Jump detection tests in financial time series – a deep learning approach

Justin Wagener

A dissertation submitted to the Faculty of Commerce, University of Cape Town, in partial fulfilment of the requirements for the degree of Master of Philosophy.

October 17, 2023

MPhil in Mathematical Finance,
University of Cape Town.

Declaration

I declare that this dissertation is my own, unaided work. It is being submitted for the Degree of Master of Philosophy in the University of Cape Town. It has not been submitted before for any degree or examination at any other university.

October 17, 2023 Signed by candidate

Abstract

In most financial market models, the asset price is driven by continuous Brownian motion. An additional complexity to such a model is the inclusion of a discontinuous jump process. Jumps are theorised to be rare, sudden, and thought to be the result of the market reacting to new information. Jump tests are identified as crucial to understand market incompleteness arising from this discontinuity. Across studies, the Lee and Mykland (2007) method emerges as one of the strongest performers in jump detection. This serves as the benchmark to the jump tests created in this dissertation. The first uses a Long Short-Term Memory (LSTM) neural network based supervised learning approach. The second uses unsupervised learning in the form of a Convolutional Neural Network (CNN) autoencoder. Bates, Merton and Stochastic Volatility double Jump (SVJJ) models provide the data used for comparison. For supervised learning, synthetic data is essential as jump labels are needed for training. The autoencoder jump test is an improvement as it does not need labelled jumps to train. This was found to be the best jump test overall when compared out of sample. Both methods were found to beat the benchmark set by Lee and Mykland (2007). The performance metrics used are suited to the imbalanced data sets arising from the assumption of jumps being rare.

Acknowledgements

To my supervisor Professor Peter Ouwehand. Thank you for placing and keeping me on the path of this dissertation.

For Professor David Taylor and the AIFMRM staff as a whole. I could never thank you enough for the endless support and opportunities you provide your students.

In loving memory of Dr Matthew Wagener
Wish you were here
1995 – 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Direction and aim of the dissertation

At an introductory modelling level, market dynamics generally incorporate movement governed by some continuous Brownian motions. More advanced market models as of Merton (1976), may recognise financial markets contain evolution that is not akin to continuous dynamics. Market models of this kind are named jump models. The market models we will simulate are known as jump diffusion models. These are much the same as continuous Brownian motion models with the inclusion of an independent discontinuous jump process. What is rarely observed (in the extreme negative case – market crashes) is unpredictable, near instantaneous, significant asset price movement.

The existence of jumps has been strongly suggested in theoretical and empirical studies of financial markets (Lee and Mykland, 2007). A theoretical motivation for the inclusion of the jump process is it provides excess kurtosis and skewness in simulated return distributions. The same goes for implied volatility smiles — matching market observation (Lee and Mykland, 2007). It is suggested in the findings of Lee and Mykland (2007), that the occurrence of jumps is associated with the arrival of new market information.

The converse of this is more insightful. Assume some form of the efficient market hypothesis, such that an asset price reflects all currently available relevant information. Investors react to information, and information arrives discretely. By this logic, discontinuous "jumps" have to occur.

Developing methods that can identify jumps in time series data is important for understanding when they occur. The arrival of jumps in a market will affect derivative hedging error. As such, detected arrival times of market jumps may then be used as points of portfolio rebalancing. The presence of jumps result in an incomplete market. Hence, the magnitude of the jump will be related to derivative hedging error (Lee and Mykland, 2007).

Aït-Sahalia (2004) suggest it is valuable, in the case of this market incompleteness at the point of a jump, to be able to separate the jump component from the diffusion dynamics. This stems from the idea in the case of a complete market, a derivative may be hedged (Shreve *et al.*, 2004). Therefore understanding the size of a jump that may have occurred is something that can be considered in market risk.

When viewing market data and simulated market models, jumps are difficult to

identify given that the Brownian process is continuous, but observation is always discrete. One challenge is determining whether the main source of variation from one time point to another, in a discrete setting, is due to the continuous Brownian motion or discontinuous jump process. An average to small jump may lead to the same variation to that caused by a volatility burst. A solution to this is to incorporate high frequency data, making it less likely to confuse the diffusion process for the jump process. Essentially by making the observations as continuous as possible, we hope to identify a discontinuous feature more easily. Furthermore we expect the jump process to be able to move the asset price by a much larger margin over a small interval, compared to *regular* dynamics moving the price, making jumps easier to spot. The downside of this applied to real high frequency data is that, there will inevitably be unmodellable market micro-structure noise present. In practice, large jumps are infrequent and in simulation, only smaller jumps are included in the model (Aït-Sahalia and Jacod, 2009).

There are numerous model independent methods available to test for jumps in financial time series. Throughout case studies, where the Lee and Mykland (2007) (LM) method is considered, it is always one of the strongest performers, or comes out as best. This method can detect jump points, as well as magnitude and sign of a detected jump.

It is noted by Eisenstein (2022), that the LM method may fail when the asset price has high volatility. An instance for this would be if we are using a stochastic volatility model, such as Bates or SVJJ (Shreve *et al.*, 2004).

The first aim of this dissertation is to determine if supervised deep learning methods can provide improved accuracy when determining whether a jump did, or did not occur — over the benchmark set by the LM test. The model has to be trained on simulated data, since real data does not have jump points labelled. Testing the model on separate data, we can compare accuracy of the new method and that of LM.

The second aim is to use unsupervised learning methods of anomaly detection, to perform jump tests. The reasoning, jumps can be viewed as anomalies, if we take the diffusion process to be our *regular* dynamics. The superiority of this approach over supervised learning methods is that we do not need to know where jumps happened beforehand, when training. Or worry about the model overfitting the stochastic market models: For it can be trained on real data.

1.2 Existing literature on jump detection with deep learning

Bashchenko and Marchal (2019) implement a supervised learning LSTM model to test for jumps (details of this are given in chapters 4 and 5,) and compare their results to the LM method. In a way this is similar to our first aim. In this paper, their training and testing feature is the LM statistic. Lee and Mykland (2007) already provides us with a very good non-parametric jump test based on this statistic. Where our aim attempts to expand on this, is to implement supervised learning, without the training feature being part of an existing jump test. This will make any per-

formance found more meaningful, since there will be no reliance on existing jump tests.

Another deep learning implementation of jump tests is Au Yeung *et al.* (2020), which use a hybrid model of both predicting asset price movement going forwards (using supervised learning,) and then having another supervised learning model component to predict jumps. They use "real" data for training, by artificially including jumps. The analysis done in this dissertation will focus on only jump testing in financial time series, not prediction of future price movements. Despite the fact that all data used in this dissertation is simulated from the models given in chapter 3, if we can achieve the second aim, we should be left with a more elegant way to jump test on real data.

1.3 Structure of dissertation

The body of this dissertation is broken up into six chapters. In chapter 2 the structure of some market models is described, as well as motivation for the inclusion of jumps in these types of models. The chapter ends with giving the three market models used in our data generation.

Chapter 3 outlines the benchmark jump test used in the analysis. For this method we will note how in reducing an asset return by its bipower variation, a suitable feature for hypothesis testing is created.

The tools needed to achieve the first aim of this dissertation are contained in chapter 4, but more intuition is given in chapter 5 when discussing supervised versus unsupervised learning. The former chapter should be informative on the key components of deep neural networks used in the experimentation. The latter chapter introduces the autoencoder based jump test.

The jump tests are evaluated as binary classification, whether a jump happened or not. We note in chapter 6, predicting a jump to have occurred ought to be far less common than predicting one not to have occurred. This result arises from assuming jumps to be infrequent. Evaluations are done using metrics best suited to imbalanced data sets.

Experimentation and results comprise chapter 7, which should be seen as the natural culmination of chapters 2-6. Here both aims are achieved, as well as specification of methodology allowing readers to recreate the work or experiment further.

Chapter 2

Market model set-up

2.1 Motivation for jump model

A market model can be given by tuple:

$$\mathcal{M} = (\Omega, \mathcal{F}, \mathrm{P}, (\mathcal{F}_t)_{t\geq 0}, (\mathbf{S}_t)_{t\geq 0}),$$

where $(\Omega, \mathcal{F}, \mathrm{P})$ is the real world probability space, with filtration $(\mathcal{F}_t)_{t\geq 0}$, satisfying the usual conditions. S_t denotes a vector of $N+1$ assets, satisfying the properties of an adapted càdlàg semi martingale (Shreve *et al.*, 2004).

Before the Black Monday market crash of 1987 (see figures 2.1, 2.2) options pricing naïvely relied on calibrating the Geometric Brownian Motion Model:

$$\frac{dS_t}{S_t} = \mu\, dt + \sigma\, dW_t \quad \text{, where } W_t \text{ is the diffusion component.}$$


"All the News That's Fit to Print"

The New York Times

Late Edition

VOL. CXXXVII . . . No. 47,298 NEW YORK, TUESDAY, OCTOBER 20, 1987 50 CENTS

STOCKS PLUNGE 508 POINTS, A DROP OF 22.6%; 604 MILLION VOLUME NEARLY DOUBLES RECORD

U.S. Ships Shell Iran Installation In Gulf Reprisal

Offshore Target Termed a Base for Gunboats

A Huge Blow to the Five-Year Bull Market

Dow's Record Fall

WORLDWIDE IMPACT

Frenzied Trading Raises Fears of Recession — Tape 2 Hours Late

Does 1987 Equal 1929?

Fig. 2.1: Market position at time of Black Monday crash, source: New York Times, clear structure not akin to a continuous model.

Fig. 2.2: Crash of the Dow Industrial Average in October 1987. Source: CNBC, "Remembering the Crash of '87" by Robert Hum on 19 October 2012.

For risk neutral pricing on a European style derivative, we would have $\mu = r - q$, where r is the continuously compounding interest rate, and q the underlying asset's dividend yield. In this context, one would only have to fit one parameter (volatility σ) to the market. It became clear that constant σ fails to describe the realised market. An observation was that options with well in the money strikes must be priced with larger σ. This contributes to the realised volatility skew.

After this financial event, practice and academia expanded from the Black and Scholes (1973) model. Two ways this was done: Modelling the volatility as a local function of price and time, or having the volatility be a coupled stochastic process.

Neither of these models would be suitable to price options with a short maturity. Short term distribution skew in returns and volatility will never be consistent to realisation, since over an infinitesimal period, the security cannot move far enough away from the present level (Shreve *et al.*, 2004). Gatheral (2011) note that including jumps has little impact when matching the volatility surface of the model to observed option prices, when the maturity is long. Interpreting this, over a long period of time, it is probable that the asset price will move significantly from the point the contract was initiated, by means of diffusion. If we wanted to price a short term option with a stochastic volatility model, we would have that the payoff be near Normally distributed. The probability that the diffusion process could lead to a skew would be infinitesimal.

2.2 Jump-diffusion models

A **jump process** is defined to be adapted to the filtration and right continuous.

The rest of the chapter proceeds as follows. We present the Bates (1996) model, and how it will be used in data generation. Then presenting two more jump-diffusion models, we can infer how they can be simulated by what has been dis-

cussed for this first market model.

Bates (1996) model can be seen as a Heston (1993) stochastic volatility model with the inclusion of a jump process. Under this framework, asset price dynamics contain a jump process, and the volatility is a stochastic process.

$$\begin{aligned} \frac{dS_t}{S_t} &= \gamma\, dt + \sqrt{V_t}\, dW_t^S + dJ_t^S\,, \\ dV_t &= \kappa\,(\theta - V_t)\, dt + \sigma_V \sqrt{V_t}\, dW_t^V\,. \end{aligned} \tag{2.1}$$

The jumps process here is modelled by a Poisson process N_t with mean λ_J:

$$dJ_t = \sum_{k \leq N_t} \Delta J_k$$

where ΔJ is related to the Lognormal distribution:

$$\log(1 + \Delta J) \sim \mathcal{N}(\mu, \delta^2).$$

In the model framework, we can impose correlation between the diffusion processes:

$$d[W^S, W^V] = \rho\, dt.$$

We have that:

$$\gamma = r - \lambda_J\,(e^{\mu + \frac{1}{2}\delta^2} - 1).$$

Our data generation uses the initial condition $V_0 = \theta$. In practice, V_0 would be calibrated to market observation, by making a volatility estimate. This would typically be done by using the variation estimates described in chapter 3.

An example of an extension of this model is to include an independent jump process to the stochastic differential equation for the volatility. All market models will have their advantages and disadvantages when describing real market dynamics. With the use of a deep neural network in mind, simulating data from a variety of models is advantageous. Data from multiple different models should prevent the neural network overfitting to some training data set, or being optimised to only one market model. The effect of this is compounded as we sample the model parameters randomly when generating a sample path by randomly selecting a market model. An example is randomly varying κ, θ, σ_V, λ_J, μ, δ and ρ in the Bates (1996) model. This together with independent diffusion and jump processes, allows for vastly different paths when generating different data sets from the same underlying process.

Initialising S_0, the model can be simulated for any length of time, for a specified time interval dt. To use deep learning to perform jump tests, we must keep track of where in the time series, jumps occurred. This is done by saving a vector of the same length as the time series, where the entry is zero if no jump occurred. For the purpose of noting whether or not jumps are present, we can register this event by the number one. This lends itself to a logical input I to the neural network. This avoids errors when detecting the size of the jump. This can be considered to be the dependent variable in the supervised learning aim. For both aims we

will need these labels for evaluating the jump tests, including the LM benchmark comparison.

