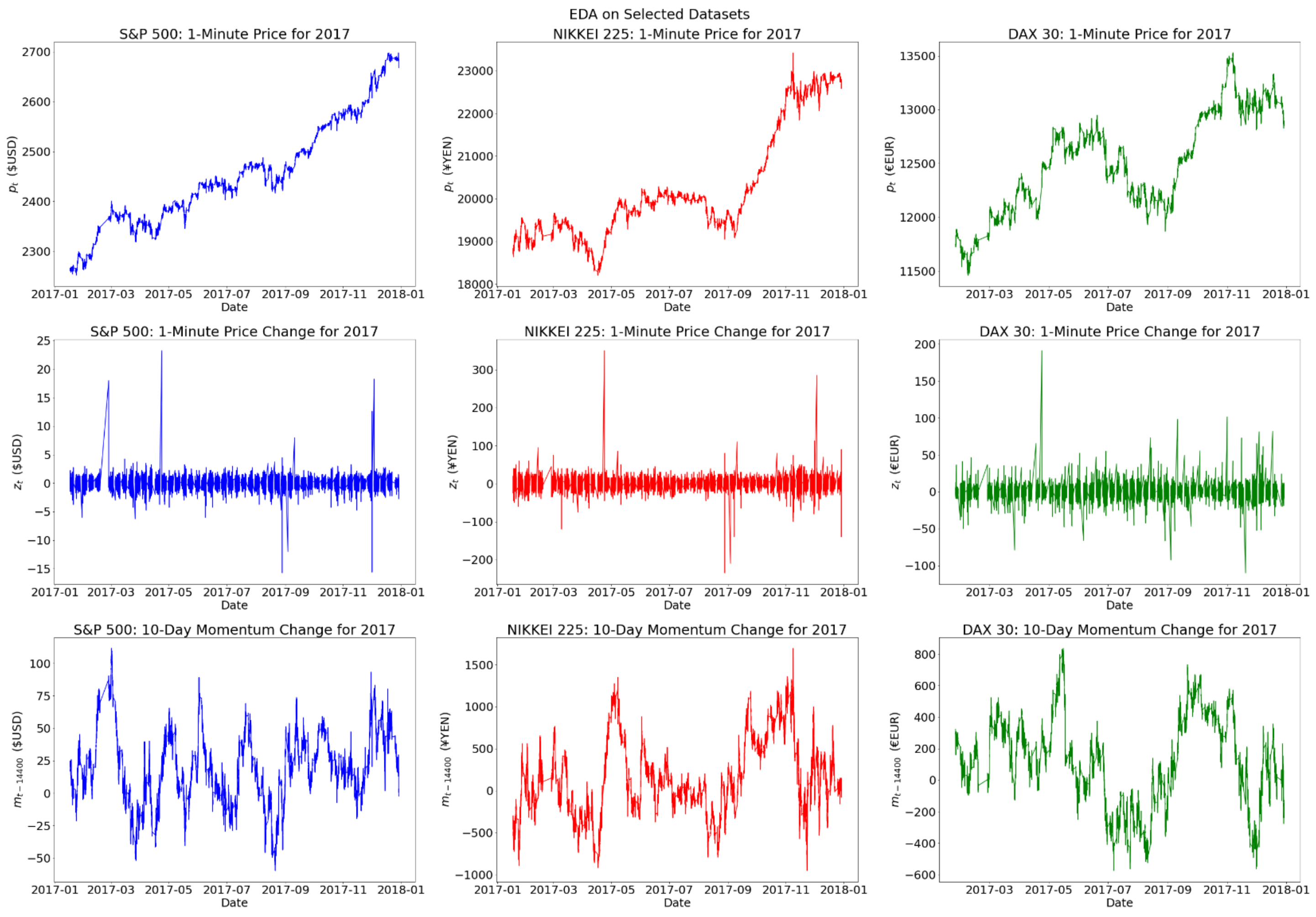


Figure 15: Plotting p_t, z_t and $m_{t-14400}$ for the S&P 500, NIKKEI 225 and DAX 30 2017 datasets.

Finally, to compare against the 1-minute price change, the 10-day momentum change $m_{t-14400}$ is plotted along the final row of graphs in Figure 15. Three key observations are noticed, the first being that whilst the mean-stationarity property is preserved in $m_{t-14400}$ across all three datasets, the amplitude of these changes fluctuating around the fixed means is much larger. This is because over a 10-day period, there is much larger room for price movement in contrast to just over a 1-minute period. Next, although subjective, it can be visually deduced that the fixed means aren't equivalent to 0 but rather positive values larger than 0. This implies that given the momentum changes, the price series are, on average, strictly rising over the time period. Theoretically, this necessarily must be true because p_t is seen to be trending upwards across all three datasets in 2017. Lastly, taking the difference in prices over a longer time period results in highly distinct patterns to emerge unlike z_t along the second row of graphs in Figure 15.

3.4 Training and Testing Algorithms for the FDRNN Model

3.4.1 Pseudocode

Now that all the desired dataset preprocessing and visualisation has been performed, it is time to discuss training and testing the FDRNN model on each of these datasets.

First, it should be noted that Deng et al. dedicate a portion of their paper to cover the various system initialisations utilised to initialise the three learning parts – parameters belonging to the fuzzy representation part contained in $v(.)$, the deep representation part contained in $g_d(.)$ and the direct reinforcement learning part contained in Θ (Deng et al., 2017). Initialisation for first part is straightforward – the only parameters to consider here are the fuzzy centres m_i and widths σ_i^2 belonging to each of the fuzzy nodes, $i = 1, \dots, m \times k$ as from before. The authors use k-means clustering to classify each of the feature vectors $\boldsymbol{f}_t$ used as part of the training set into $k = 3$ classes. This is to represent the increasing, decreasing and no-trend conditions as well as the fact that each dimension of $\boldsymbol{f}_t$ is to be connected to three fuzzy membership functions denoting these conditions. The latter is performed by calculating the mean and variance of each dimension of $\boldsymbol{f}_t$ under each assigned cluster to initialise the corresponding m_i and σ_i^2. In other words, the fuzzy nodes appearing in Figure 12 from Section 2.6 are initialised exactly as is, one-by-one from top to bottom. Next, the initialisation method used for the second part is to use an Autoencoder to optimally reconstruct each fuzzified representation corresponding to each $\boldsymbol{f}_t$ on a separate virtual layer positioned after the final hidden layer. The initialisation used for the third part is to feed in the final deep representations $\boldsymbol{F}_t$ obtained from each $\boldsymbol{f}_t$ in the training set directly into the RNN component of the FDRNN. It is of personal belief that the intuition behind these last two initialisation methods was not explained well by the authors and thus, programming these into actual code became an extremely challenging task. Due to time constraints, both initialisations needed to be left out to complete the project in time. Accordingly, the pseudocode to be provided later does not acknowledge the last two initialisation methods. Instead, initialisation for all of the trainable parameters is performed for both parts using the random uniform distribution $U\left(-\sqrt{l}, \sqrt{l}\right)$ where $l = \frac{1}{input\_features} > 0$. Here, $input\_features$ refers to the incoming input features applicable to the trainable parameters associated with single nodes (i.e. varies for nodes belonging to different layers). This specific choice of l is part of the natural initialisation method used by the chosen deep learning library for the Python implementation (PyTorch, 2021).

As explained at the end of Section 2.3, one must expect the issues of vanishing or exploding gradients to arise whenever BPTT is used to train a suitable RNN. In order to better handle these problems, the authors propose a more practical solution termed as Task-Aware BPTT which aims to bring the gradient information directly from the learning task to each layer containing trainable parameters (excluding fuzzy layers) in the unrolled FDRNN (see Figure 13). Whilst their explanation of this section is quite intuitive, it was also deemed too difficult to practically implement into Python code given the time constraints since the selected deep learning library (mentioned in Section 3.4.2) does not feature a straightforward method to do this, at least up to personal knowledge. It seems as though the raw source code of the training algorithm used in the chosen deep learning library would need to be modified to obtain a practical working implementation of Task-Aware BPTT. Although regular BPTT is used for training the FDRNN model, the reader can still view their methodology in Section IV B of the authors' paper (Deng et al., 2017).

Following this, the pseudocode used to both train and test the FDRNN model is presented as Algorithm 1 in Table 2 below. Given the extensive overview provided over this section and Chapter 2 of the thesis, the reader should be able to understand the purpose and relevance of each line. Thus, detailed line-by-line explanation of Algorithm 1 is not provided to avoid heavy repetition of concepts that were previously discussed in great detail.

```
Algorithm 1: Training and Testing of the FDRNN Model
1:  input: dataset containing feature vectors $\boldsymbol{f}_t$, training parameters and hyperparameters – $c, num\_train, num\_test, m, k, seq\_length, fc\_hidden\_size, num\_fc, \eta, l$, etc.
2:  initialisation: form $train$ and $test$ sets based on $num\_train$ and $num\_test$ samples, parameters belonging to fuzzy membership functions $v(.)$ via fuzzy clustering, deep representation and reinforcement learning parameters $\sim^{iid} U(-\sqrt{l}, \sqrt{l})$
3:  // Training the FDRNN model
4:  repeat
5:      update learning rate $\eta$ if using a (exponentially) decaying learning rate method; weight decay (L2 regularisation) $\gamma$ presented in input;
6:      for $t = seq\_length, \dots, T_{train}$ do
7:          obtain sequence $\boldsymbol{f}_t, \boldsymbol{f}_{t-1}, \dots, \boldsymbol{f}_{t-seq\_length+1}$ of length $seq\_length$ from $train$ set;
8:          perform step 1) of BPTT: unroll the RNN component at time $t$ into $seq\_length$ stacks;
9:          perform forward pass through FDRNN;
10:         compute $loss_t$ as $-1 \times$ the subset of $U_t$ attributed to the sequence $\boldsymbol{f}_t, \boldsymbol{f}_{t-1}, \dots, \boldsymbol{f}_{t-seq\_length+1}$;
11:         perform step 2) of BPTT: backpropagate the resulting gradients through the unrolled network;
12:         compute $\nabla(loss_t)_\Theta$ by averaging its gradient values on all $seq\_length$ stacks;
13:         // $\eta$ = selected optimiser's learning rate, $\lambda$ = weight decay (L2 regularisation penalty)
14:         update $\Theta_t = \Theta_{t-1} - \eta \frac{\nabla(loss_t)_\Theta}{||\nabla(loss_t)_\Theta||} - \eta\lambda\Theta_{t-1}$;
15:     end
16: until convergence of loss function;
17: // Testing the FDRNN model
18: for $t = T_{train} + 1, \dots, T_{test}$ do
19:     obtain sequence $\boldsymbol{f}_t, \boldsymbol{f}_{t-1}, \dots, \boldsymbol{f}_{t-seq\_length+1}$ of length $seq\_length$ from $test$ set;
20:     perform step 1) of BPTT: unroll the RNN component at time $t$ into $seq\_length$ stacks;
21:     perform forward pass through FDRNN;
22:     compute the subset of $U_t$ attributed to the sequence $\boldsymbol{f}_t, \boldsymbol{f}_{t-1}, \dots, \boldsymbol{f}_{t-seq\_length+1}$;
23: end
24: // Plot the training and testing results
25: plot training loss, $U_t$ from $t = seq\_length, \dots, T_{train}$, $U_t$ from $t = T_{train} + 1, \dots, T_{test}$;
```

Table 2: Pseudocode for the training and testing of the FDRNN model, modified from the author's paper to better reflect portions of their work replicated in this thesis (Deng et al., 2017).

3.4.2 Program Design

The main Python libraries used to implement Algorithm 1 from Table 2 are $matplolib, numpy, pandas, sklearn$ and $torch$. The last import represents the chosen deep learning library – PyTorch is a popular open source machine learning framework developed by Facebook's AI Research lab aimed at reducing the gap between research and deployment to production. The other two popular choices are Keras and TensorFlow – PyTorch was selected simply out of personal preference.

