Universität St.Gallen

Quantile Regression of High-Frequency Data Tail Dynamics via a Recurrent Neural Network

Nicolo Ceneda

Master of Arts in Banking and Finance

University of St Gallen

Supervisor: Prof. Dr. Christoph Aymanns

Co-supervisor: Prof. Dr. Manuel Ammann

May 18, 2020

Presented to the University of St. Gallen in fulfillment of the requirements for the Master of Arts in Banking and Finance

Abstract

The returns of financial asset are stochastic and their distributions have leptokurtic, asymmetric and time-varying tails, both conditionally and unconditionally. Although it is impossible to accurately predict future returns, it is possible to predict with precision the characteristics of their conditional distributions. To overcome the need for a distributional assumption typical of the GARCH-type models, as well as the issues of traditional quantile regression, we forecast the conditional quantiles and heavy tails of series of stock returns measured at high frequency, in a parsimonious quantile regression framework, using a parametric heavy tail quantile function first presented in Yan et al. (2018). We implement three recurrent neural networks to forecast the parameters of this quantile function: an LSTM that takes as input the series of the first four central moments of past returns, an LSTM that additionally takes in volume data, and an LSTM that additionally takes in the realized volatility. Evaluating a pinball loss function, we find that all three models outperform nine benchmark econometric models, with a consistently superior performance of the volume-based model.

Keywords: recurrent neural network, quantile regression, quantile function, tail dynamics, high-frequency data

Contents

List of Figures

List of Tables

1 Introduction

While machine learning models are generally concerned with the prediction of a single value of the output variable y, given the inputs $\boldsymbol{x}$, to estimate the conditional mean $E[y|\boldsymbol{x}]$, some situations also require the estimation of the parameters of the conditional distribution $p(y|\boldsymbol{x})$. This is the case for financial asset returns, which behave stochastically and have leptokurtic, asymmetric and time-varying tails, both conditionally and unconditionally, as shown in Cont (2001). Although it is impossible to accurately predict future returns, it is possible to predict the characteristics of their conditional distribution. An accurate estimation of these parameters is essential in the context of asset pricing and risk management.

In discrete-time Econometrics, the benchmark models for forecasting the conditional distribution $p(r_t|r_{t-1}, r_{t-2}, \dots)$ of the return r_t given past returns $\{r_{t-1}, r_{t-2}, \dots\}$ are the Generalized Autoregressive Conditional Heteroskedasticity model and its variations. The GARCH model was first introduced by Engle (1982) and Bollerslev (1986) to model time variations in volatility (heteroscedasticity), and since then several popular variants have been developed, such as Nelson (1991)'s EGARCH and Glosten, Jagannathan, and Runkle (1993)'s GJR. These models describe the conditional distribution by assuming its probability density function and letting the distribution parameters depend on past observations. Although these models were originally conceived to model and forecast the conditional volatility, they can also be used to predict the conditional quantiles, as they fully describe the conditional distribution. Besides the probability density function, quantiles are another way to describe the shape of a distribution: modeling the conditional quantiles for a finite set of probabilities is almost equivalent to modeling the conditional distribution, i.e. estimating the conditional mean, volatility, skewness and kurtosis.

Another method used to predict the characteristics of the conditional distribution is quantile regression, which was first presented in Koenker and Bassett Jr (1978) and then revisited in Koenker and Hallock (2001). Unlike the family of GARCH models, quantile regression forecasts the conditional quantiles by means of a parametric function, rather than making distributional assumptions. However, the traditional quantile regression has some shortcomings, such as quantile crossing, i.e. the lack of monotonicity of the estimated quantiles, the increasing number of parameters required to estimate more quantiles and the lack of interpretability.

To overcome the need for a distributional assumption, which may introduce tractability and ill-posedness issues, as well as the issues of traditional quantile regression, Yan et al. (2018) forecast the conditional quantiles and heavy tails of series of financial asset returns, using a parsimonious quantile regression that describes the conditional distribution $p(r_t|r_{t-1}, r_{t-2}, \dots)$ by means of a parametric heavy tail quantile function (HTQF). The Q-Q plot of the proposed

HTQF against the standard normal distribution has an inverted S shape and the heaviness of the tails is controlled by two parameters in a flexible way. Moreover, they allow the time-varying parameters to depend on past information through a long-short term memory unit (LSTM).

A long-short Term Memory unit is a particular type of recurrent neural network, first introduced by Hochreiter and Schmidhuber (1997). Since then, it has been applied to several practical problems, such as the video recognition tasks in Fan et al. (2018), Wang et al. (2018) [47] and Wang et al. (2018) [48], the video prediction task in Chen et al. (2018), and the video retrieval task in Feng et al. (2018). Unlike other machine learning models, which assume that the input data are independently and identically distributed and thus fail to recognize the order of sequential data, recurrent neural networks have memory of the past information and are capable of handling sequences and processing new information accordingly. For this reason, they are particularly suited to tackle financial tasks that require to learn the time dependencies of sequential data.