We can simulate market models using an Euler update scheme. For mathematical ease, we work with the log price process. The first step of this is taking the second order expansion. In the case of Bates (1996) model:

$$\begin{aligned} d\log S_t &= \frac{1}{S_t}\, dS_t - \frac{1}{2}\frac{1}{S_t^2}\, dS_t^2 \\ &= \gamma dt + \sqrt{V_t}\, dW_t^S + dJ_t - \frac{1}{2}\, V_t dt \end{aligned}$$

$$\implies \log S_t \approx \log S_{t-1} + \gamma\, dt + \sqrt{V_{t-1}}\, \Delta W_t^S + dJ_t - \frac{1}{2}\, V_{t-1}\, dt.$$

The volatility component is updated similarly, enforcing it stay positive,

$$V_t \approx \max(0, V_{t-1} + \kappa\,(\theta - V_{t-1})dt + \sigma_V\, \sqrt{V_{t-1}}\, \Delta W_t^V).$$

Other market models that will be used to generate data are the Merton (1976), and SVJJ (Duffie *et al.*, 2000) model. The Merton model can be seen as the Bates (1996) model without the stochastic volatility component, hence can be viewed as:

$$\frac{dS_t}{S_t} = \gamma\, dt + \sigma_S dW_t^S + dJ_t^S. \tag{2.2}$$

Here σ_S will be a randomly varied parameter between different generated data sets. Similarly, the SVJJ model can be seen as similar to the Bates (1996) model, where the stochastic volatility includes a jump process too. The SVJJ model is given by:

$$\begin{aligned} \frac{dS_t}{S_t} &= \gamma\, dt + \sqrt{V_t}\, dW_t^S + dJ_t^S\,, \\ dV_t &= \kappa\,(\theta - V_t)\, dt + \sigma_V\, \sqrt{V_t}\, dW_t^V + dZ_t. \end{aligned} \tag{2.3}$$

In this case, we will have

$$\gamma = r - \lambda_J \left(\frac{e^{\mu + \frac{1}{2}\delta^2}}{1 - \rho_J\, \mu_V} - 1 \right),$$

where ρ_J, μ_V will be varied independently, ρ_J can be seen as the correlation between jump effect in the asset price and volatility. The jump effect in the volatility process follows the exponential distribution:

$$Z_t \sim \text{Exp}(\mu_V).$$

Then conditioned on Z_t, the jump distribution in the price is given by:

$$\log(1 + \Delta J)|Z_t \sim \mathcal{N}(\mu + \rho_J\, Z_t, \delta^2).$$

Chapter 3

Benchmark jump test

3.1 Defining the test statistic

While the LM method will be the benchmark against using a deep neural network, the insight from this chapter is beneficial throughout the dissertation.

LM note that we can not formulate a null-hypothesis test to see if a jump occurred at t_i (a discrete data point / observation) based on the return r alone, where:

$$r_i = \log\left(\frac{S_{t_i}}{S_{t_{i-1}}}\right).$$

In real life observation and model simulation, we generalise observations to a data set of $n+1$ points:

$$0 = t_0 < t_1 < \cdots < t_n = T.$$

In our data generation, asset price will be simulated through equidistant points in time. The time between observations is given by:

$$\Delta t = t_i - t_{i-1}.$$

In the case of non-equidistant price recording of real data; for the same framework to hold, we require

$$\max_{i \leq n} (t_i - t_{i-1}) \to 0.$$

In other words, the time between all observations is negligibly small.

Since we can only do analysis in discrete time, a large volatility burst may lead to a large return that one would expect from a jump. Having a higher interval frequency makes it less likely the price can change significantly due to diffusion over the interval. We try reduce the effect of the volatility component by dividing returns by realised variation.

When S_t is assumed to satisfy the conditions described in chapter 2, the quadratic variation of process $(\log S_t)_t$ can be estimated at time t_i as:

$$[\log S]_{t_i} \approx \sum_{j=2}^{i} \log\left(\frac{S_{t_j}}{S_{t_{j-1}}}\right)^2.$$

Letting $\Delta t \to 0$, the approximation sign is replaced by an equals sign.

For our purpose of having two underlying processes, the diffusion being continuous, and jumps being discontinuous; the quadratic variation can be decomposed to reflect the variation effect of these components:

$$[\log S]_t = [\log S^c]_t + [\log S^d]_t.$$

With this in mind, the realised bipower variation is defined as:

$$\text{BPV}[\log S]_{t_i} \approx \sum_{j=3}^{i} |\log\left(\frac{S_{t_j}}{S_{t_{j-1}}}\right)||\log\left(\frac{S_{t_{j-1}}}{S_{t_{j-2}}}\right)|$$

Barndorff-Nielsen and Shephard (2004) show that in the limit $\Delta t \to 0$,

$$\text{BPV}[\log S]_t = c^2[\log S^c]_t.$$

The constant c is given as

$$c = \mathbb{E}[|Z|] = \frac{\sqrt{2}}{\pi} \text{ , for } \quad Z \sim \mathcal{N}(0,1).$$

The jump component vanishes in this limit of time between observations, when calculating BPV. To explain this intuitively, the diffusion process is continuous, and is therefore a component of every observation. Jumps are assumed to be a rare discontinuous occurrence. For $\Delta t \to 0$, it is impossible for two consecutive points to both contain a jump. The diffusion is modelled on a subset of $\mathbb{R}$, which uncountable and infinite. The number of jump occurrences however is finite. We therefore have that no consecutive returns will both contain a jump. The expected value of a return governed by diffusion only, will be zero.

This framework leads to the hypothesis testing in the next section.

The LM test statistic $\mathcal{L}$ considers the realised bipower variation $\hat{\sigma}$ over the time window period K:

$$\mathcal{L}(i) = \frac{\log(\frac{S_{t_i}}{S_{t_{i-1}}})}{\hat{\sigma}(t_i)} \text{ , where}$$

$$\hat{\sigma}^2(t_i) = \frac{1}{K-2}\sum_{j=i-K+2}^{i-1} |\log\left(\frac{S_{t_j}}{S_{t_{j-1}}}\right)||\log\left(\frac{S_{t_{j-1}}}{S_{t_{j-2}}}\right)|.$$

Over a high frequency observation rate, we assume log returns to be small at points where there is only Brownian motion driving the movement. This is based on observation, and is thus a characteristic of the models simulated from chapter 2. Most log returns r_i will in absolute terms, be near zero. In the calculation of $\mathcal{L}$, we are dividing r_i by $\hat{\sigma}(t_i)$, which has the same magnitude as r_i. Dividing by the number of observations in the summation, and taking the square root of $\hat{\sigma}^2(t_i)$ above assures this. In a way this is a type of normalisation, in that the expected variation order of r_i was < 1, and $\mathcal{L}$ is 1. This increase of scale will be important for the design of jump tests introduced in chapter 5.

3.2 Test statistic null hypothesis

Building on the previous section, it is assumed that between two successive observations, the variance due to either, or both drift and diffusion, is bounded. Under the assumptions of high frequency observations, we can ignore the drift effect in the statistical test. To justify this, in the dynamics; drift, diffusion and the jump process have magnitudes of order

$$O(\Delta t) < O(\sqrt{\Delta t}) < 1 \text{ respectively.}$$

Establishing the notation of observation labels $\bar{A}_n = \{1, 2, \cdots n\}$, we let the window size

$$K = O(\Delta t^{\alpha}).$$

Where $\alpha \in (-1, -\frac{1}{2})$, Lee and Mykland (2007) prove for time intervals $(t_{i-1}, t_i]$ containing no jump; as $\Delta t \to 0$, $\mathcal{L}(i \in \bar{A}_n) \to \frac{U_i}{c}$. Here U_i is a standard Normal random variable. This is due to the diffusion component evolving successive time intervals t_{i-1}, t_i.

$$\text{I.e. } (W_{t_i} - W_{t_{i-1}}) \sim \mathcal{N}(0, \sqrt{\Delta t}).$$

$$\text{Hence, } \frac{(W_{t_i} - W_{t_{i-1}})}{\sqrt{\Delta t}} \sim \mathcal{N}(0, 1).$$

If there is a jump in the interval $(t_{i-1}, t_i]$, the second theorem proves that the asymptotic behaviour of $\mathcal{L}(i)$ is dependent on the jump component in the interval. This is in addition to the convergence property in their theorem 1.

The theoretical component of the LM paper, leading to the test selection threshold, comes from their lemma 1,

$$\text{as } \Delta t \to 0, \frac{\max_{i \in \bar{A}_n} |\mathcal{L}(i)| - C_n}{S_n} \to \xi.$$

$$\text{Here, } C_n = \frac{\sqrt{2 \log n}}{c} - \frac{\log \pi + \log(\log n)}{2c\sqrt{2 \log n}}, \text{ and } S_n = \frac{1}{c\sqrt{2 \log n}}.$$

In the limit, ξ is a Gumbel random variable.

$$\text{I.e. } \mathbb{P}(\xi \leq x) = \exp(-\exp(-x)),$$

is the cumulative distribution function. The proof for this in LM is taken from Galambos (1978).

As a jump test, the null hypothesis is that no jump occurs in $(t_{i-1}, t_i]$. The null hypothesis is rejected if for significance level α^*, we have

$$\frac{|\mathcal{L}(i)| - C_n}{S_n} > -\log(-\log(1 - \alpha^*)).$$

While there are numerous jump detection tests available, that are not machine learning based; the LM test is well regarded as superior in case studies Eisenstein (2022), Dumitru and Urga (2012) and Hong and Zou (2015).

Chapter 4

Composing deep neural networks

4.1 Deep learning methods

This section attempts to describe the neural network architecture necessary for the analysis performed, from the ground up. Explanations of the elements described in this section may be found in most good deep learning textbooks, such as Bishop and Nasrabadi (2006) or Nielsen (2015).

Stripped down, a deep neural network is a sequence of neural layers connected by mathematical operations. In turn, a single layer in the network may contain multiple neural units, the fundamental component of this branch of data science.

A neural unit typically takes the scalar product of a real valued vector input $\mathbf{x}$ with the unit's weight vector $\mathbf{w}$. The output z then includes an additional bias component b.

$$z = \mathbf{x} \cdot \mathbf{w} + b$$

Non-linearity is then introduced by applying a suitable activation function f to the output of the neural unit. The output of the neural unit is then:

$$y = f(z).$$

Examples of activation functions are sigmoid, tanh and ReLU. The implementation of deep learning solutions will generally need to be bespoke to the goal at hand. The rectified linear units function ReLU allows for the output of the neural unit to be almost linear.

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

$$\sigma(z) = (1 + e^{-z})^{-1}$$

The sigmoid $\sigma(z)$ and tanh functions share similar desirable properties. Firstly, both are differentiable, necessary for using gradient decent. Allowing inputs of $\mathbb{R}$, the input of hyperbolic tangent is "squashed" between minus one and one. Similarly, the sigmoid's range is concentrated between zero and one. This behaviour (visualised in figure 4.1) makes σ useful for classification-type problems. Between different layers in a deep network, the same activation function need not be used. For regression-type problems, one would frequently use a linear or ReLU activation function for the output layer. In chapter 7 activation functions are chosen specifically around what is being attempted.

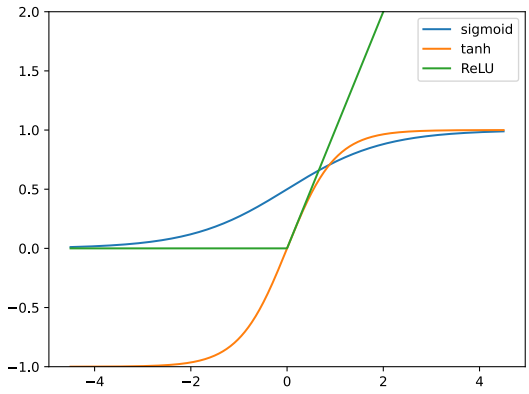

Fig. 4.1: Plot showing the properties of the activation functions described.

Each neural unit in a neural layer will then have a single output. A fully connected network may have each neural unit in a subsequent hidden layer be connected to every output of the units in the previous layer. This increases the complexity of the solution. A more complex problem will require a deep neural network with more parameters.

In financial time series, observed data is significantly complex, containing much micro-structure noise, this can be seen in figures 2.1, 2.2, even in isolation of the market crash. The market models used to describe this (given in chapter 2) are also complex, despite not modelling any micro-structure. The price movements are Lognormally distributed, with the inclusion of a jump process. It is expected that the choice of hyperparameters in deep learning models, will be crucial for the model's ability to fit the dynamics of the input data. This will be analysed in chapter 7.

A neural network is trained to fit the data by applying a loss function to the final output and target input. A minimisation routine is applied to the network, adjusting weights and biases; this attempts to have the final output match some desired target.

Bashchenko and Marchal (2019) use supervised classification of whether a jump occurs. The independent response variable takes on a discrete value. Using a distance norm as a loss function here would be disadvantageous and unstable. Here a probability distribution-type activation functions becomes useful. The jump labels provide us with an absolute (target) probability of whether a jump occurs ($p \in \{0, 1\}$.) The terminal output of a point in a time series $\hat{p}$, where the activation function is σ, will then be a value lying between zero and one. For a loss function, we would then use something like the cross-entropy loss,

$$C_{\text{CE}} = -\sum_{i=1}^{N} \left(p_i \log\left(\hat{p}_i\right) + (1 - p_i) \log\left(1 - \hat{p}_i\right)\right), \text{ for a training batch of size } N.$$

The properties of this allow for more stability when fitting the network to the data.

This holds even when the initial state leads to loss far from a global minimum. Alternatively, when modelling a continuous response variable y, for a regression type task, one uses a loss function like the mean-square error (mse,)

$$C_{\text{mse}} = \frac{1}{2N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2.$$

Deep learning approaches are amenable largely due to the efficiency and robustness of the routines used to minimise the loss function for an epoch of a training set. Epoch number refers to the number of times a given batch of data trains (the loss function being minimised) a network. Methods that do this employ a gradient descent approach, a multi-dimensional Newton-Raphson type operation.