Now the process of translating from pseudocode to Python code and the program design choices made shall be discussed extensively. Initially, the preprocessed dataset from Section 3.2 is loaded into a $DataFrame$ object using the $pandas$ library. This is followed by the setting up of several important training parameters and hyperparameters such as $c, num\_train, num\_test, m, k, seq\_length, fc\_hidden\_size, num\_fc, \eta, l$, etc. Initialising these parameters early on and in one big block of code raises two benefits. The first being that since they are located together and not scattered all over in the remaining code, it is easy to find any desired parameter of interest quickly. The second being that if the value of a parameter as a variable must be changed, it can be done directly here affecting the usage of that variable elsewhere in the remaining code i.e. no other changes need to be made. Next, $pandas$ was used once again to create the $train$ and $test$ sets formed by splitting up the original dataset into num_train and num_test samples respectively. Note that $num\_train + num\_test = length(dataset) \rightarrow num\_test = length(dataset) - num\_train$. The $KMeans$ method from $sklearn.cluster$ was combined with $numpy$ to initialise all the fuzzy means and widths belonging to the fuzzy layer via k-means clustering exactly as described in the second paragraph of Section 3.4.1 above.

Before lines 4-16 of Algorithm 1 can be translated, a complete Pythonic representation of the FDRNN model must be created. This must include all the various nodes, layers and trainable parameters that form the fuzzy representation, deep representation and reinforcement learning parts. PyTorch encourages objected-oriented programming (OOP) by focussing on the use of Python classes to represent and contain complex neural network structures. While the cost of this is increased code complexity, the benefits brought about are beyond comparison. A class implementation for the FDRNN enables code organisation, encapsulation, inheritance, reusability and even a notion of state. Accordingly, $torch$ was used to create two classes, both inheriting from the $torch.nn.Module$ class (PyTorch, 2021). The first class was written to represent the concept of a fuzzy layer that takes as input a sequence of feature vectors $\boldsymbol{f}_t, \boldsymbol{f}_{t-1}, \dots, \boldsymbol{f}_{t-seq\_length+1}$ (length depending on seq_length parameter) and performs a forward pass by generating fuzzified representations of them using the initialised fuzzy means m_i and widths $\sigma_i^2, i = 1, \dots, m \times k$. On a side note, at the heart of inputs and outputs in a PyTorch model lies $torch.tensor$ which is a multi-dimensional matrix container object storing elements belonging to a single data-type (PyTorch, 2021). They are programmed specifically for deep learning and hence $\boldsymbol{f}_t, \boldsymbol{f}_{t-1}, \dots, \boldsymbol{f}_{t-seq\_length+1}$ in practice, are fed using a single $tensor$. The second class created using $torch$ was written to represent the FDRNN model that upon initialisation also initialises the fuzzy layer class. Note that the fuzzy parameters from k-means clustering are passed on to be stored within the fuzzy layer class. The FDRNN class also takes as input a sequence $\boldsymbol{f}_t, \boldsymbol{f}_{t-1}, \dots, \boldsymbol{f}_{t-seq\_length+1}$ and performs a forward pass through multiple fuzzy layers (using the class's forward pass method), hidden layers belonging to ANN components and hidden layers belonging to the RNN component. Note that there are multiple fuzzy layers and ANN components due to unrolling of the RNN component in the FDRNN model, however only a single class is needed to represent each type of layer (fuzzy layer, fully-connected hidden layer in ANN). This is because only the input and output shapes (dimensions) need to be managed – a class for a layer is used to simply contain the internals behind that layer. Furthermore, PyTorch also supports using batches for training, the purpose being to updating model parameters more "gently" by requiring some fixed number of training samples to be processed first. If only a single sequence $\boldsymbol{f}_t, \boldsymbol{f}_{t-1}, \dots, \boldsymbol{f}_{t-seq\_length+1}$ is fed in a time, the training behaviour can be very erratic. On the contrary, a large number of sequences of results in smoother training but higher computational memory usage. Naturally, this forms a parameter labelled as $batch\_size$ that must also be provided as **input** in Algorithm 1. In order to generate batches from the $train$ and $test$ sets, a custom function $create\_batch\_generator()$ was written that creates a batch $generator$ object which can be iterated to over to obtain training and testing batches. It also provides a sequence of z_t (recall $z_t = p_t - p_{t-1}$) corresponding to all $\boldsymbol{f}_t$ in the final column of a batch along with the batch. A visual example of a single batch of shape $(batch\_size, seq\_length, m) = (5,3,50)$ and corresponding sequence of z_t from $t = 3$ onwards that could be used for training is given below:

$$\begin{bmatrix} f_1 & f_2 & f_3 \\ f_2 & f_3 & f_4 \\ f_3 & f_4 & f_5 \\ f_4 & f_5 & f_6 \\ f_5 & f_6 & f_7 \end{bmatrix}, \begin{bmatrix} z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix}$$

The rest of the program design follows naturally from the pseudocode. Line 8 did not need to be explicitly programmed since PyTorch handles unrolling of RNNs internally and line 9 is explained in detail in the above paragraph. Line 10 was performed using a function to calculate the resulting loss after performing a forward pass on a single batch. In Python, the $loss_t$ at time t is computed as:

$$loss_t = -\left(\sum_{s=seq\_length}^{t} R_s \right) + \frac{\lambda}{2} ||\Theta_t||^2$$

$$R_s = \delta_{s-1} z_s - c|\delta_s - \delta_{s-1}|$$

$$\delta_s = \tanh(< \boldsymbol{w}, \boldsymbol{F}_s > + b + u\delta_{s-1})$$

Note that the last expression in $loss_t$ is from L2 regularisation: if $\lambda = 0$, the entire term vanishes (no L2 penalty to loss function). Additionally, δ_s is the resulting output of the FDRNN from a single f_s provided as input. Consequently, if multiple sequences of $\boldsymbol{f}_t, \boldsymbol{f}_{t-1}, \dots$ are provided in a batch, the output must also contain the $\delta_t, \delta_{t-1}, \dots$ corresponding to those sequences. PyTorch handles this by ensuring that if the input is of shape $(batch\_size, seq\_length, m)$, then the resulting output must also be of shape $(batch\_size, seq\_length, *)$ where $*$ is set to 1 in practice to reflect each δ being a scalar, not vector (PyTorch, 2021). This explains why $create\_batch\_generator()$ provides a sequence of z values – in order to compute $loss_t$, R_s must be computed for $s = seq\_length, \dots, t$ and in each R_s, δ_{s-1}, δ_s and z_s are required. The visual example from above is continued by displaying what the output from a single forward pass looks like, noting an output shape of $(batch\_size, seq\_length, 1) = (5,3,1)$:

$$\begin{bmatrix} \delta_1 & \delta_2 & \delta_3 \\ \delta_2 & \delta_3 & \delta_4 \\ \delta_3 & \delta_4 & \delta_5 \\ \delta_4 & \delta_5 & \delta_6 \\ \delta_5 & \delta_6 & \delta_7 \end{bmatrix}$$

Hence, the second and last columns of the matrix containing the δ_s and the vector containing the z_s can be directly used in this form to compute each R_s and then finally compute $loss_t$. Lines 11 and 12 were handled very easily in $torch$ using $loss_t.backward()$ and the translation of line 14 was also straightforward using the $step()$ method on a chosen optimiser for the model. In practice, the Stochastic Gradient Descent optimiser was used for training the FDRNN on all three datasets. The specific hyperparameter choices such as η and λ are displayed in Table 3 in Section 3.5.

The translation of the model testing portion of the pseudocode is not discussed since it is highly similar to the above. Lastly, the $matplotlib$ library was used to plot the model training loss in addition to the accumulated rewards U_t over the training and test periods. These graphs are specifically created in order to visually evaluate model performance.

The final step was transferring any of the CPU (host) memory storing tensors, weights, biases, parameters, hyperparameters, classes and more onto GPU (device) memory. As explained earlier, GPUs, on average, possess many more cores than CPUs and are known to better handle multiple computations simultaneously by performing them in parallel. This parallelisation of computation can often result in respectable runtime speedups. PyTorch supports the use of capable NVIDIA GPUs for deep learning via NVIDIA CUDA. CUDA is a parallel programming platform and model developed by NVIDIA Corporation (NVIDIA, 2021). In the Python code, the only required steps were to ensure every $tensor$ (e.g. input batch) was converted onto the device using $tensor.to(device)$ as well as directly initialising the class representing the entire FDRNN model onto the device using $fdrnn().cuda()$.

3.4.3 Verification Procedure

The most obvious initial sign the Python implementation is working, is execution without any runtime crashes. Unfortunately, there is no way to truly verify that the personal implementation of the FDRNN model is working *exactly as intended* in the pseudocode. Whilst it has been attempted to the best of personal ability to debug and ensure the code's logic matches up with the pseudocode's, there is no benchmark model to verify against. Even if there were, the stochasticity involved with neural networks and deep learning (e.g. trainable parameters' initialisation, rounding and precision of floating-point arithmetic, etc.) nullifies any real comparisons from being made.

It is highly unclear if the FDRNN gets trapped into an undesirable local minima during training, but it should be expected that this is the case. This is because of how complicated and non-convex the loss function surface can be, noting that it is the optimiser's role to descend this appropriately. Figure 16 below provides an example of one such loss surface that is associated with a complicated neural network architecture. A potential solution is to generate an Ensemble of FDRNNs. An ensemble in ML builds a prediction model by combining multiple identical "learners" (ML algorithms) trained on the same train set and tested on unseen data using some sort of "majority" vote e.g. most predicted class for classification or average prediction for regression problems (Hastie et al., 2009). In this case however, there is no majority vote as only the "best" model post training continues onto the testing set for performance evaluation. This is because it is expected that there will inevitably be some FDRNNs which make losses over the training period, hence they should not be used on the test set as they are highly unlikely to be profitable. As mentioned before, these FDRNN models may suffer from poor performance due to the nature of their loss surfaces.

Some straightforward signs of the model training inappropriately due to issues such as vanishing or exploding gradients are poor or erratic training loss, NaNs appearing in computed loss during training or even extremely large weights (applicable only to exploding gradients). During verification runs, none of these signs were experienced and thus, the next step was to move onto experimentation to generate the results presented subsequently in Chapter 4.

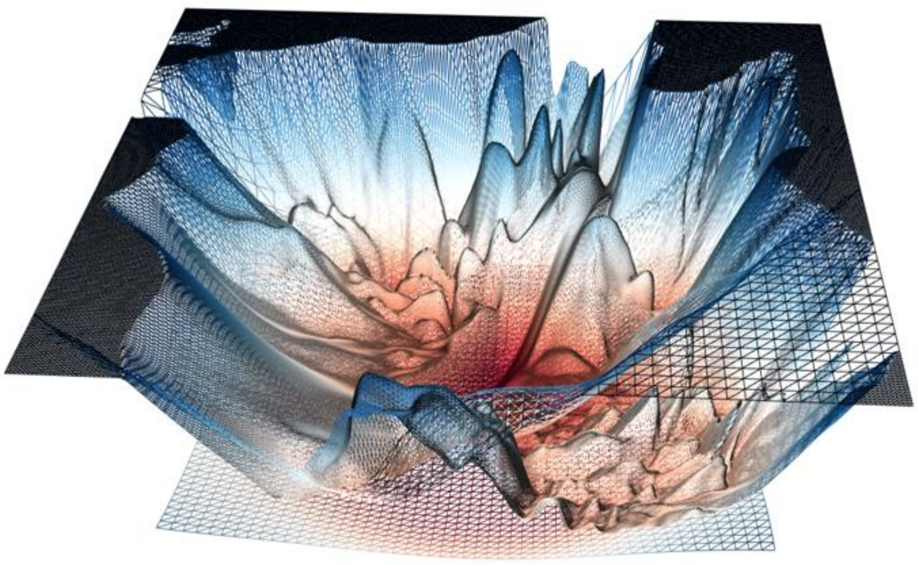


Figure 16: An example of a loss "landscape" of a complex neural network architecture (Mohan, 2019).