In this paper we forecast the conditional quantiles and heavy tails of series of financial asset returns measured at high frequency, in a parsimonious quantile regression framework, that describes the conditional distribution using Yan et al. (2018)'s parametric heavy tail quantile function. We implement three machine learning models, that we use to learn the time varying parameters of the quantile function. The first model is the Classic LSTM-HTQF, which replicates the authors' original model. More precisely, this is a densely connected network, stacked on top of a recurrent neural network with a long-short term memory unit. It takes as input a feature matrix that contains the sequence of the first four central moments of a series of past returns, and outputs the predicted parameters of the quantile function for a given target return one step ahead. The second model is the Volume LSTM-HTQF, which deviates from the previous model only in the construction of the feature matrix, which is expanded to have a fifth feature, namely the trading volume associated with each observation in the sequence of past returns. The rationale behind this model is that the feature matrix should be designed to contain as much information as possible related to the conditional distribution. The third model is the Volatility LSTM-HTQF which, similarly to the Volume LSTM-HTQF, has a fifth feature, namely the realized volatility of the returns over a period of four weeks. The rationale behind this variation is to create a hybrid model between the Volume LSTM-HTQF and the Classic LSTM-HTQF: like the former, it adds an additional feature (the volatility), as well as additional information (the returns from further in the past), but like the latter it still uses only information that can be derived from the time series of prices. We train all three models on a training subset using a pinball loss function, which is evaluated at multiple probability levels. Furthermore, for each model we

choose the combination of hyperparameters that yields the best performance on a validation subset. Like Yan et al. (2018)'s original model, our models have two hyperparameters, namely the length of the series of past returns on which the HTQF parameters depend and the number of units in the hidden state of the LSTM. However, we add an additional hyperparameter, namely the size of the batch used to feed the data to the machine learning models. Finally, we compare the performance of the three machine learning models against nine econometric models, namely the GARCH, EGARCH, GJR and their variations GARCH-t, AR-GARCH-t EGARCH-t, AR-EGARCH-t, GJR-t and AR-GJR-t, which are estimated using maximum likelihood.

The sample used in this study is made up of tick-by-tick traded prices and volumes for ten highly liquid stocks listed on the Nasdaq Stock Market (NASDAQ) and the New York Stock Exchange (NYSE). For each security, we consider a fixed time range from 9.35 a.m. to 15.55 p.m. for each trading day over a time span of 20 weeks (from March 4, 2019 to July 19, 2019) for the Classic LSTM-HTQF and the Volume LSTM-HTQF, and over a time span of 24 weeks (from February 4, 2019 to July 19, 2019) for the Volatility LSTM-HTQF. The data are available through the Trade and Quote database which, for the product considered, covers the period 10/09/2003 - present, is updated daily and has time stamps with precision of a microsecond.

As explained in Brownlees and Gallo (2006), the structure of tick data is affected by several factors such as the regulatory framework, the procedures of the institution that collects and produces the information, and the technological improvements. Moreover, the series of tick data might contain errors, might not be time ordered, and might be affected by unusual market conditions. Therefore, as a first step, we implement a data cleaning process, during which we retain only trades which have not been corrected, changed or marked as cancel or error and that have a regular timing. Furthermore, we implement a heuristic procedure that allows us to identify and remove outliers. As a second step, we implement the data manipulation process, during which we aggregate blocks of simultaneous observations and resample the data at lower frequency to obtain time series suitable to conduct the analysis. After these steps, we obtain time series at 5 second intervals for each of the ten stocks in our sample.

The data cleaning and data manipulation processes require some choices that may have consequences on the results obtained. There is extensive literature on the issues regarding market rules and procedures, as well as the data cleaning and data manipulation processes. Hasbrouck, Sofianos, and Sosebee (1993) is a reference publication for the rules and procedure of the NYSE, although several changes have been put in place since then. Madhavan and Sofianos (1998) and Sofianos and Werner (2000) study several empirical aspects about how orders are handles and executed on the NYSE, while Bauwens and Giot (2013) analyse these

issues from an econometric standpoint, with a specific focus on the Trade and Quote database. Moreover, Blume and Goldstein (1997) outline a procedure to identify erroneous quote data and Oomen (2006) suggests a filter to detect outliers in price data. Finally, Vergote (2005) criticizes the 5 second rule presented in Lee and Ready (1991) to match trade and quote data.

Starting with the econometric models used as benchmarks, we find that the simplest models GARCH, GARGH-t, EGARCH, EGARCH-t, GJR and GJR-t do not achieve the best performance on our series of high frequency data. Instead, the benchmarks that model the conditional mean in an autoregressive way show a significantly better ability to capture the dynamics of the conditional distribution of returns. Moving on to the machine learning models, we find that the Volume LSTM-HTQF consistently outperforms the other two models, as well as each of the nine benchmarks. This result confirms the idea that the matrix of features should be expanded to include all information that can aid in the prediction of the conditional distribution of the returns. Moreover, we do not find evidence of a consistent under- or over-performance of the Volatility LSTM-HTQF with respect to the Classic LSTM-HTQF. However, the frequent underperformance of the former is likely due to the excessive aggregation created by the length of the time span (20 trading days) on which the realized volatility is calculated. Lastly, for all three types of machine learning models, we find that the size of the batch has a strong impact on the results and, in particular, that a smaller batch size improves the performance.

The contributions of this paper are: testing the ability of Yan et al. (2018)'s proposed parametric quantile function to capture the dynamics of the conditional distribution of stock returns measured at high frequency; implementing two novel variations (Volume LSTM-HTQF and Volatility LSTM-HTQF) of the LSTM recurrent neural network used to learn the parameters of the heavy tail quantile function; finding a new hyperparameter (batch size) that leads to significant improvements in the performance of the machine learning models; measuring the performance of the parametric quantile function against additional benchmark models (AR-GARCH-t, EGARCH, GJR) not used in the original study by Yan et al. (2018).