For one hidden layer, performing this is rather straight forward. We can refer to the gradient operator $\nabla_{\mathbf{W}}$ as a vector of partial derivatives, each element corresponding to the weight w of a neural unit in a layer. To minimise the loss function C, we would use the chain rule by first differentiating the loss function with respect to the activation function f; and then applying the gradient operator. One can think of this gradient (positive) to be the most direct path (in terms of the parameters) the loss function is from zero. The weights $\mathbf{w}_0$ can then be updated as,

$$\mathbf{w}_1 = \mathbf{w}_0 - \lambda\nabla_{\mathbf{W}}\frac{\partial C}{\partial f}, \text{ where } \lambda > 0 \text{ is the learning rate.}$$

Learning rate ($\lambda < 1$) lessens the update of the weights, as a global minimum of the loss function is approached. It is recommended that on each successive training of the neural network, the learning rate is decreased so that the global minimum is not over-shot. This is to allow a stable minimisation process.

To do this efficiently across multi-layered neural networks, *back-propagation* algorithms are used to compute the gradients from layer-to-layer using the chain rule. Starting with the terminal output, and differentiating backwards, the calculations to update the weights in earlier layers will be dependent on the partial derivatives with respect to weights in later layers. Partly this is feasible because of efficient caches of these gradients. For most deep neural networks in practice, this would still be too slow with the sheer number of weights and connections in the model.

Another way in which gradient descent is sped up in implementation, is the use of *stochastic gradient descent*. Instead of calculating gradients based on a full data set, random smaller samples are taken to estimate the gradient in a Monte Carlo type manner. As such, the gradients are estimated with an average.

4.2 Common network architectures

The architectures described here will be used to build the models described in chapter 5, which will in turn be used to achieve the first and second aim of this dissertation.

4.2.1 Feed-forward networks

Firstly describing a dense, fully connected *feed-forward* neural network: The output of any layer is the result of mathematical operations on previous layers only. These will include matrix multiplication, weight bias, and use of an activation function. A sketch of this is given in figure 4.2

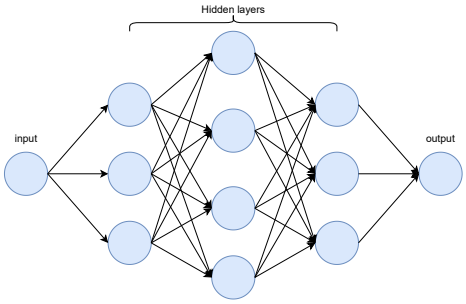

Fig. 4.2: Sketch displaying a low dimensional fully connected feed-forward neural network. Here there is a single input and output, for instance the jump detection of a single point. The blue dots indicate nodes, and the black arrows indicate connections, which happen via matrix operation.

4.2.2 Recurrent neural networks (RNNs)

A common and useful feature to add to a dense network is a cyclic layer – leading to a recurrent neural network (RNN.) Simple RNNs of this type may also be known as Elman Networks (Elman, 1990). From the activation function on the output of a dense layer (weights $\mathbf{W}$,) which feeds into the next layer, the same output cycles back into the same layer (via a recurrent layer weights $\mathbf{U}$) to be part of the next input to the network. In time series analysis, this is thought of as an encoded memory of the previous inputs.

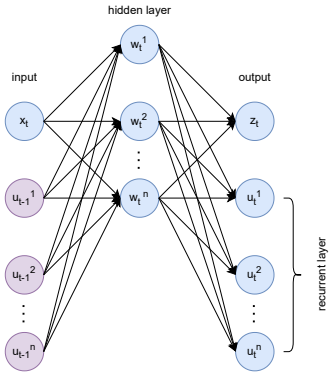

Fig. 4.3: Illustration of Elman type neural network, again for a single input. Here the hidden layer as an arbitrary number of neural units, with the same number for the recurrent layer.

With the inclusion of this advantageous aspect arises a problem. Training via back-propagation, if each output has dependence on the previous output, all the gradients (multiplied) may converge to zero, meaning no minimisation. If we have time series $\mathbf{x}_t$, the element of the time series / observation x_t will be the input for the hidden layer $\mathbf{W}_t$, along with the output of the recurrent layer $\mathbf{U}_{t-1}$. The output of the recurrent layer is in turn a function of the previous hidden layer $\mathbf{W}_{t-1}$. Since our first aim given in chapter 1 is to classify whether a jump happened or not, we will use the sigmoid activation function. The beginning of the problem, when calculating the gradient in back propagation, we need to evaluate the derivative of the sigmoid function. Seeing in figure 4.4, the derivative has a maximum of $\frac{1}{4}$. Further contributing, the weights are typically initialised from the standard Normal distribution. This requires a lot of repeated multiplication of numbers that may well be (in absolute terms) < 1. This can cause a *vanishing gradient*.

One of the solutions to this common problematic characteristic of RNNs is discussed in the next subsection.

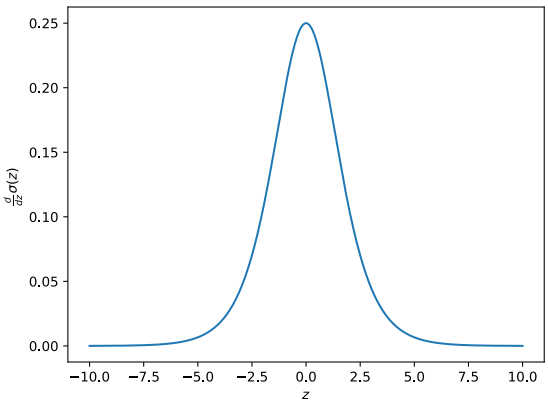

Fig. 4.4: Graph of the derivative of the sigmoid activation function, which will be part of the calculation in back propagation.

4.2.3 Long-short term memory (LSTM)

The Long Short-Term Memory neural network (Hochreiter and Schmidhuber, 1997) lessens the chance of a vanishing gradient, by encoding new relevant information in the time series, and removing the encoding of information that is no longer needed. This is done by including three *gated* RNN type structures on top of a standard RNN structure. Namely, the forget, add, and output gates.

Here we denote the baseline RNN computation by $\mathbf{r}_t$,

$$\mathbf{r}_t = \tanh(\mathbf{U}_r \mathbf{h}_{t-1} + \mathbf{W}_r \mathbf{x}_t)$$

As before, $\mathbf{x}_t$ represents the vector of inputs corresponding to time t. These may be single valued. The matrices $\mathbf{W}_r$,$\mathbf{U}_r$ are the weights of the input and recurrent operations, respectively. LSTMs are often used as hidden layers in deep learning;

hence the output of the LSTM, for the current time labeling is $\mathbf{h}_t$. To train the network, $\mathbf{h}_0 = \mathbf{0}$ is used as an initial condition.

The forget gate computations are defined as:

$$\mathbf{f}_t = \sigma(\mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{W}_f \mathbf{x}_t)$$

$$\mathbf{k}_t = \mathbf{m}_{t-1} \odot \mathbf{f}_t.$$

The matrices $\mathbf{W}_f$,$\mathbf{U}_f$ are the same structure as before, but correspond to parameters of the forget gate. The vector $\mathbf{m}_t$ can be thought of as a memory vector, as with the hidden state, it is given a zero initial condition. The Hadamard product operation $\odot$ is element wise multiplication of two vectors of the same shape, where the output is a vector of the same shape. This process is to remove encoded information, that the *optimal* network does not need.

Additionally, the LSTM *optimally* computes which encoded data is correlated with the output of the main RNN component $\mathbf{r}_t$,

$$\mathbf{a}_t = \sigma(\mathbf{U}_a \mathbf{h}_{t-1} + \mathbf{W}_a \mathbf{x}_t)$$

$$\mathbf{j}_t = \mathbf{r}_t \odot \mathbf{a}_t.$$

Again this is a similar, but a separate structure within the LSTM.

With this calculation, the memory component vector is updated as,

$$\mathbf{m}_t = \mathbf{j}_t + \mathbf{k}_t.$$

This update *closes* the loop on how the optimal encoding for the network is selected, retained, and discarded (Jurafsky and Martin, n.d.).

The output gate structure is again the same structure as the gates before it. This component *optimally* selects the encoding relevant to a current hidden output $\mathbf{h}_t$. This is done by using the memory vector $\mathbf{m}_t$ again, but for the current time point.

$$\mathbf{o}_t = \sigma(\mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{W}_o \mathbf{x}_t)$$

$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh \mathbf{m}_t.$$

The flow of these operations for a single LSTM unit is represented by figure 4.5.

A simple but effective complexity to add to this is the so called Bi-directional LSTM (Bi-LSTM.) In such a model, we would have the same method of data process; but input the data, with time reversed, in addition to what we had before. This can allow for the encoding to have a richer representation of the underlying dynamics.

The use of multiple LSTMs can in principle provide a good temporal view of the input data, as well as prevent overfitting. This is opposed to a fully connected network, where near points would be treated no differently to points further away. Another advantage is that we do not need to have a fixed length input to the LSTM. The problem with this approach is that the number of network parameters needed to achieve this nature will be computationally expensive, especially if it were to handle large time series.

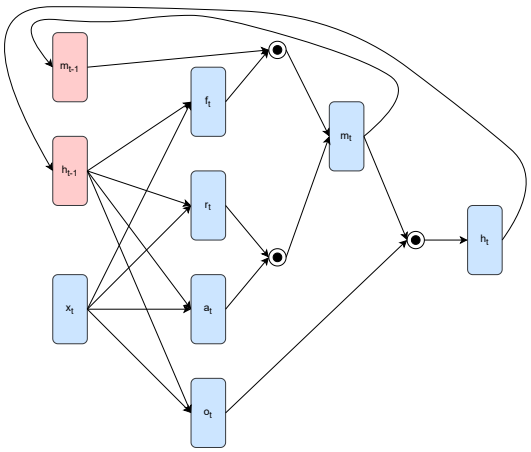

Fig. 4.5: Flow chart of a single LSTM unit. Hadamard product indicated in joins of elements. Activation functions and addition explained above are not shown. This is to show the flow of the input between the different elements in the LSTM.

4.2.4 Convolutional neural networks (CNNs)

If we were to limit ourselves to smaller subsets of data, we may wish to take direct advantage of the temporal nature of the data. This can be done by the use of convolutional architectures in CNNs. Unlike previously described network connections, convolution differs from the fully connected matrix operation connections.

Another way to think of convolution is to think of it as a *smoothing* operation of a time series $x(t)$. This smoothing can be thought of making the time series more continuous with weight/ kernel function w. The kernel can be interpreted as a probability density function (Goodfellow *et al.*, 2016). This is often represented using an asterisk notating the convolution operation:

$$s(t) = \int_{-\infty}^{\infty} x(a)w(t-a)da = (x * w)(t).$$

Naturally, all analysis data will be observations at discrete time intervals, in which case the integral above is replaced by an infinite summation operator. An infinite summation is used for the motivation and proofs of mathematical properties of CNNs, but does not hinder implementation, as the kernel multiplication will be zero in most regions. An example of such a property is that convolution is commutative, i.e.

$$(x * w)(t) = (w * x)(t) = x(t-a)w(a).$$

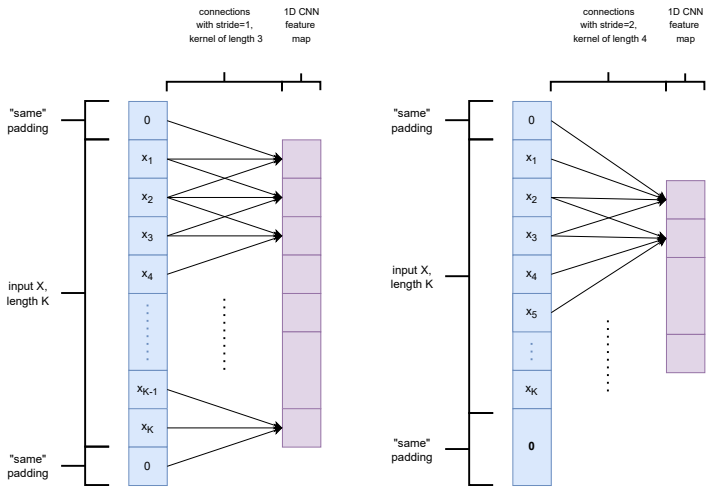

Fig. 4.6: Diagram showing two one dimensional CNN layers, each with one filter – causing a one dimensional feature map.

Convolutional layers contain sparse connectivity as apposed to full connectivity. This is via the kernel having much shorter length (one dimensional case) than the input length. Another mechanism in convolution is the sharing of weights. Instead of having different weights for each element of the input, CNNs use the same kernel everywhere (Goodfellow *et al.*, 2016).

A further mechanism to reduce complexity is to introduce *stride*. What this does is instead of sliding across by one successive element during convolution, the overlap is reduced by skipping $\bar{s}$ elements.

These ideas are displayed in figure 4.5. In both of these CNNs, there is only one - one dimensional feature map – also known as a filter. This would be fine when learning more "simple" dynamics. We expect to need more filters to learn complicated (Nielsen, 2015) market data. In this case the feature map would be a matrix, comprised of multiple single feature maps (shown in figure 4.5.)

By increasing the stride $\bar{s}$, the length of the feature map is decreased. This further reduces model complexity. In figure 4.5 the notion of "same" padding is introduced. It should be clear, in order to use every component of the input in the CNN, in the same way; zero(s) may need to be added the head and tail of the input. The potential need for more than one zero on a given start / end point of a time series is illustrated by the bold **0** in figure 4.5.