3.5 Experimentation Plan

Now that some initial runs on the Python implementation have been conducted, the experimentation plan is straightforward and is discussed briefly. Sections 3.2 and 3.3 retrieved, preprocessed and visualised three datasets to be used to train and test the FDRNN model – they are minutely financial data on futures for the S&P 500, NIKKEI 225 and DAX 30 stock market indices for the year of 2017.

Following from the concept of an ensemble of FDRNNs from the section above, the goal is then to train multiple FDRNNs but only select a single FDRNN for testing on each of the three datasets. In doing so, diagrams of the training loss and accumulated rewards over the training and testing periods must also be generated for model performance evaluation. The single FDRNN that is chosen for testing is based on which model in the ensemble had the highest total $U_{T_{train}}$ at the end of the training period i.e. highest accumulated reward over the training period based on the last training iteration.

The parameters and hyperparameters used in the experimentation setup to generate the results for Chapter 4 are displayed in Table 3 on the next page. Here, the difference between what constitutes as a parameter and a hyperparameter for the FDRNN is made explicit. Bear in mind that these values are shared across all three datasets. Additionally, most of the values selected follow from the choices made by Deng et al. in their paper (Deng et al., 2017). However, the remaining parameters and hyperparameters that were not discussed such as $batch\_size, seq\_length, num\_rec, \eta$ and λ were chosen based on trial and error during initial verification runs. Finally, the transaction cost $c = 15$ was chosen to be approximately 6 times higher than the norm for S&P 500 futures (CME Group, 2016). Hence, this can also be used for the NIKKEI 225 and DAX 30 futures as both are priced higher (due to valuation in different currencies, see Figure 15 from Section 3.3).

Parameters	
Description **Python variable**	**Value**
Number of FDRNNs to train in an Ensemble num_models	10
Transaction cost to use for training and testing c	15
Number of training samples num_train	$\frac{1}{3} \times length(dataset)$
Number of training iterations num_epochs	100
Dimension of each feature vector $\boldsymbol{f}_t$ (m) $input\_size$	50
Number of clusters in k-means clustering (fuzzy degrees) k	3
Hyperparameters	
Description **Python variable**	**Value**
Number of training samples in each batch $batch\_size$	32
Number of time steps to feed into FDRNN at a time seq_length	3
Input and output size to each hidden layer in the ANN component fc_hidden_size	128
Number of fully-connected hidden layers in the ANN component num_fc	4
Size of each output $\boldsymbol{F}_t$ after passing through fuzzy and deep representations $output\_size$	20
Number of features in a hidden state for the RNN component (shape of δ_t) rec_hidden_size	1
Number of (stacked) recurrent layers num_rec	1
Learning rate to be used during training (η) eta	0.00001
Weight decay (L2 regularisation penalty) to be used during training (λ) $lambda\_$	0.000001

Table 3: A list of the explicit values set for parameters and hyperparameters used in training an ensemble of FDRNNs on each chosen dataset.

3.6 Experimentation Methodology

To greatly reduce the total time required to carry out the experimentation plan in practice, in addition to performing training and testing on a capable NVIDIA GPU, a high-performance computing cluster was used to gain access to multiple NVIDIA CUDA capable GPUs. More specifically, the UQ School of Mathematics and Physics' Getafix cluster was utilised. The benefit achieved from moving the code onto the cluster is further parallelisation by simply executing $fdrnn.py$ for each dataset on a separate GPU device, in contrast to executing each run serially on a single GPU device and completing them one-by-one. Thus, in total, three GPUs were used to simultaneously train 10 FDRNNs and test the highest performing model for each dataset. The specific GPUs used for training were NVIDIA Tesla V100s (32 GB). Thus, the explicit setup for a single cluster job was 16 GB RAM, single CPU (type not recorded since not relevant) and one NVIDIA Tesla V100.

The total training and testing times for each dataset are recorded in Table 4 below. Note that the DAX 30 2017 dataset took much shorter to train 10 FDRNNs because there were fewer recorded samples here compared to the other datasets.

Dataset	**Training 10 FDRNNs**	**Testing best FDRNN**
S&P 500 / $\boldsymbol{SPXUSD\_2017}$	6.63 hours	50.46 seconds
NIKKEI 225 / $\boldsymbol{JPXJPY\_2017}$	6.73 hours	49.72 seconds
DAX 30 / $\boldsymbol{GRXEUR\_2017}$	3.96 hours	30.65 seconds

Table 4: Summary of total training and testing times for each dataset after running jobs on the Getafix cluster.

Chapter 4
Results and Discussion

4.1 Training Loss

The training losses from training each of the 10 FDRNNs in each selected dataset are presented in Figures 17, 18 and 19 below. Discussion of these results follow on the next page.

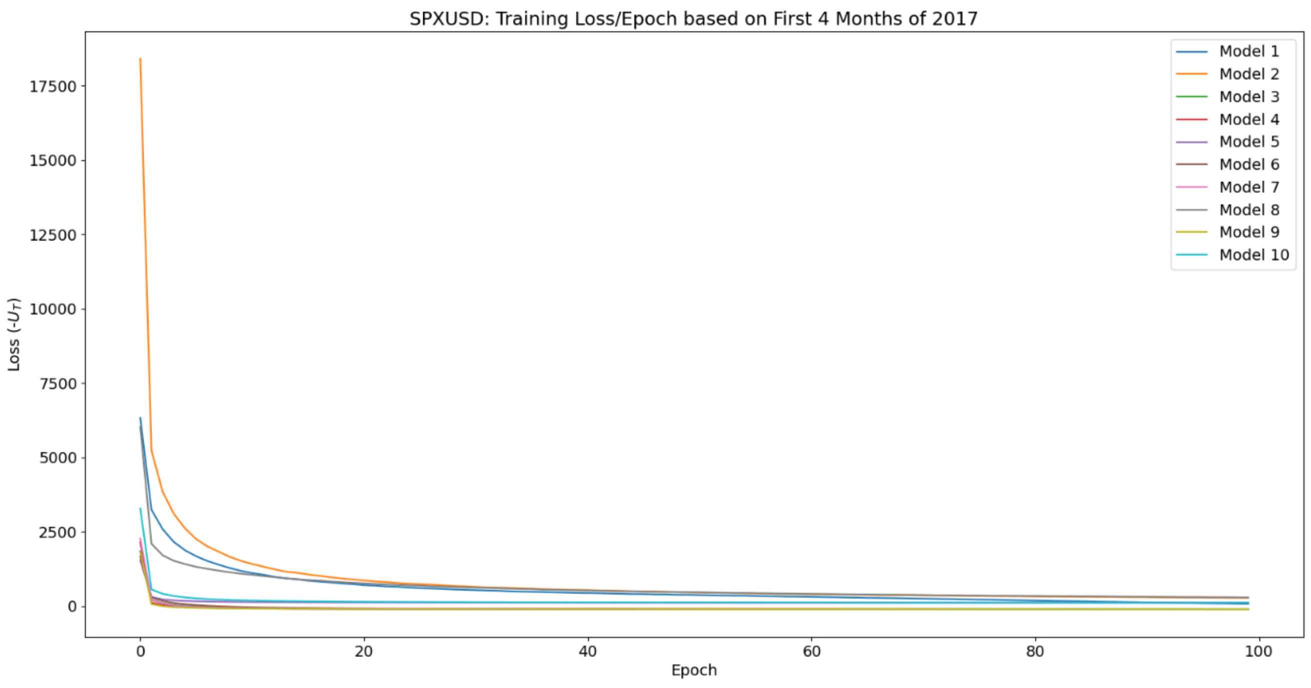


Figure 17: Training loss for each of the 10 FDRNNs on the minutely-priced S&P 500 futures in 2017 dataset.

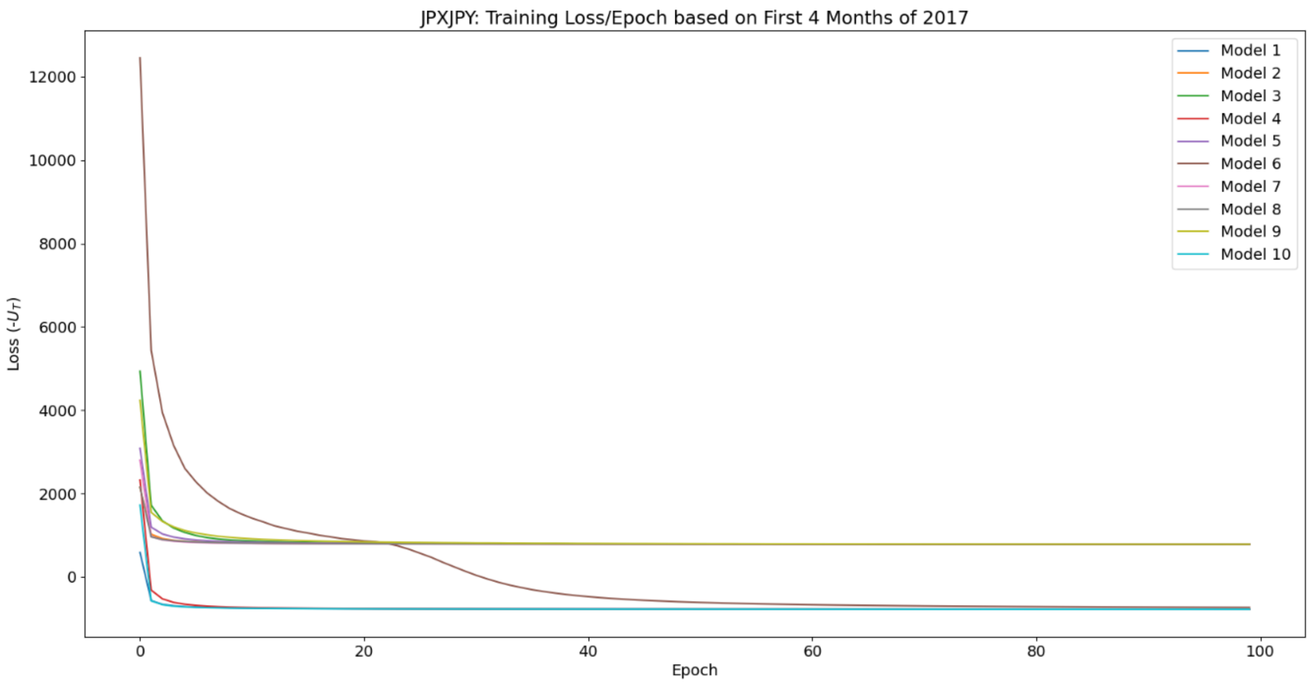


Figure 18: Training loss for each of the 10 FDRNNs on the minutely-priced NIKKEI 225 futures in 2017 dataset.