The reminder off this thesis is structured as follows: section 2 takes a mathematical stance to provide a review of the relevant machine learning models required to understand the functioning of the more articulated recurrent neural network; section 3 presents the benchmark models, the traditional quantile regression, the heavy tail quantile function and the three machine learning models implemented is this study; section 4 introduces the dataset and the sample, and explains the data cleaning and data manipulation processes; section 5 describes the data preprocessing process required for the machine learning models, the implementation of the machine learning models and the benchmark models, and discusses the results.

2 Theory Review

This section provides a mathematical review of the most important machine learning models required to understand the functioning of recurrent neural networks. Appendix I, J, K introduce other popular models, which however are outside the scope of this study.

2.1 Types Machine Learning

Machine learning is a subfield of computer science, emerged during the second half of the twentieth century from the study of computational learning in artificial intelligence. It is the science and application of self-learning algorithms that derive knowledge from data to solve different predictive and decision making tasks. Machine learning algorithms can be divided into three types: supervised learning, unsupervised learning and reinforcement learning.

Supervised learning – the focus of this study – aims to learn a model from a set of labeled training data to make predictions about unseen data. It is further divided into classification and regression tasks. The former aim to correctly classify new instances into two (binary classification) or more (multiclass classification) discrete, unordered classes. The latter aim to predict a continuous outcome, by learning its relationship with a number of explanatory variables.

Unsupervised learning aims to learn a model from a set of unlabeled training data. It is further divided into clustering and dimensionality reduction. The former is a technique that allows to organize data into subgroups, which share a certain degree of similarity, without prior knowledge of their group memberships. The latter is an approach in feature processing, that tries to remove the noise and compress high dimensionality data into a smaller dimensional subspace, while retaining the relevant information.

Reinforcement learning aims to develop a system, called agent, that learns from a series of interactions with the environment to improve its performance. It does so by trying to maximize a reward signals via a trial-and-error or deliberative planning approach.

2.2 Data Set

The most basic type of data set used in supervised learning is made up of a dataframe $\boldsymbol{X} \in \mathbb{R}^{(n\ x\ m)}$, that contains a set of m features (columns) for each of the n samples (rows), and a dataframe $\boldsymbol{y} \in \mathbb{R}^{(n)}$, that contains the target variable for each of the n samples.

$$\boldsymbol{X} = \begin{pmatrix} x_1^1 & \dots & x_1^m \\ \dots & \dots & \dots \\ x_n^1 & \dots & x_n^m \end{pmatrix} \qquad \boldsymbol{y} = \begin{pmatrix} y_1 \\ \dots \\ y_n \end{pmatrix} \tag{1}$$

2.3 Artificial Neuron

Neurons are interconnected nerve cells in the brain, that receive, process and transmit chemical and electrical signals. Figure 1 illustrates a simplified representation of their structure.

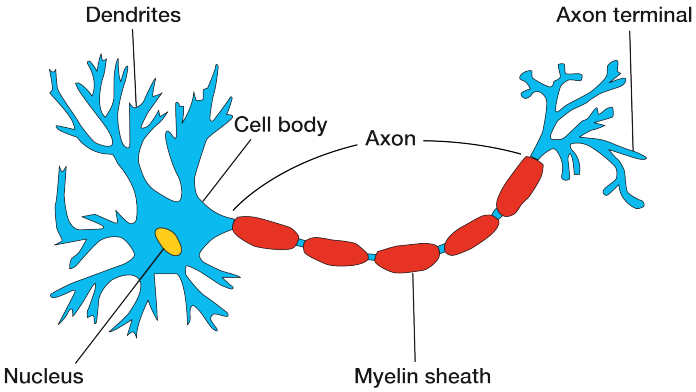

Figure 1: Neuron

McCulloch and Pitts (1943) published the first model of a simplified brain cell, designed as a simple logic gate with binary outputs, whereby multiple signals are received by the dentrites, are then integrated in the cell body and, if the accumulated signal exceeds a threshold level, an output signal is generated and passed on by the axon.

2.4 Perceptron

A few years later, Rosenbaltt (1957) published the first model of the Perceptron learning rule, based on the McCulloch and Pitts's (1943) neuron. Rosenblatt proposed an algorithm that would automatically learn the optimal weight coefficients that are multiplied by the input features to make a decision on whether the neuron fires the signal or not. In the context of supervised learning – and more precisely of binary classification – such a model of an artificial neuron can be applied to predict whether a sample belongs to a positive class (which takes the value of 1) or to a negative class (which takes the value of -1).

Let $\boldsymbol{x}_i$ be the vector of features for sample i from matrix $\boldsymbol{X}$ and $\boldsymbol{w}$ a vector of weights:

$$\boldsymbol{x}_i = \begin{pmatrix} x_i^1 & \dots & x_i^m \end{pmatrix} \qquad \boldsymbol{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_m \end{pmatrix} \tag{2}$$

Then the net-input z_i is defined as the linear combination of the two vectors:

$$z_i = \boldsymbol{x}_i \boldsymbol{w} = x_i^1 w_1 + \cdots + x_i^m w_m \tag{3}$$

Let $\phi(Z)$ be a decision function, that takes a value of 1 if the net-input Z is greater than a threshold value θ, otherwise it takes a value of -1. In the Perceptron learning rule, such a function is a variant of the unit step function:

$$\phi(Z) = \begin{cases} 1 & \text{if } Z \geq \theta \\ -1 & \text{otherwise} \end{cases} \tag{4}$$

For convenience, Equations 3 and 4 can be reformulated by bringing the threshold θ to the left of the inequality and defining a bias unit $w_0 = -\theta$ and an input value $x_i^0 = 1$:

$$z_i = x_i^0 w_0 + \boldsymbol{x}_i \boldsymbol{w} = x_i^0 w_0 + x_i^1 w_1 + \cdots + x_i^m w_m \tag{5}$$

$$\phi(Z) = \begin{cases} 1 & \text{if } Z \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{6}$$