The CNN feature map can serve as the input to another neural layer. In some cases, we would like to output a less noisy version of the original input. We can apply a transpose CNN to the output of a CNN, this allows the final output to be the shape of the original input (Goodfellow *et al.*, 2016).

Typically when convolution is implemented in deep learning, it is commonly followed by applying the ReLU activation function (Goodfellow *et al.*, 2016). Thereafter, the output of the activation may go through a *pooling* stage. Similar to convolution, "slide" through the input of this pooling stage, of a chosen pool size, in strides. Similarly, padding may be needed for this to work.

There are three main types of pooling we should be immediately concerned with (Goodfellow *et al.*, 2016). Maximum, minimum, and average pooling. For average pooling, each pool covered per stride, outputs the average value in a given pool. Pooling will become a useful architecture to use in conjunction with CNNs when implementing autoencoders, described in the next chapter.

4.2.5 A note on dropout

Including dropout layers to a deep neural network can reduce the risk of overfitting. Dropout does not stand by itself as a network architecture, but often included to the architectures mentioned above.

Using dropout, a ratio of neural units in a layer is deactivated randomly. The training is then done on this modified network. The weights of the active units are updated as was described before (on this reduced layer (Nielsen, 2015).) Across different training iterations, this deactivation sampling will occur with replacement.

Chapter 5

Anomaly detection review

5.1 Supervised and unsupervised methods

The machine learning techniques mentioned so far are well suited to the problem of *supervised* learning. We can provide a dense neural network f with a feature x derived from the log stock price, and train the network to output whether a jump occurred ($z \in \{0, 1\}$.)

This can be summarised by the mapping

$$z = f(x).$$

Jump processes are part of a mathematical explanation of an observation. To label a historical series of observations, as points that contain jumps or not, would be entirely model dependent. If we assume that jumps are a product of the efficient market hypothesis where investors react to discretely arriving information (described in chapter 1) we may want to collect windows of data around significant observed market spurts and crashes (which we could label as jumps.) This is infeasible as it is unlikely one would be able to collect high frequency data containing enough jumps to train the deep neural network. This would also make it difficult record smaller jumps.

For the first aim given in chapter 1, we therefore need simulated data, i.e. from jump models such as those given in chapter 2. The loss function here would be calculated using the jump labels from simulation, and the output of the deep neural network. Details on the model specification used is given in chapter 7.

Bashchenko and Marchal (2019) showed how a LSTM has better performance than using the statistical threshold derived in Lee and Mykland (2007). This method used $\mathcal{L}$ as its dependent variable, and used supervised learning based on jump labels from simulated data. One might argue that this not a significant finding, since the LM test is already regarded as one of the best jump tests. This is not enough to be completely cynical of the usage of a LSTM in Bashchenko and Marchal (2019). The LM test is done at a single time point. The LSTM (ideally bi-directional) will consider surrounding points too. Ideally then it would recognise points where there is an additional discontinuous process.

There are two obvious improvements to the method described above. Firstly to remove the market model dependencies of using labelled data. We would like to not rely on having labelled data when training our model. The absence of labels

means the deep neural network predictions will need to made from the behaviour of the time series, including the continuity and any lack of continuity in the data.

Secondly, to not be reliant on the magnitude of $\mathcal{L}$ as done by Bashchenko and Marchal (2019). Both log returns and therefore $\mathcal{L}$ given in chapter 3 are signals that vary about zero. As explained in chapter 3, by reducing the log return by the bipower variation, the resulting feature has an increased expected variation order of 1. Over high frequency observation periods, log returns are observed to be very close to zero. For this reason, training under log returns unscaled, the model may approximate most points as zero. Since we are not concerned with formulating a null hypothesis such as that given in chapter 3, we do not need the properties of $\mathcal{L}$ directly. We will compare the model performance of using $\mathcal{L}$ as the training feature, versus using the log return multiplied by a constant scalar. This will be part of the analysis in chapter 7.

The first improvement is a form of *unsupervised* anomaly detection. It is agnostic about how the data is simulated. We seek to have a deep neural network that lets us distinguish between points in a time series that have regular dynamics, and points that may be the arrival of some event. The key difference from before is this approach does not use labels aligning to simulated data. If trained and applied to real high frequency data, model dependency can be reduced or eliminated.

Under the assumptions of chapters 2, 3, jumps occurring are a rare, discontinuous process. In jump-diffusion models, price is primarily driven by diffusion, the arrival of a jump is infrequent, and therefore can be considered an anomaly.

5.2 Unsupervised anomaly detection

5.2.1 Traditional unsupervised methods

Unsupervised outlier detection has been a busy discipline of study in recent years; this is owing to the access to vast data sets in nearly all fields of research and industry. This quantity means it is impossible to label events manually, making supervised learning unfeasible.

Typical unsupervised anomaly detection methods make assessment on points individually. This means the temporal nature of the time series is not considered in these approaches.

5.2.2 Dimensional reduction methods

We want our unsupervised method to capture the nature of the financial time series. At high frequency, we expect mostly continuous behaviour. The jumps presenting in the time series are not common in the market, and thus should not be dominant in the overall dynamics.

The following is supposed to describe dimensionality reduction. We take a time series of observations, and then decompose it in terms of its underlying structure, and then recompose it to be a copy of the original input, based on this reduction of the input. Ideally by doing this, the underlying nature of anomalies in the data would not be adequately explained by the dimensional decomposition. By making

an analysis of the input and output of such a method, one may be able to detect anomalies. This process in the form of an autoencoder is visualised in figure 5.1.

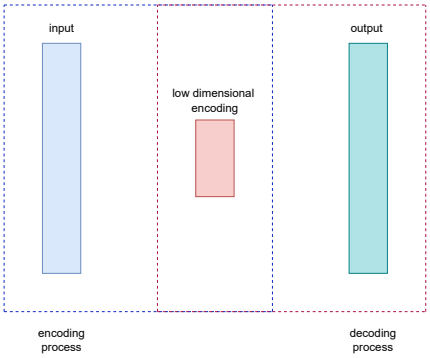

Fig. 5.1: Chart of the encoding and decoding process, no specific structure of the connections is shown, to show the process regardless of the type of neural architecture is used as an autoencoder.

Dimensionality reduction in terms of a one dimensional time series may seem absurd. But in the context of our neural networks, we are using a finite number of parameters to do this encoding. It is thought that it would be best to implement some sort of CNN autoencoder. As discussed in chapter 4, we use a kernel of significantly smaller size than the input length, and then slide this down the time series in strides, using the same kernel for each multiplication operation. We can then reconstruct the data by adding layers, corresponding to the encoding layers. The loss function here would be driven by the reconstruction error of the data, when training (Audibert, 2021).

The effect of pooling after convolution in this sort of autoencoder is twofold. Firstly it reduces the dimension of the encoded data. Secondly, we want our recreated version of the input to minimise the effect of jumps. By using average pooling described at the end of the last chapter, we reduce the influence of large jumps on our encoded state.

5.2.3 Autoencoder training data

We could make the assumption that there are not many jumps, such that the jump process will not be dominant / learnt in the encoded state. If we are training on simulated data, i.e. from the models in chapter 2, we can train the encoded state on diffusion driven data only, and then apply the autoencoder to jump-diffusion data. For training purposes then, the Merton model then reduces to a Black Scholes model, and both Bates and SVJJ reduce to the Heston model. Explained in chapter 7 is how the choice of activation functions can eliminate the assumption that not many jumps are present in the data, such that the model cannot learn to recreate them.

This forms a method that attempts to achieve our second aim. Applying an autoencoder trained in this way, we wish that by inputting jump-diffusion data

to the model, that the final output appear to be more diffusion-like than jump-diffusion-like.

By this, we would like the recreated data to well represent the input data where there are no jumps. The points where there are jumps, we would like the recreated data to be the same direction as the jump, but hopefully of smaller magnitude.

This does complicate the analysis in terms of complexity. On top of having to find suited network architecture's hyperparameters, i.e. kernel size, size of strides, number of filters, and initial learning rate, we now also need to find the best suited cutoff difference ϵ between input x_i and output z_i that classifies a jump. I.e. we would classify a jump to have occurred if:

$$\epsilon < |x_i - z_i|.$$

Chapter 6

Performance metrics

6.1 Binary classification in the context of jumps

To evaluate our jump tests described, we must consider binary classification metrics. To do so in a sophisticated way, we must consider the possible outcomes of a single jump detection measurement. This is represented by the confusion matrix:

		Predicted jump
Real jump	TP	FN
	FP	TN

Tab. 6.1: Confusion matrix

This is easily understood as a way of separating instances of whether a method has correctly or incorrectly evaluated whether a jump has occurred or not. In the confusion matrix, the T and F stand for true and false respectively. Whether the jump occurred or not, is represented by P for positive and, for no jump occurring is given as N for negative.

As is the basis for the autoencoder method, in terms of the confusion matrix; there much fewer jumps/ positives, than there are negatives/ no jump (Merton, 1976). A data set of this sort of characteristic is known as an *imbalanced* data set (Luque *et al.*, 2019). For jump testing, one would say there is low prevalence. I.e. at most points of observation, there will be no jump. This plays a role in evaluating model performance.

6.2 Performance evaluation metrics

The sensitivity metric SNS $\in (0, 1]$ (Luque *et al.*, 2019) is defined as

$$\text{SNS} = \frac{\text{TP}}{\text{TP} + \text{FN}}.$$

Where the entries from the confusion matrix represent the number of these occurrences through our jump testing of time series. It is clear that SNS rewards high true positive rates, and penalises false negatives. The number of true positive measurements would be small, since the number of jumps relative to the number of

observations is small. Furthermore, if we had jump diffusion data of 1000 observations at a frequency of a magnitude in minutes, it would not be unreasonable for the number of false negatives to rival that of number of jumps that occurred. In this example, if for instance there were seven jumps and every jump was detected by a jump detection method, let there be a similar number of false negatives. One would consider this performance more than adequate, since out of the rest ($\sim 99\%$) of the data, the model predicted correctly that no jump took place. This is an example of how data imbalance can lead to evaluation metric bias, since SNS would have a poor score in this example.

Similarly, the specificity metric SPC $\in (0, 1]$ is biased on an imbalanced data set (Luque *et al.*, 2019). This is defined as:

$$\text{SPC} = \frac{\text{TN}}{\text{TN} + \text{FP}}.$$

We would automatically expect a high performance score from SPC, there will be mostly negatives (pure diffusion process) points in the time series. A below adequate jump detection method may receive high score here. We therefore expect false positives to not penalise the metric enough, especially if the number of false positives rivals the number of true positives.

Other biased estimators, which Luque *et al.* (2019) and Powers (2003) say should not be used (as well as those above) on imbalanced data sets, are precision PRC, and negative predictive value NPV. Given as

$$\text{PRC} = \frac{\text{TP}}{\text{TP} + \text{FP}} \quad \text{and}$$

$$\text{NPV} = \frac{\text{TN}}{\text{TN} + \text{FN}}.$$

These metrics have the same ranges as before. Using similar logic by looking at the fraction, in the context of our assumed data set type, it is clear how these metrics may lead to flawed evaluations.

Frequently in anomaly detection research, F1 score Anandakrishnan *et al.* (2018) is a prominent performance metric, which is defined as:

$$\text{F1} = \frac{2 \times \text{PRC} \times \text{SNS}}{\text{PRC} + \text{SNS}}.$$

Luque *et al.* (2019) note that the bias issues of SNS and PRC are not erased by this compound metric. A reason being neither SNS or PRC contain a TN measurement. Thus the F1 score does not consider the whole confusion matrix (Eisenstein, 2022).

Luque *et al.* (2019) and Powers (2003) conclude that combining SNS and SPC, one can obtain null bias. I.e. the measurement spans the confusion matrix. The claimed two best performance metrics of this kind are the *Bookmaker informedness* BM $\in (-1, 1]$ and the so called *geometric mean* GM $\in (0, 1]$. These are given as:

$$\text{BM} = \text{SNS} + \text{SPC} - 1 \text{ and,}$$

$$\text{GM} = \sqrt{\text{SNS} \times \text{SPC}}.$$

In the case of $BM = -1$ would indicate no model performance, $BM = 0$ would indicate high rates of false negatives and false positives, and BM $= 1$ would be perfect prediction (Eisenstein, 2022). Where BM and GM are less informative is including classification error (FN and FP.) Both of these scores therefore indicate mostly classification success.

For our imbalanced data sets, we may wish to analyse results using a metric that has slightly more bias – in exchange for considering model success and failure equally. From the analysis in Luque *et al.* (2019), the Matthew's correlation coefficient MCC $\in (-1, 1]$ has the second lowest bias after BM and GM. They conclude this is the best choice over BM and GM if classification failure should be considered alongside success. It is given by:

$$\text{MCC} = \frac{\text{TP} \times \text{TN} - \text{FP} \times \text{FN}}{\sqrt{(\text{TP} + \text{FP})(\text{TP} + \text{FN})(\text{TN} + \text{FP})(\text{TP} + \text{FN})}}.$$

Here, MCC $= -1$ would indicate no classification performance, where MCC $= 1$ would be perfect performance.

Chapter 7

Experimentation and results

This chapter is dedicated to the fulfillment of the aims laid out in chapter 1. Firstly too show if supervised learning based on jump labels can provide a superior jump test out of sample, when compared to the LM method applied to the same data. Previously Bashchenko and Marchal (2019) have shown this can be done using an LSTM; we would like to show this is still possible having the training feature be something other than the LM test statistic $\mathcal{L}$. We want our jump test methods to stand alone from methods that came before it. The second aim is to remove the dependency of needing jump labels to train a deep learning based jump test. This will be done using autoencoders. The purpose of the autoencoder is to recreate jump-diffusion data as if it were diffusion data, and then be able to detect jumps by analysing the residual between the input and output of the autoencoder.