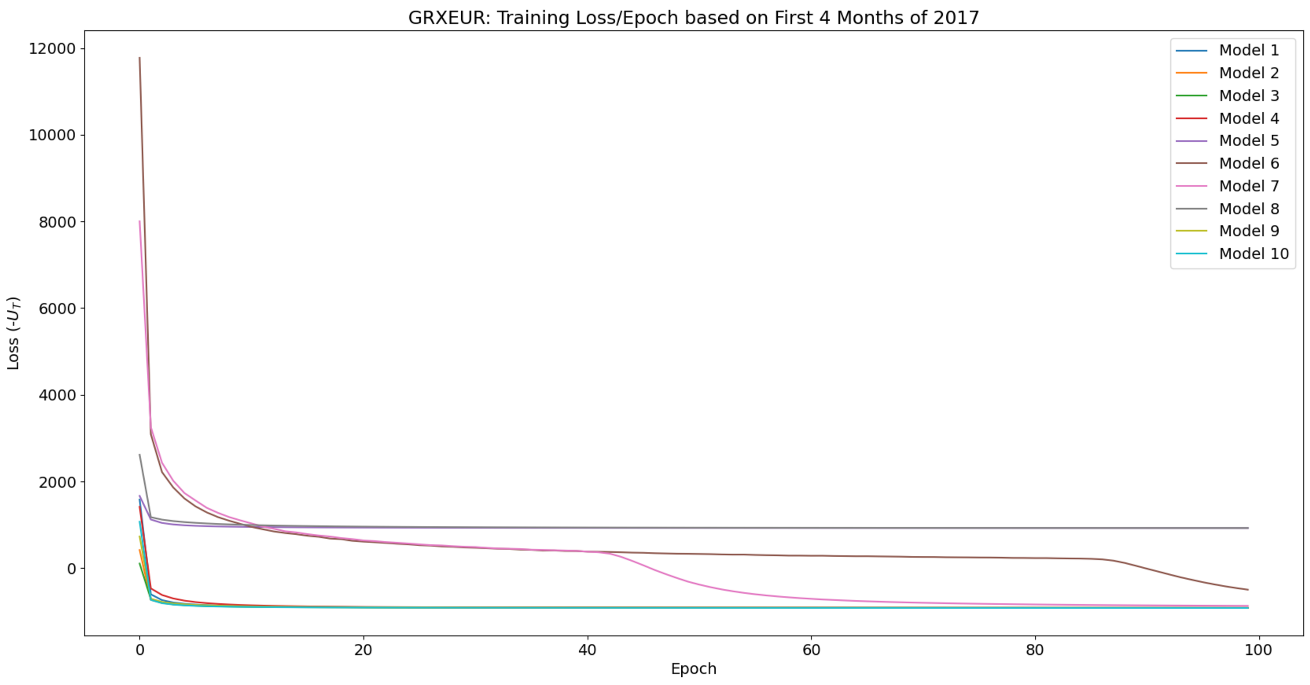


Figure 19: Training loss for each of the 10 FDRNNs on the minutely-priced DAX 30 futures in 2017 dataset.

The convex shape of all the model training loss curves seen in all of Figures 17, 18 and 19 seem to suggest stable loss convergence. There are three key points of discussion to talk about here.

The first point is that most of the loss curves converge quite rapidly: setting $num\_epochs = 100$ iterations is seemingly excessive as most models appear to converge to some local minima by the 20th iteration itself. The exceptions are models 6 and 6 and 7 on the NIKKEY 225 and DAX 30 $train$ sets (Figures 18 and 19) respectively. These provide evidence of FDRNN models jumping from one local minima at a higher loss to another local minima with lower loss during training. Nevertheless, almost all other models appear to stay in one local minima for the rest of the training process, so early stopping via using reduced training iterations num_epochs appears to be very beneficial for being efficient with the total time used in training as well as computational resource usage.

The second point is that any model whose training loss converges above 0 is considered naturally unprofitable because $U_{T_{train}} = -1 \times loss_{T_{train}} = -1 \times -1 \times U_{T_{train}}$ is then below 0. In other words, the model does not learn sufficiently well enough to automatically trade a financial asset profitably. This is the case for most models in Figures 17, 18 and 19 and is indicative of either getting trapped in common local minima or could potentially suggest even deeper issues such as vanishing or exploding gradients. Unfortunately, it can be shown that it is due to the latter case and this is proved and explained in greater detail in Section 4.3.

The final point is that creating an ensemble of FDRNNs certainly seems to be the right approach since the deeper insights gained from the first and second points would not have been realised otherwise. However, one must be extremely cautious, in general, if experiencing highly promising results as this could imply overfitting which is one of the major pitfalls to be wary of when backtesting an algorithmic trading strategy.

4.2 Accumulated Rewards Over Training Period

The accumulated rewards over the training period for each of the 10 FDRNNs in each chosen dataset are presented in Figures 20, 21 and 22 below. Discussion of these results follow on the next page.

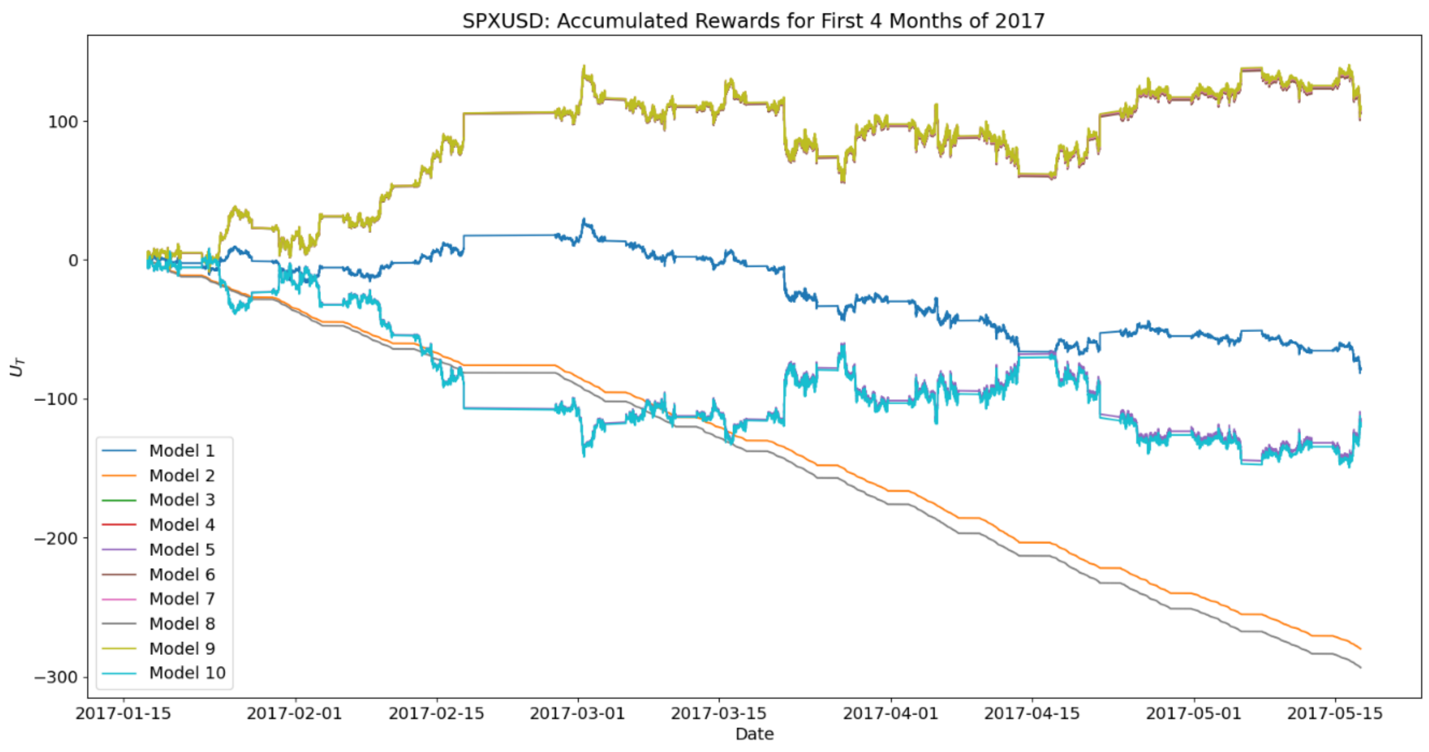


Figure 20: Accumulated rewards U_t over the training period for each of the 10 FDRNNs on the minutely-priced S&P 500 futures in 2017 dataset.

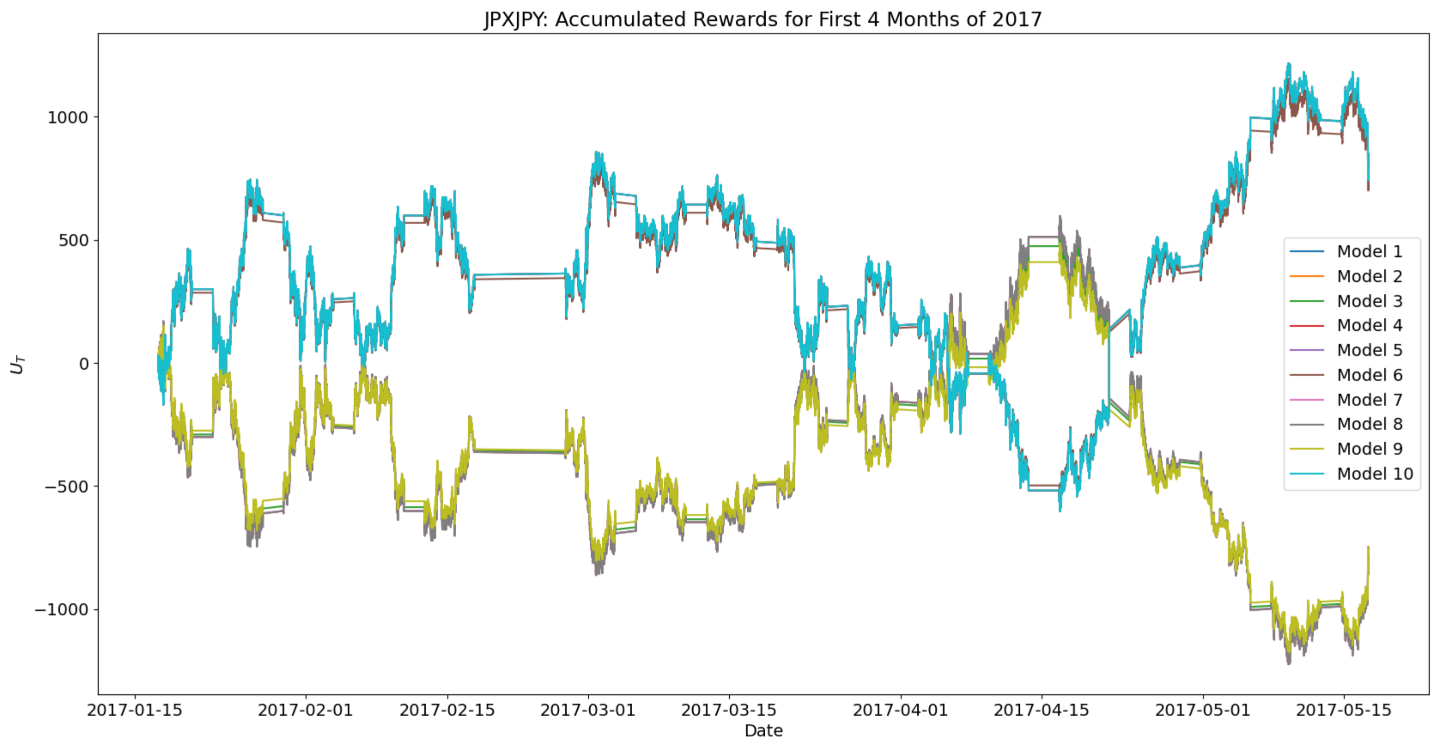


Figure 21: Accumulated rewards U_t over the training period for each of the 10 FDRNNs on the minutely-priced NIKKEI 225 futures in 2017 dataset.