We can now define Rosenblatt's Perceptron learning rule as follows:

1. Initialize the weights in $\boldsymbol{w}$ to small random numbers
2. For each epoch (i.e. number of passes over the training set) and for each training sample $\boldsymbol{x}_i$, compute the output value $\hat{y}_i$, i.e. the class label predicted by the unit step function based on the net input value, and update the weights in $\boldsymbol{w}$ simultaneously according to:

$$w_j := w_j + \Delta w_j \quad \text{where} \quad \Delta w_j = \eta(y_i - \hat{y}_i)x_i^j \tag{7}$$

where $0 < \eta < 1$ is the learning rate [1], y_i is the true class label, $\hat{y}_i$ is the predicted class label and x_i^j is feature associated with w_j (and 1 for w_0).

According to this learning rule, all the weights in the vector $\boldsymbol{w}$ are updated before the next sample is processed and the weight are modified only if the sample has been misclassified. Moreover, Rosenblatt proved that the weights (and therefore the number of misclassifications) converge to an optimal value only if the classes are linearly separable. Once the Perceptron has been trained, it can be used to predict the class label of unseen data. Finally, the capabilities of a Perceptron can be extended beyond a binary classification task to solve multi-class problems with the One-Versus-Rest approach [2]. Figure 2 illustrates the learning rule.

[1] the learning rate η only affects the classification outcome if the weights are initialized to non-zero values.

[2] The One-Versus-Rest approach consists in training one classifier per class, where the class under consideration is the positive class and all the others are negative classes and applying all classifiers to each new sample. Then the class label with the highest confidence is assigned to the new sample processed.

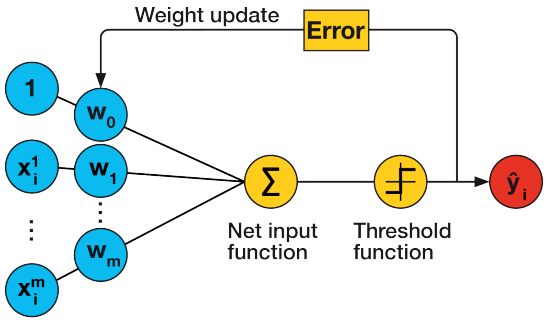

Figure 2: Perceptron learning rule

2.5 ADAptive LInear NEuron (Adaline)

Widrow (1960) brought an improvement on Rosenbaltt (1957) by introducing the Adaline learning rule, which has three key differences: first, it uses a linear activation function instead of a step function to update the weights; second, it optimizes an objective function during the learning process; third, it calculates the weight updates based on the whole training set instead of processing one sample at the time.

Let $\boldsymbol{X}$ be the matrix of features for all n samples and $\boldsymbol{w}$ the vector of weights:

$$\boldsymbol{X} = \begin{pmatrix} x_1^1 & \dots & x_1^m \\ \dots & \dots & \dots \\ x_n^1 & \dots & x_n^m \end{pmatrix} \qquad \boldsymbol{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_m \end{pmatrix} \tag{8}$$

Then the vector of net-inputs $\boldsymbol{z}$ for all n samples is defined as the product of the matrix of features by the vector of weights plus the vector of bias units of size n:

$$\boldsymbol{z} = \boldsymbol{w}_0 + \boldsymbol{X}\boldsymbol{w} \tag{9}$$

Note that, differently from Equation 5, the net input is now a vector of length n rather than a scalar. Now, let the linear activation function $\phi(Z)$ be the identity function of the net input:

$$\phi(Z) = Z \tag{10}$$

This implies that, while the Perceptron learns the optimal weights by comparing the true class label to the one predicted by a unit step function, the Adaline does so by comparing the true class label to the continuous valued output of a linear activation function $\phi(Z)$. However, once the weights have been optimized, the Adaline still uses the threshold function of Equation 6 to make the final class prediction.

Moreover, the Adaline algorithm introduces a very important element in machine learning theory, namely the presence of an objective function, which is optimized during the learning process. In this case, this is a cost function $J(\boldsymbol{w})$, which is defined as the sum of the squared errors between the true class label and the predicted one:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i \Big(y_i - \phi(z_i)\Big)^2 \tag{11}$$

This cost function is convex and, since the error term is calculated via a continuous activation function, it is also differentiable. This allows to use the gradient descent optimization algorithm to find the optimal weights that minimize the cost function. At each epoch, a step determined by the product of the learning rate η and the gradient $\nabla J(\boldsymbol{w})$ is taken in the direction opposite to the gradient:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w} \quad \text{where} \quad \Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w}) \tag{12}$$

As proven in Appendix A the update of weight w_i can be reformulated as:

$$\Delta w_j = \eta\sum_i \Big(y_i - \phi(z_i)\Big)x_i^j \tag{13}$$

By comparing Equations 7 and 13 it is clear that what differentiates the weight update in the Adaline algorithm from the one in the Perceptron is that now, for each epoch, the update is calculated based on all the samples in the training set, rather than updating the weight after the evaluation of each training sample. For this reason, the optimization algorithm takes the name of batch gradient descent. Furthermore, it is important to notice that the choice of the learning rate η is crucial in the success of the optimization process: a rate that is too large (small) would cause the gradient descent algorithm to overshoot (not to reach) the global minimum. Figure 3 illustrates the concepts of the gradient descent algorithm and of overshooting.