The machine learning techniques described in chapters 4–5 are implemented in *Python* (3.7.13) with the deep learning package *Keras* (2.10.0,) using *Tensorflow* (2.3.1) as backend. The methods are intended to be implementable by understanding the components from chapters 2–5, and referring to corresponding open source documentation of these packages. All graphical results are generated using *Matplotlib*, most of the mathematics done uses *Numpy*. All computation is done on a 2017 MacBook Pro, relevant hardware being a 2.3GHz Dual-Core Intel Core i5, with 8GB memory, and Intel Iris Plus Graphics 640 1536MB. For all learning, the *Adam* optimiser is used, which is stochastic gradient descent-type algorithm (see explanation in chapter 4.)

7.1 Data generation

The control parameter selection for the market models 2.1, 2.2 and 2.3, that generate the data, is given in table 7.1. The exception is when training the model for the second aim, the jump process is then eliminated, and only include it when testing. Having this variation, there are an infinite number of parameter combinations to create data sets with. Naturally all Brownian motions and Poisson processes will be independent across all generation.

Three market models are used in this dissertation (Merton, Bates and SVJJ,) as separate functions in the code. Each time the function is called, control parameters are randomly selected in their allowed range. The outputs of these functions are an array containing log returns, corresponding K-day bipower variation, and logical

labels of jumps. Testing the neural networks out of sample, we can use the same data to perform the LM test allowing direct comparison. To avoid the repeated training of a particular market model, we randomly select one of these three market models whenever a data set is created.

Basing the neural network feature on log returns and not asset price: we can learn a signal that is about zero. Asset price will have an upward or downward trajectory, over a given range of time. This would be a more difficult learning task.

Each data set generated has the same observation frequency, with the terminal time of half a year. To get the update interval we assume that the market is open for 6.5 hours per day, and there are 250 trading days in a year. Each path of this kind is considered a single data set, i.e. one path would constitute one epoch when training. When testing, the given jump test model is applied individually to multiple different generated paths. Scores from performance metrics are calculated by cumulatively keeping track of jump test predictions and known jump labels from each of our paths.

By the trading time assumptions, we can calculate the update interval for the Euler method as $dt = 2(250 \times 6.5 \times 60)^{-1}$. For the calculation of bipower variation, in accordance with this dt, Lee and Mykland (2007) and chapter 3, we set $K = 273$.

Parameter	Merton	Bates	SVJJ	Variation
S_0	100			-
frequency	2 minutes			-
T	0.5 years (250 trading days, market open 6.5 hours per day)			-
κ	-	10		$[-5, 5]$
θ	-	0.16		$[-0.12, 0.12]$
σ_V	0.1			$[-0.05, 0.05]$
μ	0			$[-0.05, 0.05]$
δ	0.01			$[-0.05, 0.05]$
ρ	-	-0.4		$[-0.4, 0.4]$
μ_V, μ_S	-		0.025	$[-0.025, 0.025]$
ρ_J	-		-0.4	$[-0.4, 0.4]$
λ_J	25			$[-10, 10]$

Tab. 7.1: Table showing the mean model parameters, as well as the random range of component added to the mean.

7.2 Neural network inputs and targets

For our one dimensional arrays of log returns, bipower variation, and logical jump labels, are all of length $24\,102$. Since we have a $K = 273$ bipower variation, the first entry for the log returns and label arrays is the 274^{th} point in the original sequence. Either by calculating $\mathcal{L}$, or multiplying the log returns by 100, we have our training feature.

In the case of supervised learning, the corresponding jump labels will be the

target **training** variable of the deep neural network. To reiterate the goal, we then wish to be able to predict these jumps out of sample, and then score the performance with the known jumps.

For the unsupervised learning autoencoder, the target variable is the training feature. The goal is to learn the temporal nature of the input data while excluding jumps.

The "keras.utils.timeseries_dataset_from_array" function is used to get the training feature and target into the correct shape for our models.

7.3 Supervised learning approach

In the context of binary prediction of jump labels, the supervised model is essentially performing a probabilistic based prediction. Labels being learnt provide an absolute known probability $p \in \{0, 1\}$. All input layer types attempted have inputs described in the previous section with labels as the target. Those attempted are CNNs, RNNs, and LSTMs. After this component, a dropout of 5% is used. Recalling from chapter 4, this deactivates 5% of the neural units in the previous layer for a given training iteration. We proceed to a fully connected layer with single node output. Applying a sigmoid activation function, the terminal output is an estimated probability $\hat{p}$ of jumps occurring. From chapter 4, cross-entropy loss is suited for the minimisation process.

7.3.1 High level experimentation

Wishing to have our jump test be independent of the LM framework, it was attempted to train the model based on unscaled log returns. Since most of the values are almost zero, the model failed to learn any form of structure. With ease the same neural network trained using $\mathcal{L}$. It was thought the reason of this success was the described increase in scale, over the log return. Hence as another exercise, the same model was trained under the log returns scaled by 100, which overcame the problems of using log returns unscaled.

Several types of input layers were attempted to achieve the first aim. CNNs were attempted across different hyperparameter sizes. While these models in most attempts learnt and had reasonable performance out of sample, the accuracy was not as good as the RNNs and LSTMs attempted. The reason why the *vanishing gradient problem* was discussed in chapter 4, was that the simple RNN model is hyperparameter sensitive. It was found that using a Bi-LSTM was very robust and reliable where the number of units was greater than 40.

If a jump has occurred ($p = 1$,) the output of this neural network $\hat{p}$ should if training was successful, have a value that is not in the tail of the sigmoid function range near zero. The obvious choice of selection here is to declare that a jump has happened if $\hat{p} > \frac{1}{2}$. To validate this previous claim, for all hyperparameter combinations tried, decreasing this cut off probability threshold down to 0.2, made no change to the performance of the jump test.

Another notable finding: Batching individual paths generated from table 7.1 into smaller inputs for the model, to aid computation time, deteriorated the model's

ability to learn. This is likely down to the assumption that jumps are rare. On smaller data sets there are fewer jumps to learn from, and the model then failed to identify true jumps. Hence each time the models below are trained, a path of half a year is used.

7.3.2 Bi-LSTM hyperparameter grid search

This subsection validates the choice of hyperparameters with a grid search. The "grid" is ranges for the hyperparameters to take on. By means of nested *for loops*, every combination of hyperparameters creates a different model. For control, the same training data is used (30 random paths from our models.) Each model is then evaluated with the same testing data. This uses 40 random paths generated from our market models. For this grid search, $\mathcal{L}$ is used as the model feature.

It should be noted that using only 30 training data sets and using each for only one epoch, will impact performance of models with certain hyperparameter combination. Consider a model with a slow learning rate $\lambda << 1$ (introduced in chapter 4.) A very small learning rate can mean more accuracy of the model, since there is much higher chance of convergence to a local minimum when training. But there may not be enough training to get close to a local minimum.

For the number of Bi-LSTM units, combinations were chosen from the range $\{50, 60, 70, 80, 90, 100\}$. The initial learning rate of the models were chosen from $\lambda = \{0.0001, 0.001, 0.01\}$. As explained in chapter 4, we will want to decrease the learning rate each time the model is trained, to allow for convergence to local minima. For this reason, also making up our total combinations was the rate at which the learning rate decreased per training iteration. These rates being $\{0.8, 0.9, 0.99, 0.999\}$. An example of this supervised learning model type is given in figure 7.1.

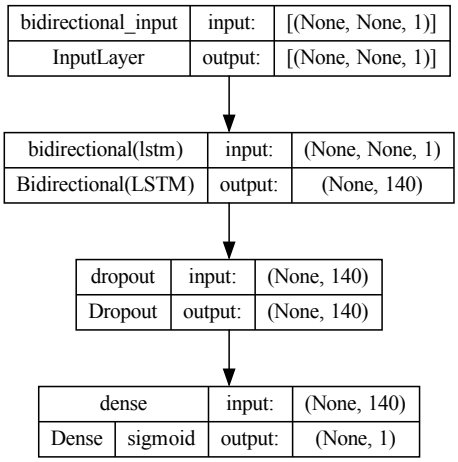

Fig. 7.1: Description of the supervised model described, in this case the LSTM containing 70 units. The 140 units reflects that it is bidirectional. The input shapes "None" correspond to the training feature and target, meaning the model can handle inputs of any batch size.

units	λ	λ shrink rate	F1	BM	GM	MCC	SNS	SPC
50	0.0001	0.999	0.51259	0.34462	0.58704	0.58684	0.34462	1
50	0.0001	0.99	0.40382	0.25299	0.50298	0.50278	0.25299	1
50	0.0001	0.8	0.00043	-0.79283	0.00000	-0.89031	0.20717	0.00000
50	0.001	0.999	0.91401	0.84163	0.91741	0.91733	0.84163	1
50	0.001	0.99	0.91224	0.83865	0.91578	0.91570	0.83865	1
50	0.001	0.9	0.87633	0.77988	0.88311	0.88301	0.77988	1
50	0.01	0.999	0.87482	0.90819	0.95300	0.87528	0.90837	0.99982
50	0.01	0.9	0.90213	0.82171	0.90648	0.90640	0.82171	1
70	0.001	0.999	0.90870	0.83267	0.91251	0.91243	0.83267	1
70	0.001	0.99	0.91342	0.84064	0.91686	0.91679	0.84064	1
70	0.01	0.999	0.91202	0.90331	0.95043	0.91197	0.90339	0.99992
70	0.01	0.8	0.89670	0.81275	0.90153	0.90144	0.81275	1
80	0.001	0.999	0.91047	0.83566	0.91414	0.91406	0.83566	1
80	0.001	0.99	0.91283	0.83964	0.91632	0.91624	0.83964	1
80	0.01	0.999	0.93423	0.89838	0.94783	0.93491	0.89841	0.99997
80	0.01	0.8	0.91518	0.84363	0.91849	0.91842	0.84363	1
90	0.001	0.999	0.91459	0.84263	0.91795	0.91787	0.84263	1
90	0.001	0.99	0.92907	0.86753	0.93141	0.93135	0.86753	1
90	0.01	0.999	0.93482	0.87849	0.93728	0.93669	0.87849	1.00000
90	0.01	0.99	0.93809	0.89043	0.94363	0.93938	0.89044	0.99999
90	0.01	0.9	0.91401	0.84163	0.91741	0.91733	0.84163	1
100	0.0001	0.999	0.75819	0.61055	0.78138	0.78122	0.61056	1
100	0.001	0.999	0.93135	0.87151	0.93355	0.93349	0.87151	1
100	0.001	0.99	0.91342	0.84064	0.91686	0.91679	0.84064	1
100	0.001	0.9	0.88258	0.78984	0.88873	0.88863	0.78984	1
100	0.01	0.999	0.93819	0.88446	0.94046	0.93987	0.88446	1.00000
100	0.01	0.8	0.91165	0.83765	0.91523	0.91515	0.83765	1

Tab. 7.2: Table showing some of the best and worst hyperparameter combinations, for our grid search, over the 40 testing data sets

.

What table 7.2 shows us, is that by increasing the number of Bi-LSTM units, we can marginally increase performance scores in the jump test. It is also clear, by comparing the models with 50 units to those with 100, is that the larger model is less sensitive to hyperparameter choice, for it to have decent performance. It was generally found that models with the smallest initial learning rate $\lambda = 0.0001$, that performance could not be found. This was true even when the learning rate was shrinking by the least amount. Interestingly, across all model sizes, the smallest initial learning rate of $\lambda = 0.01$, with the least shrinkage (0.999) per training data set, seemed to have the best performance. This is likely to be related to the fact that we only trained each model with 30 data sets. Other configurations may have

had better performance with more training, where models with large learning rates may fail to get significantly better with more training. It should be noted that the entries in table 7.2 given in yellow, have perfect specificity SPC. This means that these models are still identifying correctly, all the non jump points, and no false positives. What brings the scores down is that they do not detect many true jumps.

The models produced in this grid search took at maximum, 4 minutes to train.

7.3.3 Direct comparison to the Lee and Mykland jump test

While we may have created a model which appears to have very good scores, it must still be seen how the LM test performs on the same testing data set. Recall from chapter 3, within the null hypothesis test, we have to choose the confidence level α^* of the test. It was found that for all confidence levels, the LM test does not score as high as the top performing Bi-LSTM models given in the previous subsection.

We view the performance of the LM test for a range of confidence levels, on the same testing data set as we had before.

α^*	F1	BM	GM	MCC	SNS	SPC
0.2	0.90691	0.82968	0.91087	0.91079	0.82968	1
0.15	0.90632	0.82869	0.91032	0.91024	0.82869	1
0.1	0.90393	0.82470	0.90813	0.90805	0.82470	1
0.05	0.90153	0.82072	0.90593	0.90585	0.82072	1
0.025	0.90033	0.81873	0.90483	0.90475	0.81873	1
0.005	0.89427	0.80876	0.89931	0.89922	0.80876	1
0.001	0.88877	0.79980	0.89432	0.89422	0.79980	1

Tab. 7.3: Table showing performance of the LM jump test, across a selection of values for α^*

From this we can say that most of the models shown in green in table 7.2, are better than the performance of the LM test (given in table 7.3.) Paying particular attention to the BM, GM and MCC metrics (see chapter 6 on imbalanced data sets,) we can conclude the Bi-LSTM with 100 units, and initial learning rate of 0.01 (reducing by $\times 0.999$ per epoch) has noticeable improvement on the LM test. This margin is significant considering the LM test is considered one of the best jump tests.