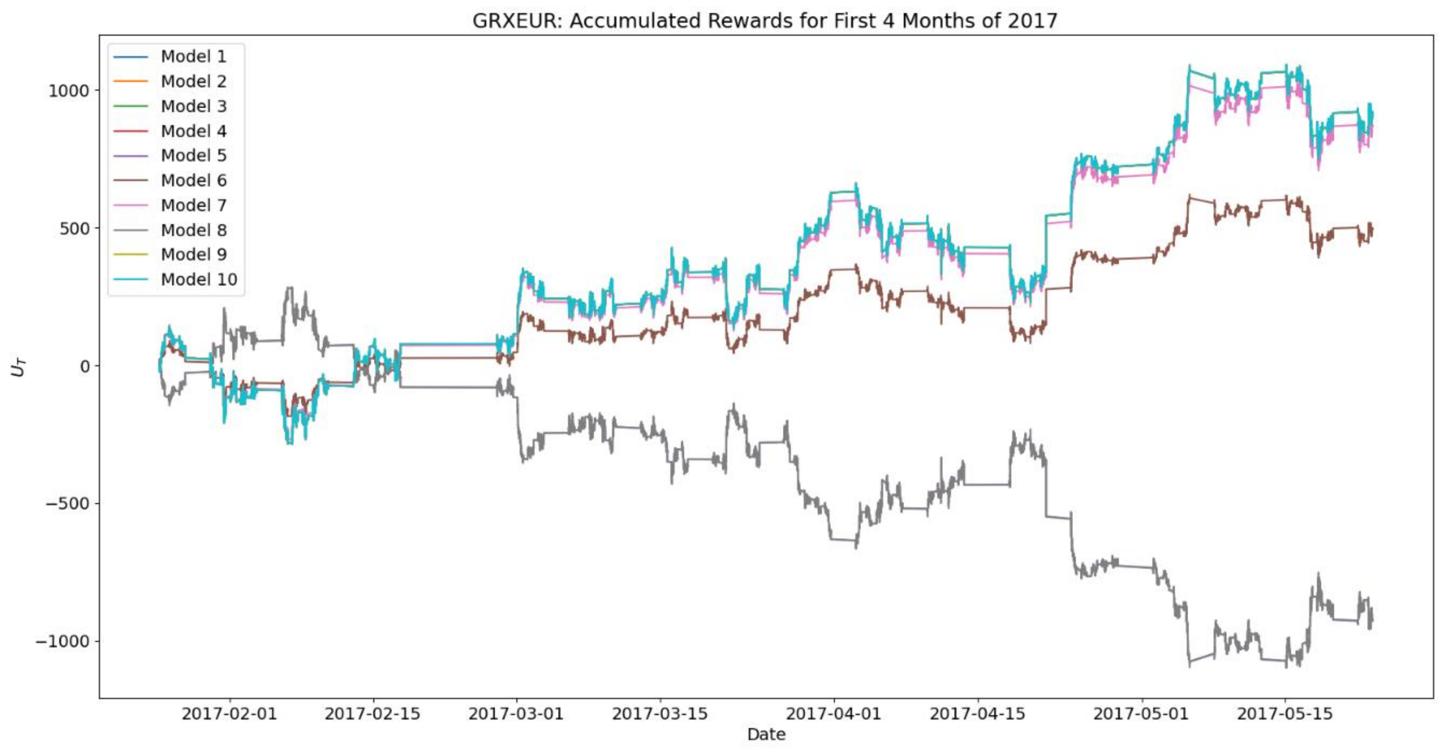


Figure 22: Accumulated rewards U_t over the training period for each of the 10 FDRNNs on the minutely-priced DAX 30 futures in 2017 dataset.

Figures 20, 21 and 22 showcase results that are unfortunately unexpected and absurd. It can be seen that each of the 10 trained models either share the same pattern or mirror the complete opposite pattern of accumulated rewards U_t as other models in each dataset. Observe that this is only possible if each of the 10 trained models either replicate or mirror the opposite trade decisions made by the other models in each dataset. This is because each trading decision δ_t at time t affects R_t which in turn affects U_t. The shared pattern-like behaviour witnessed in these Figures is absurd because the original goal of training an ensemble of FDRNNs for each dataset was to generate separate stochastic models whose losses would converge in different local minima thereby yielding a range of different possible paths of U_t over the training period. As mentioned before, this type of strange behaviour can be shown to be a direct consequence of the issue of exploding gradients. To avoid repetition, the full mathematical reasoning behind how this is possible is provided in Section 4.3 below.

One key observation made is that the results seen in the figures of this section are directly related to the results seen in the figures of Section 4.1 above. More specifically, the models among the 10 FDRNNs that achieved lower loss at convergence in Figures 17, 18 and 19 seem to perform better in terms of accumulated rewards in Figures 20, 21 and 22 respectively. This is logical since $U_t = -1 \times loss_t$ so a model whose loss converges at a lower level automatically must achieve higher accumulated rewards over the training period.

4.3 Accumulated Rewards Over Testing Period

The accumulated rewards over the testing period for the highest performing FDRNNs in each selected dataset are presented in Figures 23, 24 and 25 below. Discussion of these results follow on the next page.

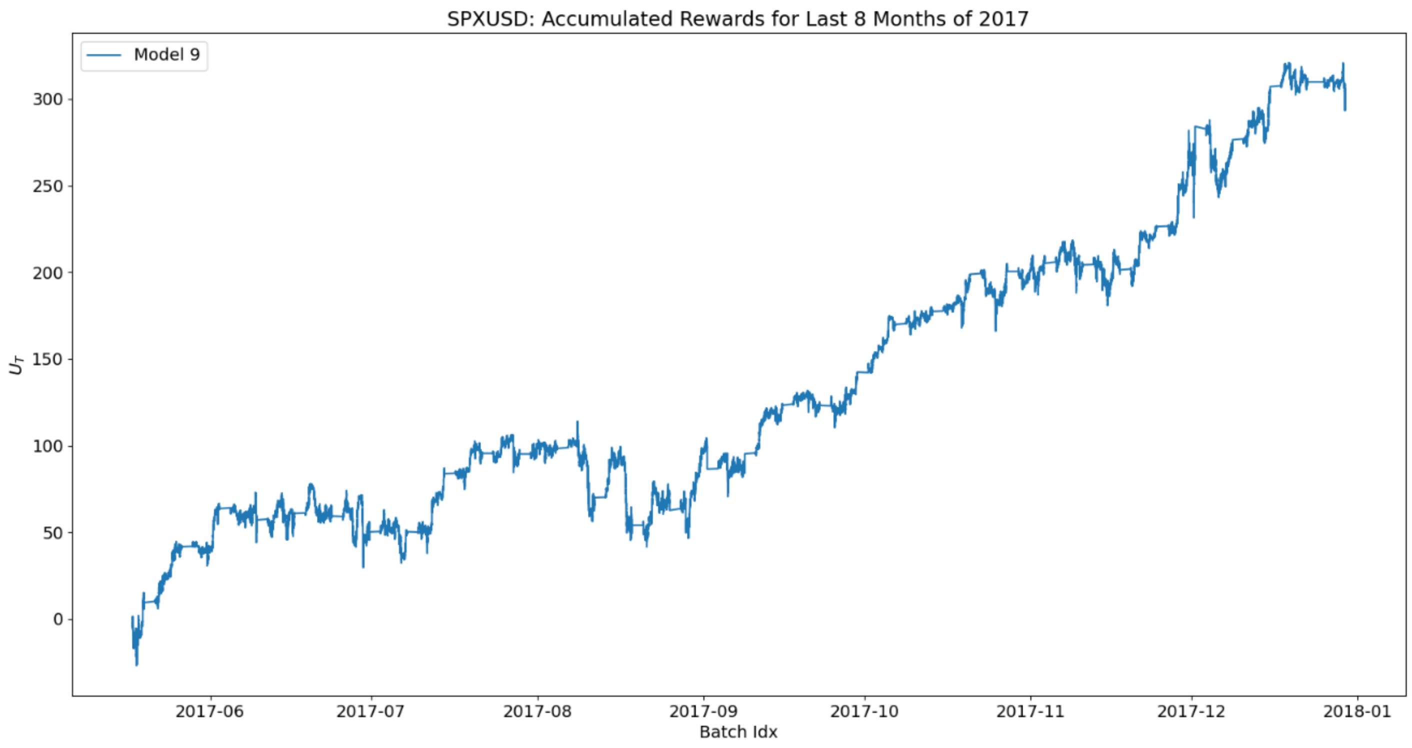


Figure 23: Accumulated rewards U_t over the testing period for the highest performing FDRNN on the minutely-priced S&P 500 futures in 2017 dataset.

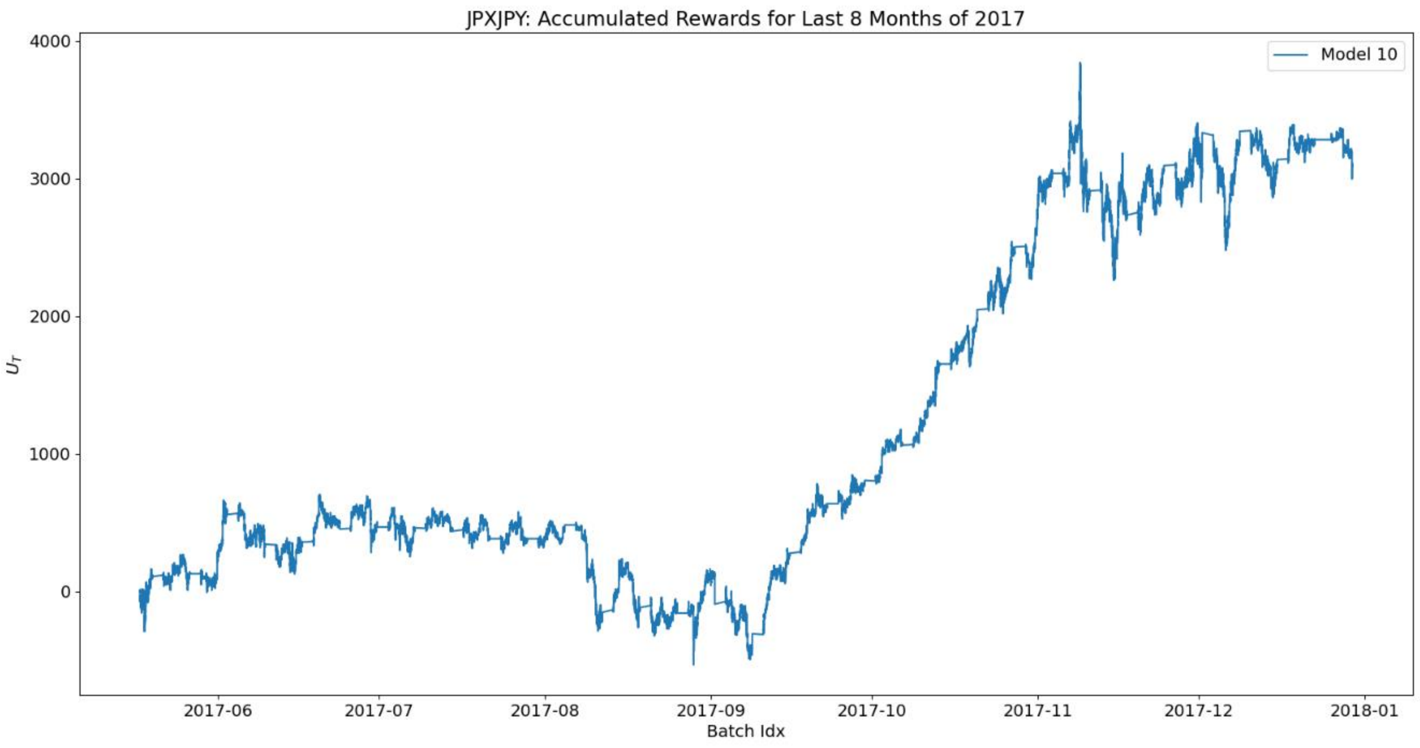


Figure 24: Accumulated rewards U_t over the testing period for the highest performing FDRNN on the minutely-priced NIKKEI 225 futures in 2017 dataset.