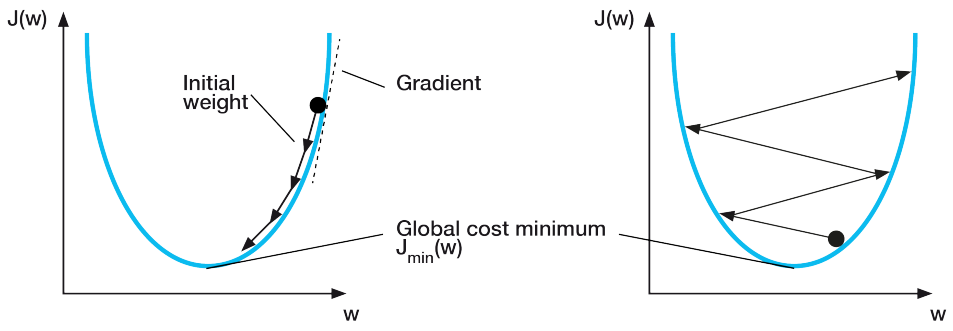

Figure 3: Gradient descent algorithm and overshooting

Finally, the convergence of the gradient descent algorithm benefits from a process of feature scaling, which, for the Adaline model, consists in z-scoring (i.e. demeaning and dividing by the volatility) all vectors of features $\boldsymbol{x^j}$ of the training and test sets using the first two moments of the training set:

$$\boldsymbol{x}^j := \frac{\boldsymbol{x^j} - \mu^j}{\sigma^j} \tag{14}$$

Widrow's Adaline learning rule can now be defined as follows:

1. Scale the matrix of features by z-scoring
2. Initialize the weights in $\boldsymbol{w}$ to small random numbers
3. For each epoch, compute the vector of activations, i.e. the output of the linear activation function $\phi(Z)$ based on the vector of net inputs $\boldsymbol{z}$, and update the weights in $\boldsymbol{w}$ simultaneously according to:

$$w_j := w_j + \Delta w_j \quad \text{where} \quad \Delta w_j = \eta \sum_i \Big(y_i - \phi(z_i)\Big) x_i^j \tag{15}$$

where $0 < \eta < 1$ is the learning rate, y_i is the true class label, $\phi(z_i)$ is the output of the linear activation function and x_i^j is feature associated with w_j (and 1 for w_0).

Once the Adaline has been trained, it can be used to predict the class label of unseen data. Figure 4 illustrates the learning rule.

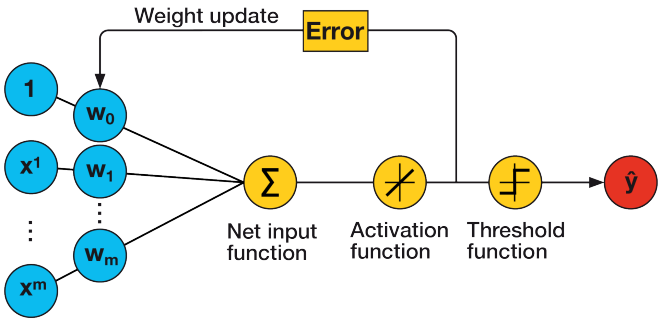

Figure 4: Adaline learning rule

Stochastic (also called iterative or online) gradient descent is a powerful alternative to batch gradient descent that is employed when the training set is very big. Indeed, as explained above, the latter technique calculates the weight update based on the whole training set, which may be computationally very expensive. Instead, the former technique calculates the weight update incrementally after each training sample, like in the Perceptron learning rule.

$$w_j := w_j + \Delta w_j \quad \text{where} \quad \Delta w_j = \eta \Big(y_i - \phi(z_i)\Big) x_i^j \tag{16}$$

To apply this algorithm, it is necessary to feed the training samples in a random order (hence the name stochastic) and to shuffle the training set before each epoch to prevent cycles. The power of stochastic gradient descent is the more frequent number of updates, that allows the algorithm to reach convergence much faster and escape local minima more easily. Another advantage is that it can be used for online learning, which consists in training the model continuously as new training samples come in. After processing the new training sample and updating the model, the data can be immediately discarded if storage space is a limit.

A compromise between batch and stochastic gradient descent is mini-batch learning, which consists in applying batch gradient descent to subsets of the training data.

2.6 Logistic Regression

Building on the model of the Adaline algorithm, the Logistic Regression introduces a more powerful learning rule to solve problems of binary and multi-class classification. It has three key differences: first, it uses a logistic sigmoid activation function instead of a linear function; second, it has a threshold function which returns the class labels 0 and 1 instead of 1 and -1; third, it has a different cost function, which however results in the same weight update.

The Logistic Regression is fed the same matrix of features $\boldsymbol{X}$ and the same vector of weights $\boldsymbol{w}$ as the Adaline, to calculate the same vector of net inputs $\boldsymbol{z}$. However, differently from the Adaline, which uses a linear activation function (Equation 10) to learn the optimal weights, the Logistic Regression uses the logistic sigmoid function. Appendix B gives a detailed explanation of the idea behind this new activation function:

$$\phi(Z) = \frac{1}{1 + e^{-Z}} \tag{17}$$

The output of the activation function is interpreted as the probability that sample i belongs to the positive class ($y_i = 1$), given its vector of features, parametrized by the vector of weights. This predicted probability can be converted to a binary classification problem via a threshold function:

$$\hat{Y} = \begin{cases} 1 & \text{if } \phi(Z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{18}$$

which is equivalent to:

$$\hat{Y} = \begin{cases} 1 & \text{if } Z \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

Therefore, like the Adaline, once the weights have been optimized, the Logistic Regression still

uses a threshold function to make the final class prediction, but with class labels 0 and 1 instead of -1 and 1.