7.3.4 Training under scaled log returns

This subsection can be seen as the validation of a hypothesis that has been thrown around through this dissertation. That the benefit of using $\mathcal{L}$ as the training variable is mostly down to the increase in the scale of the expected log return variation.

We are not doing a full grid search as was done for training under $\mathcal{L}$. Instead we just want to see if by training (and testing) the Bi-LSTM under the same data sets, but by using the scaled log returns $100 \times r_i$, we can beat the LM test by a similar margin.

Presented are the results using a Bi-LSTM with 70 units, an initial learning rate of $\lambda = 0.01$, and shrinkage of 0.99. The same testing data sets were used as before.

	F1	BM	GM	MCC	SNS	SPC
Bi-LSTM $(100 \times r_i)$	0.94254	0.89133	0.94410	0.94400	0.89132	1
LM test	0.90691	0.82968	0.91087	0.91079	0.82968	1

Tab. 7.4: Table showing the performance jump testing by means of training and testing under scaled returns.

While we cannot say how much better this way of doing this is, over training under $\mathcal{L}$, we can at least say somewhat. This strong statement is validated by the fact that we have the highest MCC score **so far**, for this data set, without having done a grid search.

7.3.5 Visually analysing these results

The Bi-LSTM used to create the figures in this section has 70 units, with learning rate $\lambda = 0.001$, and shrinkage of 0.99. For purposes of illustration, the varied jump parameter λ_J in table 7.1 is changed to a range of $40 \pm [0, 10]$. The reason being, we will have a higher chance of having false positives for all methods, as well as instances where one test predicts a jump correctly, and the other does not.

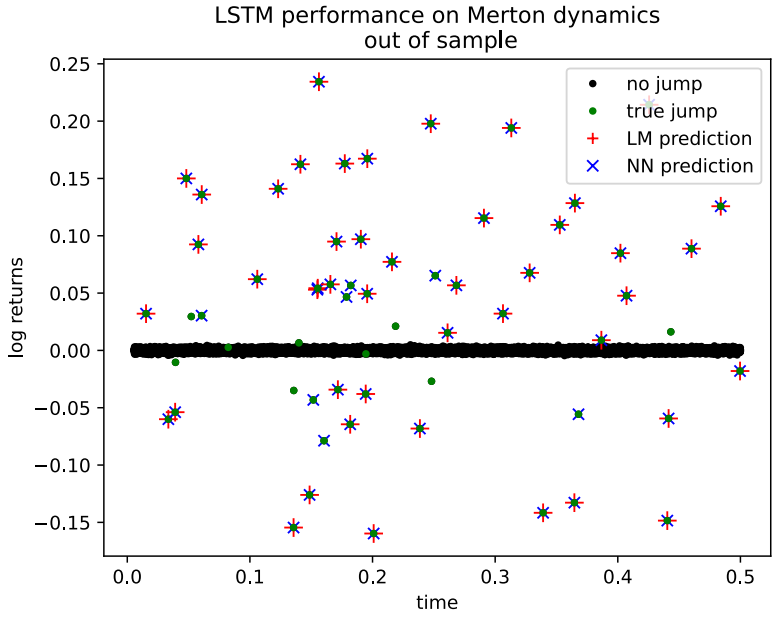

Fig. 7.2: Plot showing performance of the LSTM compared to the LM test, out of sample on Merton model, trained on all models described. Model trained under $\mathcal{L}$.

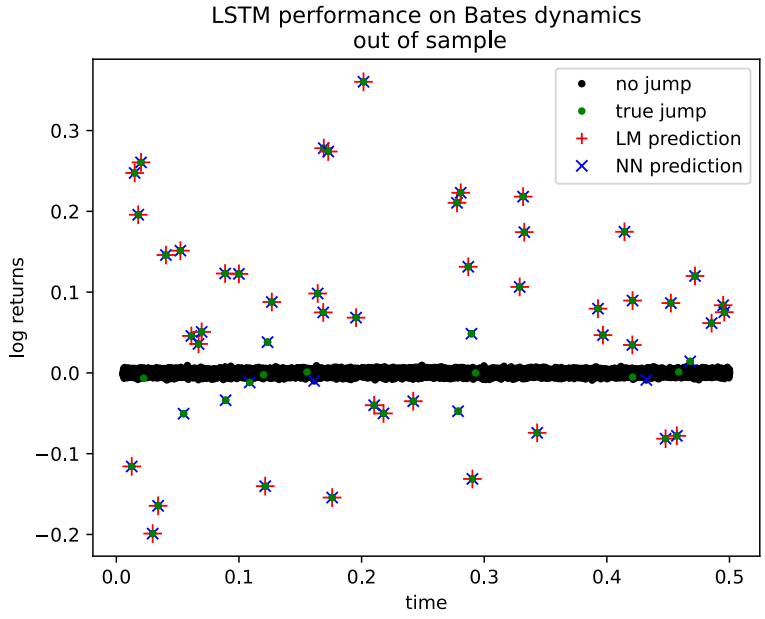

Fig. 7.3: Plot showing performance of the LSTM compared to the LM test, out of sample on Bates model, trained on all models described. Model trained under $\mathcal{L}$.

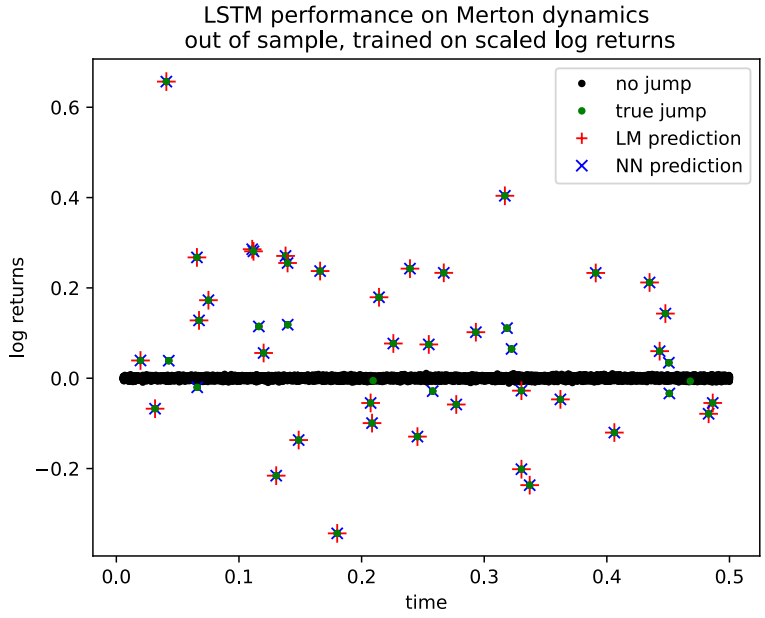

Fig. 7.4: Plot showing performance of the LSTM compared to the LM test, out of sample on Merton model, trained on all models described. Model trained on scaled log returns.

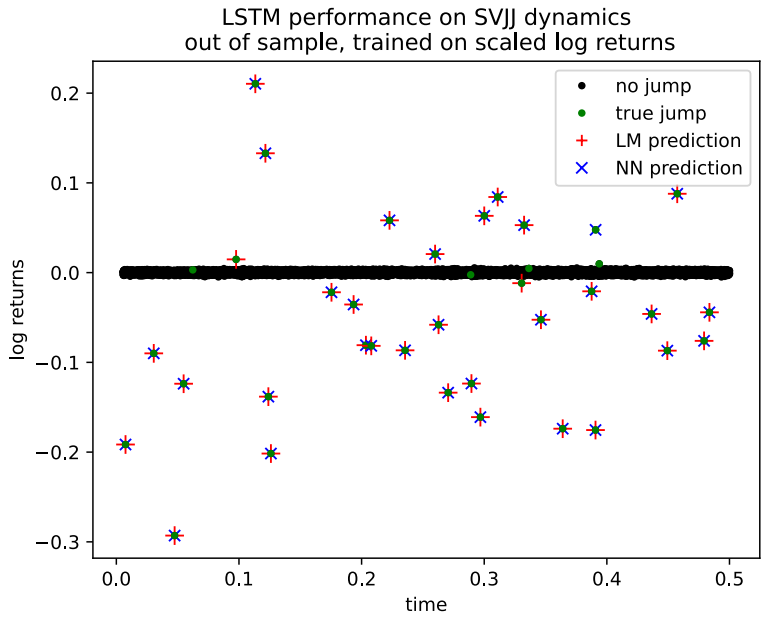

Fig. 7.5: Plot showing performance of the LSTM compared to the LM test, out of sample on SVJJ model, trained on all models described. Model trained on scaled log returns.

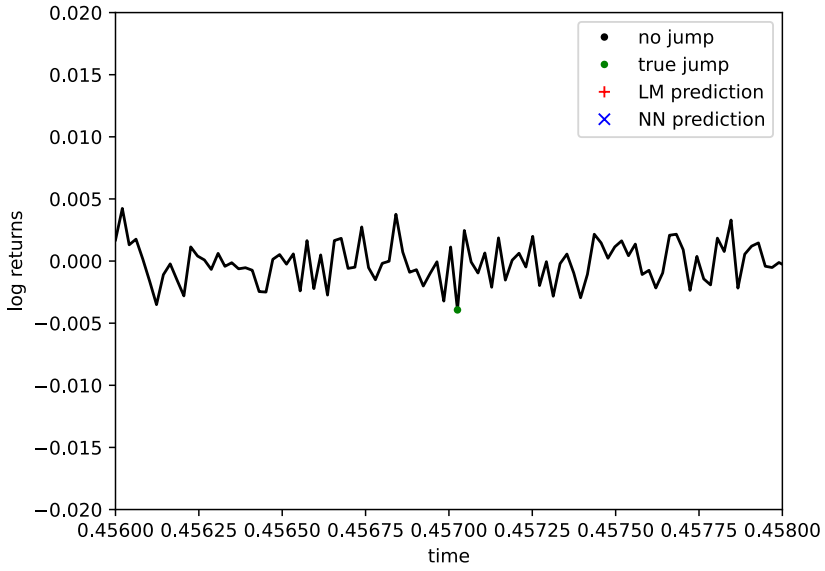

Fig. 7.6: Plot showing an example of a false negative for both models. Clearly an easy situation where the jump is small and has similar nature to the surrounding volatility spikes.

First concerning ourselves with models trained under $\mathcal{L}$, from figures 7.2, 7.3, seen is that smaller jumps have a higher likelihood of being detected using the

neural network. With this though, comes the potential for false positive detection, as can be seen in figure 7.3. Both methods (in the case of the Bi-LSTM – only mostly,) fail to detect very small jumps, as can be seen in figure 7.6.

It is clear from figure 7.2 that the Bi-LSTM is less affected by larger volatility, since log returns are plotted, not $\mathcal{L}$. For some returns where the LM test has a true positive, there exist larger returns where the same test has a false negative.

By training under the log returns scaled by 100, we see in figure 7.4, 7.5 similar performance from the same neural network parameters. What is most interesting from doing this, is that it appears more jumps are again detected using the Bi-LSTM, but for the first time we begin to see jumps that only the LM test managed to predict (see figure 7.5.) This suggests that some level of characteristic described in chapter 3, the Bi-LSTM fails to learn from returns alone.

7.4 Unsupervised learning approach

As mentioned, the paths from market models Merton, Bates and SVJJ are set up as separate functions. These functions are amended to accept an input of 0 or 1. Input 0, the path is driven by a diffusion process only. The Merton model then reduces to the Black Scholes model, and both Bates and SVJJ reduce to the Heston model. Input of 1, the jump-diffusion path is generated as before.

We would like to train a neural network on diffusion data, and the output of the model be a recreated version of the input. We wish this latent state between input and output to have *learnt* diffusion dynamics, such that when the model is applied to jump-diffusion data, the data is accurately recreated in regions of no jumps, but under fits where there is a jump. This can be seen as a regression-type problem, where we only want the fit to be "good" at non-jump points.

Through the initial experimentation phase for this aim, both CNN and LSTM autoencoders were attempted. It was found that successful LSTM autoencoders were too bulky (in terms of number parameters needed) to be considered an efficient jump test. This was true for our individual half year paths being split into batches of 200 points.

Batching is not needed in this second aim, for the autoencoder is CNN based. Recalling from chapter 4, a kernel of some fixed length is chosen, which then moves down the time series in strides of a chosen fixed size, performing the convolution operation. This is intended to be understood by figure 4.6. The huge upside to this is that by using the same kernel, we greatly reduce the complexity when applied to large time series.

The loss function used for this regression-type task is mse.

7.4.1 Choice of activation functions

By using the ReLU activation function in an autoencoder, we run the risk of negative returns being recreated poorly, or as zero. This is illustrated in figure 7.7. Since our CNN layers have multiple feature maps, we could still use ReLU in CNN layers prior to the output layer. In this case, we would expect half the feature maps to

be modelling the negative returns, after training. This creates potential inefficiency in the number of parameters used.

The problem shown in figure 7.7 can be mended by using a linear activation function on the output layer. With this comes another problem. Using ReLU and linear activations functions only, there is no upper bound for the absolute value of the output. What this means is that we could get the model to fit the diffusion data well. However for larger inputs, the autoencoder scales up the output accordingly. Figure 7.8 shows us how these linear-type activation functions lead to a poor autoencoder for us to design a jump test around.