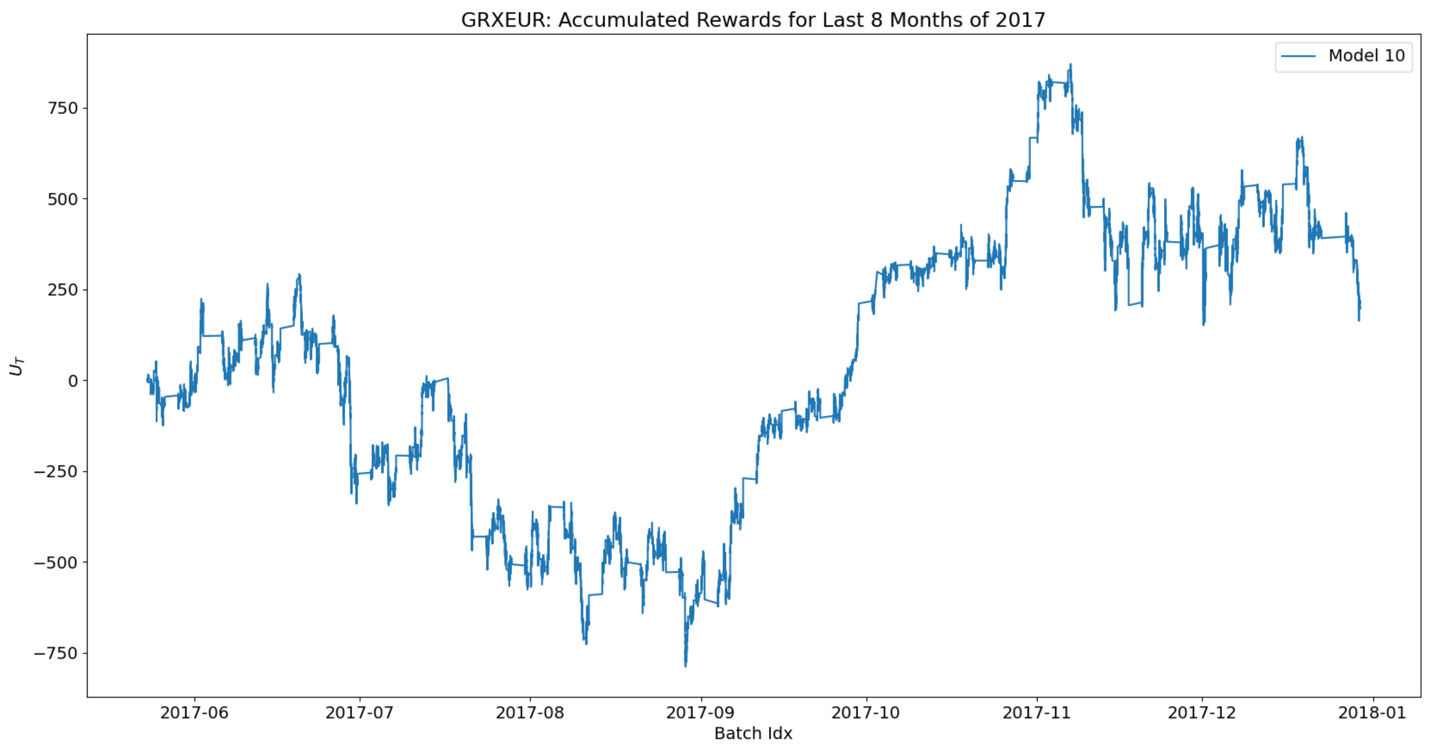


Figure 25: Accumulated rewards U_t over the testing period for the highest performing FDRNN on the minutely-priced DAX 30 futures in 2017 dataset.

Whilst the test results in Figures 23, 24 and 25 appear to look quite promising, it is heavily misleading. A closer comparison between each of these figures and the first row of graphs (p_t) from the EDA showcased in Figure 15 Section 3.3 highlight that the highest performing FDRNNs are simply replicating each price time-series of the three respective datasets. In other words, the equivalent trading decision δ_t made at each time t is to just buy a single futures contract and hold it indefinitely.

By backtracking, it can be mathematically reasoned that this fault is attributed to the issue of exploding gradients. First recall that each $\delta_t = \tanh(< \boldsymbol{w}, \boldsymbol{F}_t > +b + u\delta_{t-1})$. Hence, if the accumulated rewards U_t for each of the highest performing FDRNN models over the training and testing periods exactly match the minutely-price time series in each of the datasets, that must mean that $\delta_t = 1$ for all $t = seq\_length, \dots, T_{test}$ i.e. the only trading decision ever made is to simply buy and hold a single futures contract. Now since the hyperbolic tangent activation function $\tanh \in (-1,1)$, the only way to obtain $\delta_t = 1$ is for the entire inner expression $(< \boldsymbol{w}, \boldsymbol{F}_t > +b + u\delta_{t-1})$ to evaluate to a very large positive value. Thus, more than one of $\boldsymbol{w}, \boldsymbol{F}_t, b$ or u is extremely large, noting that δ_{t-1} is bounded $\in (-1,1)$ just like δ_t again due to the hyperbolic tangent. In practice, it is possible that $\boldsymbol{F}_t$ itself could be very large since the trainable parameters associated with the ANN components in the unrolled FDRNN could also be very large hence amplifying the inputs passed through the deep representations. Finally, the reason why these trainable parameters in addition to $\boldsymbol{w}$, b and u are relatively large can be attributed to the issue of exploding gradients.

An alternative explanation for buying and holding is that this is actually the optimal policy founded by each of the highest performing FDRNN models on each of the respective datasets. However, this is unlikely because of the mirroring effect evident in Figures 20, 21 and 22 – the fact that short-selling and holding whilst suffering increasing loss even forms part of the optimal policy guarantees that the underlying issue is truly due to exploding gradients. This makes it apparent that the weight decay (L2 regularisation) used to reduce the weights may not be sufficient – further experimentation by modifying this value is required.

Chapter 5
Conclusion

5.1 Summary

The purpose of this project thesis was to implement, test and review an innovative algorithmic trading strategy based on the novel paper “Deep Direct Reinforcement Learning for Financial Signal Representation and Trading” by Deng et al. Initially, to set the context, an extensive summary of the field of algorithmic trading was provided including its advantages, disadvantages and the different strategies that were invented. Next, a brief summary of the paper was provided. It was observed during personal reading that heavy use was made of various elements of AI in developing the novel FDRNN model. Accordingly, an overview covering the necessary concepts in RL, ML, ANNs and Deep Learning, RNNs and BPTT were provided to prepare the reader to better understand the decisions made by the authors in developing the FDRNN system. After setting up necessary mathematical notation, the authors’ model was incrementally built bottom-up from the first revision DRL model to the second revision DRNN model to the final revision FDRNN model. Following this, the specifics associated with a programmed implementation were discussed in great detail. Python was chosen as the programming language of preference due to its ease and simplicity, automated memory management, support for OOP and thousands of external libraries as well as largescale community support. Three datasets were obtained, preprocessed and visualised in preparation for training and testing the FDRNN model as a viable algorithmic trading strategy. These were minutely-prices for the S&P 500, NIKKEI 225 and DAX 30 futures over 2017. Following this, a modified version of the authors’ pseudocode in addition to the personal program design and verification procedure were covered extensively. An experimentation plan involving training an ensemble of FDRNNs and evaluating the highest-performing model on test data was setup. The experimentation plan was then executed on UQ SMP’s Getafix cluster using powerful NVIDIA CUDA GPUs to massively reduce the time taken to generate the results. The subsequent results were used to evaluate the FDRNN’s performance, as implemented in this thesis, by visualising training losses as well as accumulated rewards over the training and the testing periods. These uncovered a persistent issue of exploding gradients that perturbed the ability to assess the FDRNN’s viability as a profitable algorithmic trading strategy. This provides a natural transition to conclude the thesis by discussing some of the discovered limitations and extensions to consider for future work on the FDRNN model.

5.2 Limitations

Some of the limitations of the FDRNN model along with this thesis are provided below:

- The model is highly likely to suffer from exploding gradients when trained using standard BPTT. Accordingly, the results suffered but this does not reflect the true potential of the FDRNN as a viable algorithmic trading strategy.
- The trading policy δ_t at each time t is generated using hyperbolic tangent $\tanh \in (-1,1)$ which is a continuous mapping. It's not clear what an output that $\notin \{-1,0,1\}$ should imply, whether it should be rounded up or down, etc.
- All the datasets selected for training and testing of the FDRNN model are predominantly upwards trending. This is true even for the DAX 30 futures which is the most volatile dataset for 2017 (see Figure 15 in Section 3.3).
- Constant parameters (notably transaction cost c) and hyperparameters were used in training and testing multiple FDRNN models across multiple datasets. These should be selected more cautiously but varying them to specifically suit a dataset might result in overfitting and poor generalisation on unseen data – a pitfall to be wary of when backtesting an automated trading strategy.
- The FDRNN model does not take into account specific asset features e.g. contract expiry for futures. An optimal policy that buys/sells and holds a futures contract over long periods of time may then be unrealistic due to contract expiration.
- Unrealistic assumptions such as no taxes, sufficient liquidity available at all times, etc.

5.3 Extensions

Lastly, to conclude this thesis, a few of the natural extensions to the work done in this thesis now follow:

- Following the authors' work, using an Autoencoder to directly initialise the trainable parameters associated with the deep representation part of the FDRNN in order to guide model training.
- Following the authors' work, completely implementing the Task-Aware BPTT algorithm to handle the issues of vanishing or exploding gradients.
- Replacing the standard Elman Neural Network representing the RNN component with a more sophisticated and modern RNN such a Long-Short Term Memory or Gated Recurrent Unit model.
- Further experimentation with altering the hyperparameters. Logically, it is very time consuming and wasteful of computational resources to fine-tune each of them one-by-one at a time (standard Grid Search). Perhaps considering using Random Grid Search here would be highly beneficial.

Bibliography

Hasbrouck, J., Sofianos, G. and Sosebee, D., 1996. *New York Stock Exchange Systems and Trading Procedures*. [online] p.21. Available at: http://people.stern.nyu.edu/jhasbrou/Research/Working%20Papers/NYSE.PDF [accessed 17 November 2021].

Moore, G., 1965. *Cramming more components onto integrated circuits*. **Journal of Electronics**, [online] 38(8). Available at: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf [accessed 17 November 2021].

Ritchie, H. and Roser, M., 2013. *Technological Progress*. [online] **Our World In Data**. Available at: https://ourworldindata.org/technological-progress [accessed 16 November 2021].

Aquilina, M., Budish, E. and O'Neill, P., 2021. *Quantifying the high-frequency trading "arms race"*. **BIS Working Papers**, [online] (955), p.0. Available at: https://www.bis.org/publ/work955.pdf [accessed 17 November 2021].

Hu, E., 2018. *Intentional Access Delays, Market Quality, and Price Discovery: Evidence from IEX Becoming an Exchange*. **Division of Economic and Risk Analysis' Working Paper Series**, [online] p.1. Available at: https://www.sec.gov/files/07feb18_hu_iex_becoming_an_exchange.pdf [accessed 17 November 2021].

Seth, S., 2015. *The World of High-Frequency Algorithmic Trading*. [online] **Investopedia**. Available at: https://www.investopedia.com/articles/investing/091615/world-high-frequency-algorithmic-trading.asp [accessed 16 November 2021].

Ni, J. and Zhang, C., 2007. *An Efficient Implementation of the Backtesting of Trading Strategies*. [online] p.126. Available at: https://www.researchgate.net/publication/220945302_An_Efficient_Implementation_of_the_Backtesting_of_Trading_Strategies [accessed 17 November 2021].