Moreover, the Linear Regression algorithm uses a new cost function $J(\boldsymbol{w})$, whose full derivation is shown in Appendix C:

$$J(\boldsymbol{w}) = \sum_i \Big[-y_i \, log\Big(\phi(z_i)\Big) - (1-y_i) \, log\Big(1-\phi(z_i)\Big)\Big] \tag{20}$$

Like for the Adaline, this cost function allows to use the gradient descent optimization algorithm to find the optimal weights. Appendix D shows that the weight update formula is equal to the one in the Adaline model:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w} \quad \text{where} \quad \Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w}) \tag{21}$$

which results in:

$$\Delta w_j = \eta \sum_i \Big(y_i - \phi(z_i)\Big) x_i^j \tag{22}$$

Therefore, the logistic regression learning rule is defined in the same way as for Widrow's Adaline algorithm, with the above mentioned differences. Figure 5 illustrates the learning rule.

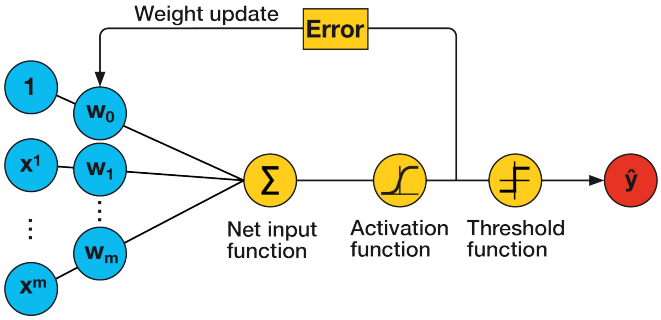

Figure 5: Logistic Regression learning rule

Appendix E introduces the problems of over- and under-fitting and explains how these issues can be tackled with regularization techniques. Under L2-regularization, the cost function $J(\boldsymbol{w})$ of the Linear Regression algorithm becomes:

$$J(\boldsymbol{w}) = \sum_i \Big[-y_i \, log\Big(\phi(z_i)\Big) - (1-y_i) \, log\Big(1-\phi(z_i)\Big)\Big] - \frac{\lambda}{2}||\boldsymbol{w}||^2 \tag{23}$$

2.7 Multilayer Perceptron

A Multilayer Perceptron is a fully connected neural network made up of multiple single neurons. More precisely, it is a type of Feedforward Neural Network, i.e. an architecture where each layer serves as the input to the next layer, without loops. This section introduces a simple architecture with only three layers, where the activation units in the input, hidden and output layers are fully connected. A neural network with more than one hidden layer is said to have a deep architecture [3]. Figure 6 illustrates the architecture of the Multilayer Perceptron.

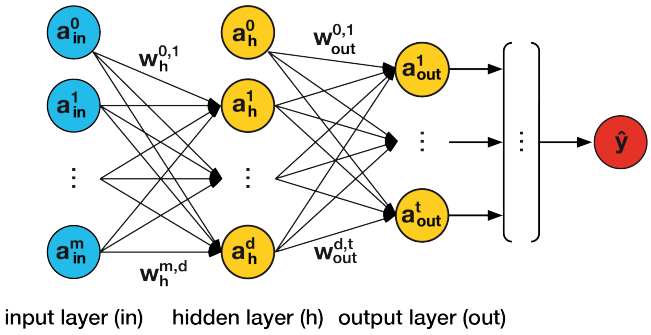

Figure 6: Multilayer Perceptron

First off, since a Multilayer Perceptron is generally trained via mini-batch learning, let $\boldsymbol{X}^{mb} \in \mathbb{R}^{(bs\ x\ m)}$ be the matrix of features for a mini-batch containing mb samples:

$$\boldsymbol{X}^{mb} = \begin{pmatrix} x_1^1 & \dots & x_1^m \\ \dots & \dots & \dots \\ x_{bs}^1 & \dots & x_{bs}^m \end{pmatrix} \tag{24}$$

Moreover, the convergence of the gradient descent algorithm benefits from a process of feature scaling, which, in the case of image recognition with a Multilayer Perceptron, consists in normalizing all features in the range $[-1, +1]$. More precisely, the features are scaled individually, rather than using the scaling parameters derived from the training set to scale each column in the train and test sets.

Let's now describe the architecture of the network. The activation units are denoted as $\boldsymbol{a}^j_{(in)}$, $\boldsymbol{a}^j_{(h)}$, $\boldsymbol{a}^j_{(out)}$ for the input, hidden and output layers respectively, where $\boldsymbol{a}^0_{(in)}$ and $\boldsymbol{a}^0_{(h)}$ are the vectors of unit values to be multiplied by the bias units (which are set to 1). Since each activation unit $\boldsymbol{a}^j_{(l)}$ in a layer (l) is connected to each unit $\boldsymbol{a}^k_{(l+1)}$ in the next layer $(l+1)$ via

[3] The number of layers and units in a neural network can be regarded as hyperparameters to be optimized using cross-validation techniques. As the error gradients calculated via backpropagation become increasingly small as more layers are added to the network's architecture (a problem called vanishing gradient), special algorithms have been designed to train such deep neutral networks (a field called deep learning).