We want the autoencoder to model points of diffusion process, but then be incapable of recreating a large return that cannot be disguised as a diffusion process. By using the tanh activation function, we hold onto the linearity for inputs of smaller returns around zero, but then this activation *squashes* larger values to $\{-1, 1\}$. The desired effect of this is when testing on jump diffusion data, the larger returns in the diffusion training set provide an upper bound when recreating points where jumps occur. This could lead to good recreation of data for smaller jumps, but the supervised methods (and LM) fail in these instances too. The effect of using the tanh activation function is shown in figure 7.9, which can be compared to figure 7.7.

In figure 7.9 we use a tanh activation function on the output as well. But it was found the plot looked the same when a linear activation was used on the output. Using tanh on the output this way is sound since from the data generation parameters in table 7.1, it is near impossible for a return driven by diffusion alone to have a value outside of ± 1. The same is obviously not true if we use $\mathcal{L}$ as the training variable, here we would need to use linear activation on the output.

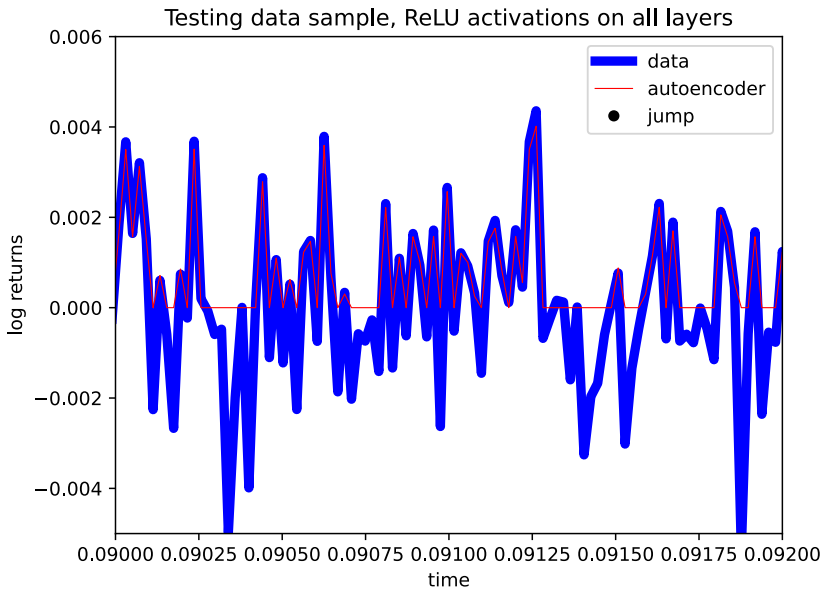

Fig. 7.7: Plot showing the application of a CNN autoencoder using only ReLU activation functions.

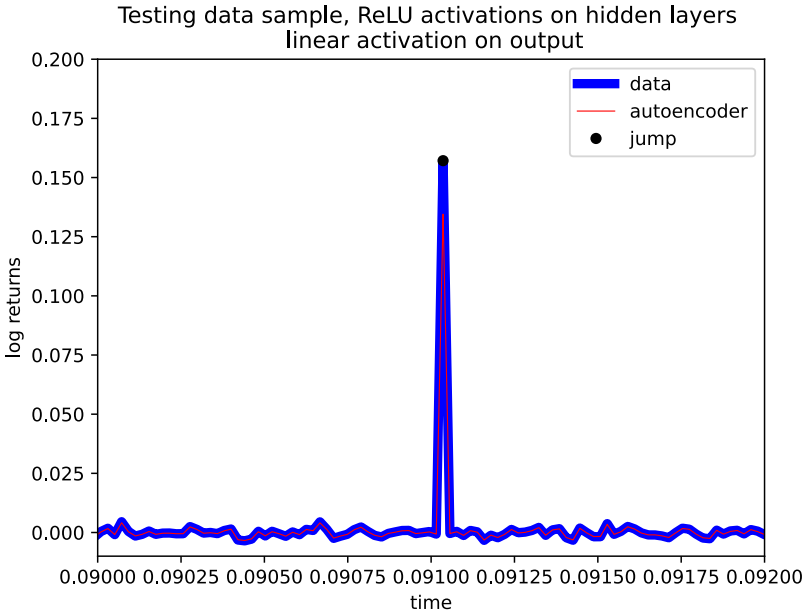

Fig. 7.8: Plot showing the application of a CNN autoencoder using only ReLU activation functions on all hidden layers, and a linear one on the output.

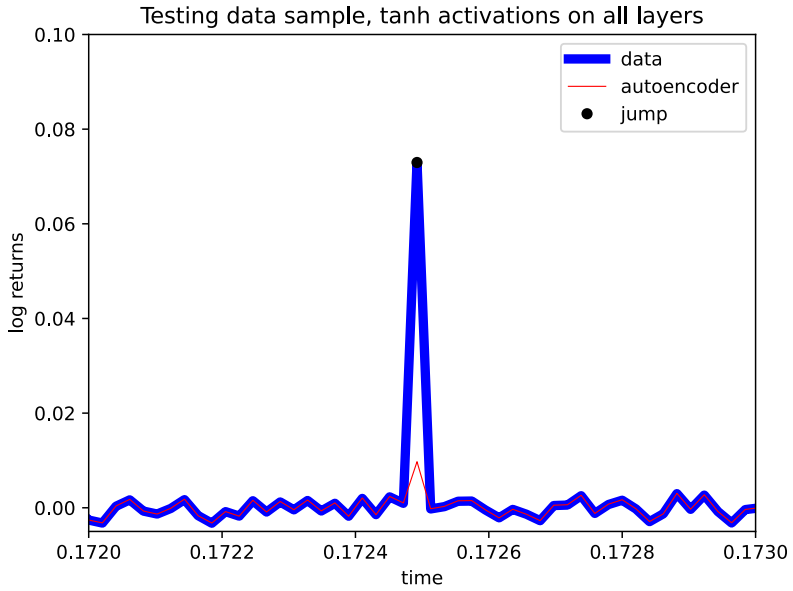

Fig. 7.9: Plot showing the application of a CNN autoencoder using only tanh activation functions.

The plots in this section are generated by scaling log returns by 100, then reducing them back to the original scale for plotting. Each CNN autoencoder here was only trained on 10 random diffusion data sets, with one epoch for each set.

The testing data was the same as that in the supervised analysis. The kernel size for each layer was set to 7. There were two CNN encoding layers, the first with 16 feature maps, the second with 8. After the first CNN encoding layer, average pooling was done with pool sizes of 2. In *Keras*, the convolution is undone in the decoding process by transpose CNN layers *Conv1Dtranspose*. Similarly the size reduction by pooling is done by *upsampling*, by the size of our pooling. This model took ~ 25 seconds to train. Less analysis on learning rate and the shrinkage of this hyperparameter across training was done than the previous section. However for the models produced, consistent performance was had using $\lambda = 0.001$, where the value shrunk by a factor of 0.99 each training set.

It should be noted that this use of tanh can stop the autoencoder being able to fit jump points if we were to train on data sets containing lots of jumps. This is potentially useful if we had to train the model on real data that we think may contain jumps. This is thought to be down to larger (in absolute terms) values after convolution being squashed to ± 1. To illustrate this we would have to include a plot which would show the same characteristic as figure 7.9. Figure 7.10 shows this autoencoder, which is also used as the foundation of the unsupervised jump test.

The dropout used throughout the autoencoder given in figure 7.10 was set to a rate of 20%.

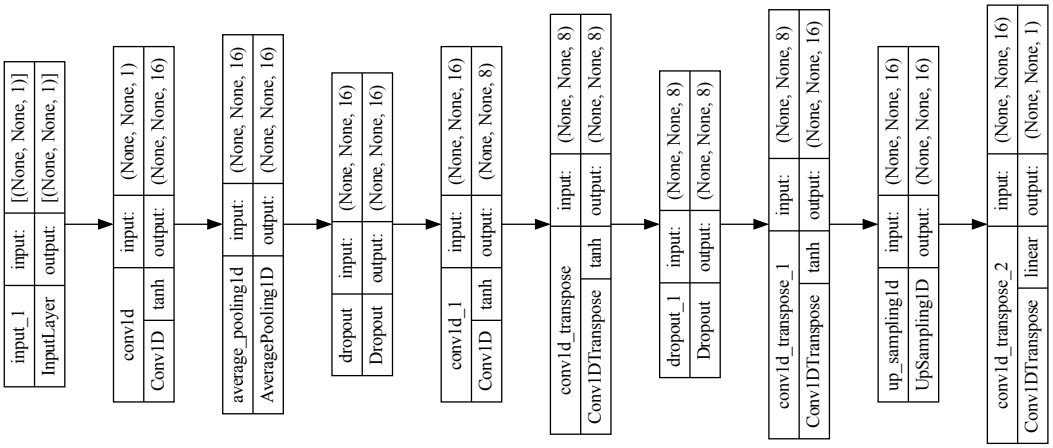

Fig. 7.10: Flow of autoencoder used as the jump test in this section, from input to output.

7.4.2 A note on pooling

Assuming most jumps result in an irregularly large return (positive or negative,) we expect larger values in the feature maps when we come to a jump point. As a way to decrease this effect, pooling is employed. Since the return can be positive or negative, minimum or maximum pooling would not work in our case and likely lead to worse results. It is thought that average pooling is a good choice here. It will not get rid of the effect of large jumps in the autoencoder feature map (when using testing data,) but should reduce it. The use of pooling in this situation is from

the assumption that no two jumps can be next to each other during high frequency observation (see chapter 3.)

Another viable pooling method that could be done is to create a pooling function that works like minimum pooling, but chooses the value for each pool as the number closest to zero. This is not done in this dissertation since we are using the *Keras* library, which does not include this as a pooling function.

7.4.3 How scaling affects a potential jump test

As mentioned in chapter 5, to use an autoencoder as a jump test in this scenario, we must find ϵ such that if a jump occurs we have:

$$\epsilon < |x_i - z_i|.$$

Where x_i is the input and z_i is the recreated data. The effect of scaling the log returns was removed in figures 7.7, 7.8 and 7.9 by undoing the scaling on the input and output of the autoencoder when plotting. It appears from figure 7.9, that using the input and output with the scaling undone, we should be able to select an ϵ to act as a jump test. Reason being, where there is no jump, the autoencoder output sits well on top of the input jump diffusion data.

If we were to use $\mathcal{L}$ in the autoencoder, we know from chapter 3 that jumps become more pronounced in the signal. In this case, a larger value of ϵ would be usable to perform a jump test.

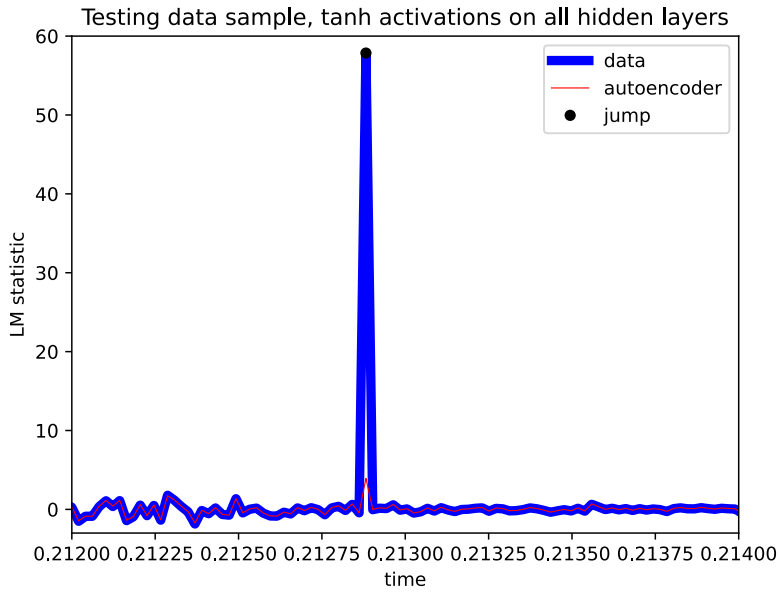

Fig. 7.11: Plot showing a similar model to figure 7.9, but here we are using $\mathcal{L}$ as our training feature.

In figure 7.11, the autoencoder point of the jump is around the largest values of $\mathcal{L}$ where there is only diffusion. However for the input, the jump is much more

pronounced by $\mathcal{L}$. In this case one could try values of ϵ around and above one, potentially having good performance.

But this is not the best way to use the scaling effect from the bipower variation. There should be a larger working range of ϵ if we use the autoencoder to output signals like that in figure 7.9, but then compute $\mathcal{L}$ for this output (once we have undone the simple scaling.) We however do not need this, by looking at the residuals between the signals like those plotted in figure 7.9, we can get a good range of values of ϵ to iterate through.

7.4.4 Autoencoder performance and ϵ search

We kept with the same autoencoder hyperparameters given when explaining which activation function we should use. The tanh activation was used on all CNN hidden layers, with a linear activation on the output layer. Where the autoencoder may recreate a jump point well, is where the jump that happens is small. Thus under this structure, we still cannot avoid misreading a reasonably small jump for a large volatility burst. We do not expect there to be an autoencoder built under this framework that we could say there exists some ϵ such that all jumps would be classified correctly. Thus finding the best epsilon for a good autoencoder under this framework, is about finding the best trade-off between false positives and false negatives.

For evaluation, we used the same 40 testing jump diffusion data sets as we had in the supervised learning section. It was found that the best value of ϵ, for this model and testing set was $\epsilon = 0.000853$. If we decreased ϵ further, getting fewer false negatives, we began to see significantly more false positives. We give the performance metrics of using this value of ϵ to classify jumps in our autoencoder, along with the scores for the LM test done on the same data, as done in the supervised learning analysis. The results for this are shown in table 7.5

	F1	BM	GM	MCC	SNS	SPC
Autoencoder $\epsilon = 0.000853$	0.94640	0.90103	0.94923	0.94755	0.90103	1.00000
LM test	0.90691	0.82968	0.91087	0.91079	0.82968	1

Tab. 7.5: Table showing the performance of the autoencoder model described, along with the LM test performance on the same data sets.