Kumar Dubey, R., 2021. *ALGORITHMIC TRADING: IS THE MACHINE TAKING OVER OUR EQUITY MARKETS?* [online] p.30. Available at: https://www.researchgate.net/publication/310261822_ALGORITHMIC_TRADING_IS_THE_MACHINE_TAKING_OVER_OUR_EQUITY_MARKETS [accessed 20 November 2021].

Nanex Research, 2014. *Citadel's Rogue Algos and FINRA's lethargic response*. [online] **Nanex**. Available at: http://www.nanex.net/aqck2/4668.html [accessed 16 November 2021].

Mandes, A., 2016. *Algorithmic and High-Frequency Trading Strategies: A Literature Review*. [online] (25), pp.3-19. Available at: https://www.econstor.eu/bitstream/10419/144690/1/860290824.pdf [accessed 21 November 2021].

Almgren, R. and Chriss, N., 2000. *Optimal Execution of Portfolio Transactions*.

Ryś, P. and Ślepaczuk, R., 2018. *Machine Learning Methods in Algorithmic Trading Strategy Optimization – Design and Time Efficiency*. **Central European Economic Journal**, [online] 5(52), p.209. Available at: https://www.researchgate.net/publication/335114551_Machine_Learning_Methods_in_Algorithmic_Trading_Strategy_Optimization_-_Design_and_Time_Efficiency [accessed 21 November 2021].

Deng, Y., Bao, F., Kong, Y., Ren, Z. and Dai, Q., 2017. *Deep Direct Reinforcement Learning for Financial Signal Representation and Trading*. **IEEE Transactions on Neural Networks and Learning Systems**, [online] 28(3). Available at: http://cslt.riit.tsinghua.edu.cn/mediawiki/images/a/aa/07407387.pdf [accessed 16 November 2021].

Poole, D. and Mackworth, A., 2010. *Dimensions of Complexity*. [online] **Artificial Intelligence: Foundations of Computational Agents**. Available at: https://artint.info/html/ArtInt_12.html [accessed 17 November 2021].

Bhatt, S., 2018. *5 Things You Need to Know about Reinforcement Learning*. [online] **KDnuggets**. Available at: https://www.kdnuggets.com/2018/03/5-things-reinforcement-learning.html [accessed 17 November 2021].

Hausknecht, M. and Stone, P., 2016. *On-Policy vs. Off-Policy Updates for Deep Reinforcement Learning*. [online] p.1. Available at: https://www.cs.utexas.edu/~pstone/Papers/bib2html-links/DeepRL16-hausknecht.pdf [accessed 17 November 2021].

IBM, 2020. *Neural Networks*. [online] **IBM Cloud Education**. Available at: https://www.ibm.com/au-en/cloud/learn/neural-networks [accessed 17 November 2021].

Ravichandiran, S., 2020. *Deep Reinforcement Learning with Python*. 2nd ed. [S.l.]: **Packt Publishing**, Chapter 1 Section 5.

Zhang, A., Lipton, Z., Li, M. and Smola, A., 2020. *Dive into Deep Learning*. 1st ed. pp.131-173, 325-344.

Larson, A., 2020. *A Review of the Math Used in Training a Neural Network*. [online] **Morioh**. Available at: https://morioh.com/p/d70aa769173a [accessed 20 November 2021].

Delua, J., 2021. *Supervised vs. Unsupervised Learning: What's the Difference?* [online] **IBM Cloud Education**. Available at: https://www.ibm.com/cloud/blog/supervised-vs-unsupervised-learning [accessed 17 November 2021].

Goodfellow, I., Bengio, Y. and Courville, A., 2016. *Deep Learning*. 1st ed. **MIT Press**, p.200.

Elman, J., 1990. *Finding Structure in Time*. **Journal of Cognitive Science**, [online] 14, p.183. Available at: https://crl.ucsd.edu/~elman/Papers/fsit.pdf [accessed 17 November 2021].

IBM, 2020. *Recurrent Neural Networks*. [online] **IBM Cloud Education**. Available at: https://www.ibm.com/cloud/learn/recurrent-neural-networks [accessed 17 November 2021].

Cao, Y., Zeng, Z. and Huang, T., 2018. *A Modified Elman Neural Network with a New Learning Rate Scheme*. **Journal of Neurocomputing**, [online] 286, p.2. Available at: https://www.researchgate.net/publication/322843098_A_Modified_Elman_Neural_Network_with_a_New_Learning_Rate_Scheme [accessed 17 November 2021].

Nielsen, M., 2015. *Chapter 5 – Why are deep neural networks hard to train? Neural Networks and Deep Learning*. **Determination Press**. Available at: http://neuralnetworksanddeeplearning.com/chap5.html#what%27s_causing_the_vanishing_gradient_problem_unstable_gradients_in_deep_neural_nets [accessed 18 November 2021].

Moody, J. and Saffell, M., 2001. *Learning to Trade via Direct Reinforcement*. **Journal of IEEE Transactions on Neural Networks**, [online] 12(4), pp.876-877. Available at: https://ieeexplore.ieee.org/document/935097 [accessed 18 November 2021].

Lin, C. and Lee, C., 1991. *Neural-network-based fuzzy logic control and decision system*. **Journal of IEEE Transactions on Computers**, [online] 40(12), pp.1320-1336. Available at: https://ieeexplore.ieee.org/document/106218 [accessed 18 November 2021].

Williams, M., 2020. *FutureSharks / financial-data*. [online] **GitHub**. Available at: https://github.com/FutureSharks/financial-data [accessed 19 November 2021].

Wei, W. and Reilly, D., 1999. *Time Series Analysis: Univariate and Multivariate Methods*. 2nd ed. p.2.

PyTorch, 2021. *MODULE*. [online] **PyTorch**. Available at: https://pytorch.org/docs/stable/generated/torch.nn.Module.html [accessed 21 November 2021].

PyTorch, 2021. *TORCH.TENSOR*. [online] **PyTorch**. Available at: https://pytorch.org/docs/stable/tensors.html [accessed 21 November 2021].

PyTorch, 2021. *RNN*. [online] **PyTorch**. Available at: https://pytorch.org/docs/stable/generated/torch.nn.RNN.html [accessed 21 November 2021].

NVIDIA, 2021. *CUDA Zone*. [online] **NVIDIA Developer**. Available at: https://developer.nvidia.com/cuda-zone [accessed 21 November 2021].

Hastie, T., Tibshirani, R. and Friedman, J., 2021. *The Elements of Statistical Learning*. 2nd ed. **Springer**, p.605.

Mohan, A., 2019. *Neural Network Loss Visualization*. [online] **Telesens**. Available at: https://www.telesens.co/2019/01/16/neural-network-loss-visualization/ [accessed 22 November 2021].