a weight coefficient $w_{(l+1)}^{j,k}$, we can define the weight matrix connecting the input and hidden layers as $\boldsymbol{W}_{(h)} \in \mathbb{R}^{(m+1\ x\ d)}$ and the weight matrix connecting the hidden and output layers as $\boldsymbol{W}_{(out)} \in \mathbb{R}^{(d+1\ x\ t)}$:

$$\boldsymbol{W}_{(h)} = \begin{pmatrix} w_{(h)}^{0,1} & \dots & w_{(h)}^{0,d} \\ \dots & \dots & \dots \\ w_{(h)}^{m,1} & \dots & w_{(h)}^{m,d} \end{pmatrix} \qquad \boldsymbol{W}_{(out)} = \begin{pmatrix} w_{(out)}^{0,1} & \dots & w_{(out)}^{0,t} \\ \dots & \dots & \dots \\ w_{(out)}^{d,1} & \dots & w_{(out)}^{d,t} \end{pmatrix} \tag{25}$$

While an architecture with one unit in the output layer is sufficient to perform a binary classification task, it necessary to have as many units in the output layer as the number of classes to perform multi-class classification tasks via the one-versus-all technique. To do so, the class labels must be represented with the one-hot representation. For example, the class labels $Setosa = 0$, $Versicolor = 1$, $Virginica = 2$ can be encoded as follows:

$$0 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad 1 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \qquad 2 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{26}$$

We can now define the Multilayer Perceptron learning rule as follows. For each epoch and for each batch:

1. Activate the units in the input layer.
2. Forward propagate the patterns of the training data through the network to generate the output.
3. From the output, calculate the error to be minimized using a cost function.
4. Backpropagate the error, calculate its derivative with respect to each weight and update the model.

Once the Multilayer Perceptron has been trained, apply forward propagation to calculate the output and apply a threshold function to obtain the predicted class label in the one-hot representation.

Let's now analyze in detail each step. The activation of the units in the input layer is just the inputs values plus the unit values to be multiplied by the bias units:

$$\boldsymbol{A}_{(in)} = \begin{pmatrix} \boldsymbol{a}_{(in)}^0 & \boldsymbol{a}_{(in)}^1 & \dots & \boldsymbol{a}_{(in)}^m \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \boldsymbol{x}^1 & \dots & \boldsymbol{x}^m \end{pmatrix} \tag{27}$$

The forward propagation of the patterns of the training data starts with the calculation of

the pre-activations (net inputs) and the respective activations of the units of the hidden layer:

$$\boldsymbol{Z}_{(h)} = \boldsymbol{A}_{(in)}\boldsymbol{W}_{(h)} = \begin{pmatrix} z^1_{(h),1} & \dots & z^d_{(h),1} \\ \dots & \dots & \dots \\ z^1_{(h),bs} & \dots & z^d_{(h),bs} \end{pmatrix} \tag{28}$$

$$\boldsymbol{A}_{(h)} = \begin{pmatrix} \boldsymbol{a}^1_{(h)} & \dots & \boldsymbol{a}^d_{(h)} \end{pmatrix} = \phi(\boldsymbol{Z}_{(h)}) = \begin{pmatrix} \phi(\boldsymbol{z}^1_{(h)}) & \dots & \phi(\boldsymbol{z}^d_{(h)}) \end{pmatrix} \tag{29}$$

where $\boldsymbol{A}_{(in)} \in \mathbb{R}^{(bs\ x\ m+1)}$ is the matrix of features and unit values for a batch of training samples, $\boldsymbol{W}_{(h)} \in \mathbb{R}^{(m+1\ x\ d)}$ is the weight matrix connecting the input and hidden layers and $\boldsymbol{Z}_{(h)} \in \mathbb{R}^{(bs\ x\ d)}$ is the matrix of pre-activations. Moreover, $\boldsymbol{A}_{(h)} \in \mathbb{R}^{(bs\ x\ d)}$ is the matrix of activation units calculated using a differentiable non-linear activation function $\phi(Z)$, such as the sigmoid function, which maps a real input value into a continuous range between 0 and 1:

$$\phi(Z) = \frac{1}{1 + e^{-Z}}$$

Now, we redefine the matrix of activation units $\boldsymbol{A}_{(h)}$ to add a vector of unit values:

$$\boldsymbol{A}_{(h)} := \begin{pmatrix} \boldsymbol{a}^0_{(h)} & \boldsymbol{a}^1_{(h)} & \dots & \boldsymbol{a}^d_{(h)} \end{pmatrix} = \begin{pmatrix} \boldsymbol{1} & \phi(\boldsymbol{z}^1_{(h)}) & \dots & \phi(\boldsymbol{z}^d_{(h)}) \end{pmatrix} \tag{30}$$

and conduct similar calculations for the output layer:

$$\boldsymbol{Z}_{(out)} = \boldsymbol{A}_{(h)}\boldsymbol{W}_{(out)} = \begin{pmatrix} z^1_{(out),1} & \dots & z^t_{(out),1} \\ \dots & \dots & \dots \\ z^1_{(out),bs} & \dots & z^t_{(out),bs} \end{pmatrix} \tag{31}$$

$$\boldsymbol{A}_{(out)} = \begin{pmatrix} \boldsymbol{a}^1_{(h)} & \dots & \boldsymbol{a}^d_{(h)} \end{pmatrix} = \phi(\boldsymbol{Z}_{(out)}) = \begin{pmatrix} \phi(\boldsymbol{z}^1_{(out)}) & \dots & \phi(\boldsymbol{z}^t_{(out)}) \end{pmatrix} \tag{32}$$

where $\boldsymbol{A}_{(h)} \in \mathbb{R}^{(bs\ x\ d+1)}$, $\boldsymbol{W}_{(out)} \in \mathbb{R}^{(d+1\ x\ t)}$, $\boldsymbol{Z}_{(out)} \in \mathbb{R}^{(bs\ x\ t)}$ and $\boldsymbol{A}_{(out)} \in \mathbb{R}^{(bs\ x\ t)}$.