While the LM test produced no false positives compared to the autoencoder's 3, the autoencoder produced less than two thirds of the LM test's false negatives, 96 to 150. The performance that this method has over the LM test is of a similar margin to the supervised learning Bi-LSTM's. In fact, this shows slightly better performance than the best Bi-LSTM model shown in table 7.4.

It could be argued that this choice of ϵ does not make this comparison fair, since the LM test is not producing any false positives. If the value is increased to $\epsilon = 0.00135$, we get 112 false negatives instead of 96 and no false positives. This is still $\sim 25\%$ fewer false predictions over the LM test. The performance gained getting

more true jump predictions should motivate a choice for a smaller value of ϵ. Going back to the discussion on imbalanced data sets, in total across the 40 testing sets, we have near a million observations and less than 1000 total jump points. In other words we have near a million non jump points. The negative effect of having a handful of jump predictions on the $\sim$ million points that are not jumps, is not equal to the positive effect that having significantly more true jump predictions. This being down to the fact that jumps here are assumed to be rare.

Now we have used the testing data set to find this value of ϵ. We should make sure this choice of ϵ and the performance of it as a jump test in table 7.5, is not dependent on this testing data set.

Thus we create a new set of testing data, also of 40 random paths, where the market model parameters are sampled from table 7.1. We then do not do an iterative search for values of ϵ, but instead do a jump test using $\epsilon = 0.000853$. Similarly we perform the LM test on this new testing set.

	F1	BM	GM	MCC	SNS	SPC
Autoencoder $\epsilon = 000853$	0.95500	0.91620	0.95718	0.95568	0.91620	1.00000
LM test	0.93081	0.87060	0.93305	0.93298	0.87058	1

Tab. 7.6: Table showing a validation test for $\epsilon = 0.000853$

For the data produced in table 7.6, we see that the LM test has higher performance than before. So does the autoencoder result to a less degree. What is not lost, is the autoencoder's superior performance over the LM test. This validates that this "good" choice of ϵ is not dependent / only suited to the data sets we searched for it on.

Chapter 8

Discussion and conclusion

8.1 Achievements

In chapter 2 we spoke about the importance of including jumps into a diffusion driven market model. The selection of jump-diffusion models presented were used to generate the data in all of the analysis. When generating a data set, we randomly choose between the Bates, Merton and SVJJ models, and then select random model parameters out of the ranges given in table 7.1. This together with independent Brownian motions across different data sets was intended to prevent the deep learning jump tests being overfitted to the training data, meaning the jump test is more robust.

By reviewing the derivation of the LM jump test we gained more insight into the general assumptions of theoretically having jumps present in financial data. The LM test described in chapter 3 was then used as a benchmark comparison throughout the analysis.

As fundamental to the analysis, we had to review the components of deep learning in chapter 4. This was important to tailor the model types described in chapter 5, to do the tasks we wanted. Key examples include choices of loss and activation functions.

It became clear early on, that if we are training with log returns (unscaled) for data of this frequency of 2 minute observations, the majority of the data was too close to zero for any of the deep learning methods to achieve meaningful results. Explained in chapter 3 was how the LM statistic is suited to hypothesis testing. But being created by dividing the log return by the bipower variation (square root which is the same order as the log return,) we create a feature that is in a sense normalised. The effect of the jump in the bipower variation vanishes as observation period goes to zero, making jumps in the signal of this feature more prominent. For both using the supervised learning Bi-LSTM, and the unsupervised CNN autoencoder, using the LM statistic as our training feature, led to superior performance to using the LM statistic in the accompanying null hypothesis test. It was worried that this feature which exaggerated most jumps, and reduced the effect of the diffusion process, was key to the performance of the deep learning jump tests that were implemented. This was proven not the case when the problem of all values being too close to zero was overcame: by simple scaling of the log returns. In the case of the autoencoder, we then could undo the scaling on the output of the model. It was

clear from the results of both methods that there was no benefit to using the LM statistic as a training feature, instead of just scaling the log returns enough to allow the model to train. In fact there appeared there may be benefit in not using the LM statistic as a training feature.

This was an importance of this achievement since in Bashchenko and Marchal (2019) supervised deep learning jump detection was done using the LM statistic as the training feature. What this dissertation shows is that we need not be dependent on that existing high performing jump detection framework, whilst still having superior performance to this benchmark, over our out of sample simulated paths. Furthermore by using a Bi-LSTM, we could use as low as 70 units and 30 training paths in order to get superior performance. This and only using each data set for a single epoch ensured we were not overfitting to the training data. Having this supervised learning jump test beat our performance benchmark, and be independent of existing jump detection framework, means the first aim has been strongly achieved in the analysis.

To link the first and second aim, note an important distinction. Using the supervised approach, the loss function targets known jump points. In a sense we chose a set of parameters, and hoped they could model the relationship between the signal and whether or not there is a jump at a given point. No assumption on the dynamics is made.

To formulate the second method which used autoencoders, this was largely not the case at all. Largely in chapter 5 we speak about the field of unsupervised anomaly detection being used in more and more fields. When distinguishing between a jump and an anomaly in a financial time series, we argue that is a sort of nomenclature. Let's say all jumps are anomalies but not all anomalies are what we are calling jumps in this context. Most of these unsupervised methods are unsuited to asset price data. This is because most methods do not consider temporal movements in asset price movement. To use these methods, single value features estimating temporal effect would need to be artificially created. This reason strongly motivated the use of CNNs for the anomaly detection since the temporal nature would be modelled through the convolution. Specifically we used a CNN based autoencoder that recreated the input signal.

The general idea behind our autoencoder is as follows. You input data to this model, it goes through layers of convolution, activation functions and pooling. Then from this reduced state, the signal gets recreated. Mostly one would train an autoencoder on the data that is available. The finite set of model parameters and amount of training data would mean that the model is not optimised for what would be deemed as anomalies, since by definition anomalies would have to be rare and not frequent enough in the training data set. Thus if there is an anomaly when applied out of sample, the autoencoder should not be able to model the point where there is an anomaly. A relationship between the input and output of the autoencoder is drawn to determine an anomaly.

The use of autoencoders in this dissertation takes this idea a step further. Assuming the validity of the market model assumptions in chapter 2, we have the autoencoder be modelled / trained to the diffusion process. The justifiable benefit to this is that when we feed in data to the autoencoder, that contains jumps, we

try fit the diffusion process to points that jumps occur. We showed what this looks like in figure 7.9. This allowed us to easily find a value ϵ such that if the absolute difference between the input and output of the autoencoder was greater than ϵ, we had a competitive jump test. By competitive jump test we mean beating the LM test on the same data sets. The choice of ϵ was validated by using it on separate data sets that were not used for its selection. The same improvement over the LM test was found there.

By having this method which takes few assumptions other that jumps are rare, and model what we deem common dynamics, we erase the need for having jump labels in the training data. This creates much more opportunity for application to real data, since jumps (being theoretical explanation of observation) would not have to be synthetically added or speculatively labelled in the training sets. The use of autoencoders thus allowed the achievement of the second aim of this dissertation.

8.2 Issues of application

What we saw in the previous chapter is deep learning methods having superior performance to the LM test, when scored on the same out of sample testing data sets. For both the supervised learning approach and the autoencoder method, the best jump tests were able to detect significantly more true jumps. This improvement was only marginally countered by the few instances of false positive prediction. This is compared to the LM test seemingly not making false positive measurements. For the number of non jump points in the imbalanced data set, this can be seen as not a big weakness.

It has been covered that for the supervised approach we cannot train a prediction model of this form on real data. This is because jumps are a theoretical construct. Observation of asset prices is always discrete: Therefore it is not possible to say whether movement is because of a theorised continuous process, or a discontinuous one. As mentioned we do need to have enough jumps in the data sets, for the model to actually learn how to identify a jump. Thus it would be difficult to create training data with enough jumps if one tried to relate large positive or negative returns, coinciding with the arrival relevant market information. Difficult being a key word, for this is not far off some industry practices. For example some hedge funds do quantitative trading based on sentiment analysis. While sentiment analysis sounds more qualitative, here it is not. The models are trained with natural language data from news sources and social media, arriving in real time. Coincidentally, one would also consider using a LSTM to do this. If we assume that observed jumps are the result of the arrival of new market information, under the efficient market hypothesis, natural language processing in trading, is a sort of jump analysis.

What was done in Au Yeung *et al.* (2020), was to train the supervised learning model on real data, where Poisson jumps were added across the training sets. What our unsupervised method has shown, is that we do not need to have jump labels to uncover the same jump test performance as we had using supervised learning.

In fact it is clear that using the autoencoder is at the very least, the same or better than the supervised learning approach tried.

We must note two key issues with using deep learning methods in practice. Firstly access to data. Typically deep learning methods require vast amounts of data sets. Secondly run time. With these vast amounts of data, one typically needs neural networks consisting of many layers and many parameters. One may even need to use each data set for multiple epochs. Both the supervised and unsupervised approaches done in this dissertation mostly avoid these issues. The Bi-LSTM used only 30 training sets with 1 epoch each. All models of this type did not take longer than 4 minutes to train. This is good considering we still beat the LM test out of sample. Using the CNN autoencoder, we were allowed to create a model of much fewer parameters over the Bi-LSTM. What is more, the performance found was using just 10 data sets for training, again using 1 epoch each. This only took 25 seconds to train. A lot of deep learning theoretical findings will have their potential application limited by their large computation time. That is not a concern for the methods that have been implemented in this dissertation, since calibration is fast.

Bibliography

Aït-Sahalia, Y. (2004). Disentangling diffusion from jumps, *Journal of Financial Economics* **74**(3): 487–528.

Aït-Sahalia, Y. and Jacod, J. (2009). Testing for jumps in a discretely observed process, *The Annals of Statistics* **37**(1): 184–222.

Anandakrishnan, A., Kumar, S., Statnikov, A., Faruquie, T. and Xu, D. (2018). Anomaly detection in finance: Editors' introduction, *KDD 2017 Workshop on Anomaly Detection in Finance*, PMLR, pp. 1–7.

Au Yeung, J., Wei, Z.-k., Chan, K. Y., Lau, H. and Yiu, K.-F. (2020). Jump detection in financial time series using machine learning algorithms, *Soft Computing* **24**: 1789–1801.

Audibert, J. (2021). *Unsupervised Anomaly Detection in Time-Series*, PhD thesis, Sorbonne Université.

Barndorff-Nielsen, O. E. and Shephard, N. (2004). Power and bipower variation with stochastic volatility and jumps, *Journal of Financial Econometrics* **2**(1): 1–37.

Bashchenko, O. and Marchal, A. (2019). Deep learning, jumps, and volatility bursts, *Swiss Finance Institute Research Paper* (20-10).

Bates, D. (1996). Jumps and stochastic volatility: Exchange rate processes implicit in Deutsche mark options, *The Review of Financial Studies* **9**(1): 69–107.

Bishop, C. and Nasrabadi, N. (2006). *Pattern Recognition and Machine Learning*, Vol. 4, Springer.

Black, F. and Scholes, M. (1973). The pricing of options and corporate liabilities, *Journal of Political Economy* **81**(3): 637–654.

Duffie, D., Pan, J. and Singleton, K. (2000). Transform analysis and asset pricing for affine jump-diffusions, *Econometrica* **68**(6): 1343–1376.

Dumitru, A.-M. and Urga, G. (2012). Identifying jumps in financial assets: A comparison between nonparametric jump tests, *Journal of Business & Economic Statistics* **30**(2): 242–255.

Eisenstein, K. (2022). *Identifying jumps in financial time series: A comparative study of jump detection tests*, Master's thesis, University of Cape Town.

Elman, J. (1990). Finding structure in time, *Cognitive Science* **14**(2): 179–211.

Galambos, J. (1978). *The Asymptotic Theory of Extreme Order Statistics*, John Wiley & Sons.

Gatheral, J. (2011). *The Volatility Surface: A Practitioner's Guide*, John Wiley & Sons.

Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep learning*, MIT press.

Heston, S. L. (1993). A closed-form solution for options with stochastic volatility with applications to bond and currency options, *The Review of Financial Studies* **6**(2): 327–343.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory, *Neural Computation* **9**(8): 1735–1780.

Hong, L. and Zou, J. (2015). Jump tests for semimartingales, *South African Actuarial Journal* **15**(1): 93–108.

Jurafsky, D. and Martin, J. H. (n.d.). Speech and language processing: An introduction to natural language processing, computational linguistics, and speech recognition.

Lee, S. and Mykland, P. (2007). Jumps in financial markets: A new nonparametric test and jump dynamics, *The Review of Financial Studies* **21**(6): 2535–2563.

Luque, A., Carrasco, A., Martín, A. and de Las Heras, A. (2019). The impact of class imbalance in classification performance metrics based on the binary confusion matrix, *Pattern Recognition* **91**: 216–231.

Merton, R. (1976). Option pricing when underlying stock returns are discontinuous, *Journal of Financial Economics* **3**(1–2): 125–144.

Nielsen, M. (2015). *Neural networks and deep learning*, Vol. 25, Determination press San Francisco, CA, USA.

Powers, D. (2003). Recall and precision versus the bookmaker, *Cognitive Science - COGSCI* pp. 529–534.

Shreve, S. *et al. (2004).* Stochastic Calculus for Finance II: Continuous-Time Models, *Vol. 11, Springer.*