Appendix

Python script $fdrnn.py$ to train and test FDRNN model on a given dataset:

```
# MATH7013 Project
# Deep Direct RL for Financial Signal Representation and Training - paper by Deng et al.
# Program for training the FDRNN
# Implemented by Joel Thomas
import sys
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
import torch
import timeit

pd.options.mode.chained_assignment = None
# Set font size when plotting
plt.rcParams.update({'font.size': 14})



# Class for fuzzy layer
class FuzzyLayer(torch.nn.Module):
    """
    Custom PyTorch neural network module that implements a single fuzzy layer used as the second
layer in the FDRNN.
    """
    def __init__(self, input_size, fuzzy_deg, fuzzy_params, batch_size, seq_length):
        """
        Initialises the fuzzy layer with important parameters.
        """
        super(FuzzyLayer, self).__init__()
        self.input_size = input_size                        # Size of each input f_t
        self.fuzzy_deg = fuzzy_deg                           # Number of fuzzy degrees to assign
to each dimension of f_t
        self.fuzzy_params = fuzzy_params                     # 2D Matrix of means and widths of
each node in the fuzzy layer
        self.batch_size = batch_size                         # Number of total inputs in a single
batch fed into the FDRNN
        self.seq_length = seq_length                          # Number of consecutive timesteps
(f_ts) considered in one input
        self.output_size = self.input_size*self.fuzzy_deg    # Size of each output after passing
f_t through fuzzy layer

    def forward(self, x):
        """
        Perform a forward pass of input x (batch containing multiple sequences of f_ts) through
the fuzzy layer and return
        the fuzzied output for each f_t in each sequence in the given batch.
        """
        # Stores fuzzied x
        fuzzied_x = []
        for seq in x:
            # Stores fuzzied (sequence of) f_ts
            fuzzied_seq = []
            for f_t in seq:
                # Stores fuzzied f_t
                fuzzied_f_t = []
                for k in range(self.fuzzy_deg):
                    curr_idx = k*self.input_size
                    next_idx = (k + 1)*self.input_size
                    # Use Gaussian membership function in generating fuzzied representation of
each provided f_t
                    fuzzied_f_t.append(np.exp(-1*np.divide(np.square(f_t                          -
self.fuzzy_params[curr_idx:next_idx, 0]),
                                                                self.fuzzy_params[curr_idx:next_idx,
1])))

                fuzzied_seq.append(fuzzied_f_t)

            fuzzied_x.append(fuzzied_seq)

        fuzzied_x   =   np.reshape(np.array(fuzzied_x,   dtype=np.float32),   (self.batch_size,
self.seq_length, self.output_size))
        # Store on device memory to help train model on GPU
        x = torch.cuda.FloatTensor(fuzzied_x)
        return x

    def extra_repr(self):
        """
        Use when printing information about an initialised fuzzy layer.
        """
        return f"in_features={self.input_size}, out_features={self.output_size}"



# Class for FDRNN model
class FDRNN(torch.nn.Module):
    def   __init__(self,   input_size,   fuzzy_deg,   fuzzy_params,   batch_size,   seq_length,
fc_hidden_size, num_fc, output_size,
                 rec_hidden_size, num_rec):
        """
        Initialises the FDRNN with important parameters.
        """
        super(FDRNN, self).__init__()
        self.input_size = input_size            # Size of each input f_t
        self.batch_size = batch_size              # Number of total inputs in a single batch fed
into the FDRNN
        self.seq_length = seq_length            # Number of consecutive timesteps (f_ts) considered
in one input
        self.fc_hidden_size = fc_hidden_size     # Number of hidden cells in each fully-connected
(dense/linear) layer
        self.num_fc = num_fc                    # Number of fully-connected layers
        self.output_size = output_size           # Size of each output F_t after passing through
fuzzy and all deep layers
        self.rec_hidden_size = rec_hidden_size   # Number of features in a hidden state for the
RNN component (shape of delta_t)
        self.num_rec = num_rec                  # Number of (stacked) recurrent layers
        # Fuzzy layer
        self.fuzzy  =  FuzzyLayer(self.input_size,  fuzzy_deg,  fuzzy_params,  self.batch_size, 
self.seq_length)


        # Fully-connected layers (deep transformations)
        self.fcs        =         torch.nn.ModuleList([torch.nn.Linear(self.fuzzy.output_size, 
self.fc_hidden_size)])
        self.fcs.extend([torch.nn.Linear(self.fc_hidden_size,  self.fc_hidden_size)  for  i  in 
range(self.num_fc - 2)])
        self.fcs.append(torch.nn.Linear(self.fc_hidden_size, self.output_size))


        # RNN layer (to calculate delta_t for each f_t for each sequence in given batch)
        self.rnn   =   torch.nn.RNN(self.output_size,   self.rec_hidden_size,   self.num_rec, 
nonlinearity='tanh', bias=True,
                                batch_first=True)


    def forward(self, x):
        """
        Perform a forward pass of input x (batch containing multiple sequences of f_ts) through 
the fuzzy, fully-connected and
        RNN layer and returns delta_t for each f_t in each sequence in the given batch.
        """
        x = self.fuzzy.forward(x)
        for fc in self.fcs:
            x = fc(x)
        # Next line is equivalent to delta_t, h = self.rnn(x)
        x, h = self.rnn(x)
        # Flatten from (batch_size, seq_length, rec_hidden_size)-shape tensor to 1D tensor
        # x = torch.flatten(x)
        return x



#  Batch generator for generating batches and profit function to be used during training and 
testing the FDRNN
def make_batch_generator(df, batch_size, seq_length):
    """
    Creates a batch generator that yields a single batch and corresponding sequence of z_ts 
whenever a call to the
    resulting generator object is made. Generated batches are used as raw input to the FDRNN 
during training and testing.
    The corresponding sequence of z_ts is used to help calculate the resulting loss after a forward 
pass has been made.
    """
    for i in range(0, len(df) - batch_size, batch_size):
        batch = []
        z_ts = []
        for j in range(batch_size):
            seq = df.iloc[i+j:i+j+seq_length].values
            batch.append(seq)
            z_t = df["z_t"].iloc[i+j+seq_length-1]
            z_ts.append(z_t)
        batch = np.reshape(np.array(batch), (batch_size, seq_length, input_size))
        z_ts = torch.reshape(torch.tensor(z_ts), (batch_size, 1)).to(device)
        yield (batch, z_ts)



# Function to calculate U_t
def calc_U_t(delta_ts, z_ts, c, pnl_over_time):
    """
    Calculate (a slice) of U_t based on the forward pass output of a single batch fed into the FDRNN. The loss function for
    the FDRNN is given by the negative of this function and hence, this function can be used to calculate the loss on the
    output of a single batch fed into the FDRNN. Each loss is used to update the parameters belonging to the fully-connected
    layers and RNN layer via regular backpropagation and backpropagation through time (BPTT) respectively. The fuzzy layer is
    excluded since it does not contain any parameters (recall used k-means to finalise the means and widths of the fuzzy nodes).
    """
    # Short way - use this for calculating gradients and updating
    U_t = torch.sum(torch.mul(delta_ts[:, -2], z_ts) - c*torch.abs(delta_ts[:, -1] - delta_ts[:, -2]))
    # Long way - use this for plotting accumulated rewards R_ts (i.e. true U_T) over time
    if pnl_over_time != []:
        for i in range(delta_ts.shape[0]):
            pnl_over_time.append(pnl_over_time[-1] + delta_ts[i, -2].item()*z_ts[i].item() -
                                 c*abs(delta_ts[i, -1].item() - delta_ts[i, -2].item()))
    # Return U_t since loss = -1*U_t and use loss.backward() for updating all gradients
    return U_t



if __name__ == "__main__":
    # Check if GPU is available for training the FDRNN, quit program otherwise
    if torch.cuda.is_available():
        device = torch.cuda.current_device()
        print(f"NVIDIA CUDA GPU available: {torch.cuda.get_device_name(device)}\n")
    else:
        print("Could not detect any capable NVIDIA CUDA GPU!")
        print("Exiting program...")
        exit()



    asset = sys.argv[1]
    year = sys.argv[2]
    # Load in a preprocessed dataset
    df = pd.read_csv(f"../../Data Preprocessing/{asset}_{year}.csv", index_col="date")
    df.drop("p_t", axis=1, inplace=True)
    # Convert date column to datetime type
    df.index = pd.to_datetime(df.index)

    # Set important parameters
    # Number of FDRNNs to train in an Ensemble
    num_models = 10
    # Transaction cost to use for training and testing
    c = 15
    # Number of training samples
    num_train = len(df)//3
    # Training set
    train_df = df.iloc[:num_train]
    # Testing set
    test_df = df.iloc[num_train:]
    # Number of training iterations
    num_epochs = 100
    # Dimension of each feature vector f_t
    input_size = len(df.columns)
    # Number of clusters in k-means clustering (fuzzy degrees)
    k = 3


    # Set important hyperparameters
    # Number of training samples in each batch
    batch_size = 32
    # Number of time steps to feed into FDRNN at a time
    seq_length = 3
    # Input and output size to each hidden layer in the ANN component
    fc_hidden_size = 128
    # Number of fully-connected hidden layers in the ANN component
    num_fc = 4
    # Size of each output F_t after passing through fuzzy and deep representations
    output_size = 20
    # Number of features in a hidden state for the RNN (shape of delta_t)
    rec_hidden_size = 1
    # Number of (stacked) recurrent layers
    num_rec = 1
    # Learning rate to be used during training
    eta = 0.00001
    # Weight decay (L2 regularisation penalty) to be used during training
    lambda_ = 0.000001

    # Use k-means clustering to store each fuzzy node's mean and width to be used later (i.e.
fuzzy layer initialisation)
    kmeans = KMeans(n_clusters=k).fit(train_df)

    # Get cluster labels for samples and make a new column
    train_df["label"] = kmeans.labels_

    # Stores means and widths required for fuzzy layer initialisation
    fuzzy_params = []
    # Calculate the mean and variance of each dimension (feature) in each cluster using all
training samples
    for label in range(k):
        for column in train_df.columns[:-1]:
            feat_given_k = train_df[train_df["label"] == label][column]
            fuzzy_params.append([feat_given_k.mean(), feat_given_k.var()])

    train_df.drop("label", axis=1, inplace=True)
    # Will always be of shape (input_size*k, 2)
    fuzzy_params = np.array(fuzzy_params, dtype=np.float32)


    # Training several FDRNN models - select "best" model as the one that has the highest final
profit over the training period
    # after having trained each model for num_epochs epochs
    print("Beginning FDRNN training...")
    start_time = timeit.default_timer()

    # Initialise FDRNN model on GPU
    fdrnns = []
    fdrnns_epoch_losses = []
    fdrnns_pnl_over_time = []
    for model in range(num_models):
        print(f"\nTraining model {model + 1}")
        fdrnn = FDRNN(input_size, k, fuzzy_params, batch_size, seq_length, fc_hidden_size, num_fc,
output_size,
                    rec_hidden_size, num_rec).cuda()

        # Initialise SGD optimiser with specified learning rate and L2 penalty
        optimiser = torch.optim.SGD(fdrnn.parameters(), lr=eta, weight_decay=lambda_)

        # Begin training the FDRNN model
        epoch_losses = []
        for epoch in range(1, num_epochs + 1):
            print(f"Epoch: {epoch}")
            batch_generator = make_batch_generator(train_df, batch_size, seq_length)
            epoch_loss = 0
            if epoch == num_epochs:
                pnl_over_time = [0]
            for batch, z_ts in batch_generator:
                optimiser.zero_grad()
                delta_ts = fdrnn(batch)
                # Minimising loss here is equivalent to maximising total accumulated rewards
                if epoch == num_epochs:
                    loss = -1*calc_U_t(delta_ts, z_ts, c, pnl_over_time)
                else:
                    loss = -1*calc_U_t(delta_ts, z_ts, c, [])
                loss.backward()
                optimiser.step()
                epoch_loss += loss.item()

            epoch_losses.append(epoch_loss)

        fdrnns.append(fdrnn)
        fdrnns_epoch_losses.append(epoch_losses)
        fdrnns_pnl_over_time.append(pnl_over_time)

    end_time = timeit.default_timer()
    print(f"\nTotal training time: {end_time - start_time}s\n")

    # Plot training loss function
    plt.figure(figsize=(20, 10))
    for model in range(num_models):
        plt.plot(range(num_epochs), fdrnns_epoch_losses[model], label=f"Model {model + 1}")
    plt.xlabel("Epoch")
    plt.ylabel("Loss (-$U_T$)")
    plt.legend()
    plt.title(f"{asset}: Training Loss/Epoch based on First 4 Months of {year}")
    plt.savefig(f"{asset}_{year}_loss.png", facecolor="w")
    plt.close()

    # Plot accumulated profits over training period using best trained model after final epoch
    plt.figure(figsize=(20, 10))
    for model in range(num_models):
        plt.plot(train_df.index[seq_length-2:len(pnl_over_time)],
fdrnns_pnl_over_time[model][1:], label=f"Model {model + 1}")
    plt.xlabel("Date")
    plt.ylabel("$U_T$")
    plt.legend()
    plt.title(f"{asset}: Accumulated Rewards for First 4 Months of {year}")
    plt.savefig(f"{asset}_{year}_train.png", facecolor="w")
    plt.close()

    # Save ONLY the best trained FDRNN model
    # Find best model based on which one achieved the highest final profit over the training
period after having trained
    # each model for num_epochs epochs
    highest_pnl = max([fdrnns_pnl_over_time[model][-1] for model in range(num_models)])
    idx = [fdrnns_pnl_over_time[model][-1] for model in range(num_models)].index(highest_pnl)
    best_fdrnn = fdrnns[idx]

    # Save best model's training loss function
    np.save(f"{asset}_{year}_epoch_losses.npy",               np.array(fdrnns_epoch_losses[idx],
dtype=np.float32))
    # Save best model's training profit and loss over time
    np.save(f"{asset}_{year}_train_pnl_over_time.npy",       np.array(fdrnns_pnl_over_time[idx],
dtype=np.float32))
    # Save best model's parameters (weights, biases, etc.)
    torch.save(best_fdrnn.state_dict(), f"{asset}_{year}.pt")

    # Testing the best FDRNN model
    print("Beginning FDRNN testing...")
    start_time = timeit.default_timer()

    batch_generator = make_batch_generator(test_df, batch_size, seq_length)
    pnl_over_time = [0]
    with torch.no_grad():
        for batch, z_ts in batch_generator:
            delta_ts = best_fdrnn(batch)
            calc_U_t(delta_ts, z_ts, c, pnl_over_time)

    end_time = timeit.default_timer()
    print(f"Total testing time: {end_time - start_time}s\n")

    # Save best model's testing profit and loss over time
    np.save(f"{asset}_{year}_test_pnl_over_time.npy", np.array(pnl_over_time, dtype=np.float32))

    # Plot accumulated profits over testing period
    plt.figure(figsize=(20, 10))
    plt.plot(test_df.index[seq_length-2:len(pnl_over_time)],   pnl_over_time[1:],   label=f"Model
{idx + 1}")
    plt.xlabel("Batch Idx")
    plt.ylabel("$U_T$")
    plt.legend()
    plt.title(f"{asset}: Accumulated Rewards for Last 8 Months of {year}")
    plt.savefig(f"{asset}_{year}_test.png", facecolor="w")
    plt.close()

    print(f"Successfully trained and tested FDRNN model on {asset} {year} dataset!")
```

Slurm bash script $goslurm\_SPXUSD\_2017.sh$ that uses $fdrnn.py$ to train and test ensemble of FDRNN models on $SPXUSD\_2017$ dataset:

```
#!/bin/bash -l
#
#SBATCH --job-name=MATH7013_project
#SBATCH --nodes=1
#SBATCH --ntasks=1
#SBATCH --ntasks-per-node=1
#SBATCH --cpus-per-task=1
#SBATCH --partition=gpu
#SBATCH --gres=gpu:1
#SBATCH --mem-per-cpu=16G  # memory (GB)
#SBATCH --time=1-00:00           # time (D-HH:MM)

export OMP_NUM_THREADS=${SLURM_CPUS_PER_TASK}
export MKL_NUM_THREADS=${SLURM_CPUS_PER_TASK}
echo "This is job '$SLURM_JOB_NAME' (id: $SLURM_JOB_ID) running on the following nodes: "
echo $SLURM_NODELIST
echo "Running with OMP_NUM_THREADS = $OMP_NUM_THREADS"
echo "Running with SLURM_TASKS_PER_NODE = $SLURM_TASKS_PER_NODE"
echo

# Activate PyTorch Python virtual environment
source ~/VirtualEnv/PyTorch/bin/activate

asset=SPXUSD
year=2017

echo "Running fdrnn.py on ${asset} ${year} dataset..."

# Train and test FDRNN model on selected dataset
time python ../fdrnn.py $asset $year >> ${asset}_${year}.txt
echo

echo "FDRNN training and testing on ${asset} ${year} dataset successfully finished!"
```