The optimal weights are calculated by minimizing a logistic cost function, which is very similar to the one employed in the Logistic Regression model. The only difference is that, since the Multilayer Perceptron returns a vector of length t for each sample (to be compared to the respective target vector in the one-hot encoding representation), the cost function needs to be generalized to all t activation units. Moreover, the L2-regularization term is added:

$$J(\boldsymbol{W}) = \sum_{i=1}^{n}\sum_{j=1}^{t}\Big[-y^j_{(oh),i}\ log(a^j_{(out),i}) - (1-y^j_{(oh),i})\ log(1-a^j_{(out),i})\Big] + \frac{\lambda}{2}\sum_{l+1\in\{h,out\}}\sum_{j=1}^{u_l}\sum_{k=1}^{u_l+1}\Big(w^{j,k}_{(l+1)}\Big)^2 \tag{33}$$

where $\boldsymbol{W}$ is a three-dimensional tensor made up of the matrices $\boldsymbol{W}_{(h)}$ and $\boldsymbol{W}_{(out)}$, $y^{j}_{(oh),i}$ is an element of the one-hot representation matrix $\boldsymbol{Y}_{(oh)} \in \mathbb{R}^{(bs\ x\ t)}$ for the whole sample, $a^{j}_{(out),i}$ is an element from the activation matrix $\boldsymbol{A}_{(out)} \in \mathbb{R}^{(bs\ x\ t)}$ for the whole sample, u_l is the number of units in layer (l), and where the regularization term calculates the squared sum of all weights (excluding the bias term) in the weight matrices. To minimize the cost function $J(\boldsymbol{W})$, we need to calculate the partial derivative with respect to each weight for every layer in the network. To do so we use the backpropagation algorithm (introduced in Appendix F), which allows to compute these partial derivatives efficiently.

Therefore, the forward propagation step returns a matrix $\boldsymbol{A}_{(out)} \in \mathbb{R}^{(bs\ x\ t)}$, which contains a vector of length t (i.e. the number of unique classes) for each sample in the mini-batch, that can be compared to a matrix $\boldsymbol{Y}_{(oh)} \in \mathbb{R}^{(bs\ x\ t)}$ in a one-hot representation. The difference between these two matrices returns the matrix $\boldsymbol{\Delta}_{(out)} \in \mathbb{R}^{(bs\ x\ t)}$ of errors of the output layer:

$$\boldsymbol{\Delta}_{(out)} = \boldsymbol{A}_{(out)} - \boldsymbol{Y}_{(oh)} \tag{34}$$

Next, it is possible to back propagate the error and calculate the matrix $\boldsymbol{\Delta}_{(h)} \in \mathbb{R}^{(bs\ x\ d)}$ of errors terms of the hidden layer:

$$\boldsymbol{\Delta}_{(h)} = \boldsymbol{\Delta}_{(out)}\big(\boldsymbol{W}_{(out)}\big)^{\top} \odot \big(\boldsymbol{A}_{(h)} \odot (1 - \boldsymbol{A}_{(h)})\big) \tag{35}$$

where $\boldsymbol{\Delta}_{(out)} \in \mathbb{R}^{(bs\ x\ t)}$, $\boldsymbol{W}_{(out)} \in \mathbb{R}^{(d\ x\ t)}$, $\boldsymbol{A}_{(h)} \in \mathbb{R}^{(bs\ x\ d)}$ and $\odot$ denotes the element wise multiplication. After calculating the error matrices, the weights can be updated. Appendix G explains how to derive Equation 35.

To minimize the cost function $J(\boldsymbol{W})$, we need to calculate the partial derivative with respect to each weight for every layer in the network.

$$\frac{\partial}{\partial \boldsymbol{W}_{(out)}} J(\boldsymbol{W}) = \big(\boldsymbol{A}_{(h)}\big)^{\top} \boldsymbol{\Delta}_{(out)} \tag{36}$$

$$\frac{\partial}{\partial \boldsymbol{W}_{(h)}} J(\boldsymbol{W}) = \big(\boldsymbol{A}_{(in)}\big)^{\top} \boldsymbol{\Delta}_{(h)} \tag{37}$$

The weight updates are now as follows:

$$\boldsymbol{W}_{(out)} := \boldsymbol{W}_{(out)} - \eta \Delta \boldsymbol{W}_{(out)} \quad \text{where} \quad \Delta \boldsymbol{W}_{(out)} = \big(\boldsymbol{A}_{(h)}\big)^{\top} \boldsymbol{\Delta}_{(out)} + \lambda \boldsymbol{W}_{(out)} \tag{38}$$

$$\boldsymbol{W}_{(h)} := \boldsymbol{W}_{(h)} - \eta \Delta \boldsymbol{W}_{(h)} \quad \text{where} \quad \Delta \boldsymbol{W}_{(h)} = \big(\boldsymbol{A}_{(in)}\big)^{\top} \boldsymbol{\Delta}_{(h)} + \lambda \boldsymbol{W}_{(h)} \tag{39}$$

where the correction term is added to all weights except the bias unit.

2.8 Recurrent Neural Network

2.8.1 Sequential Data

In general, supervised learning models assume that the input data are independently and identically distributed. Because of this mutual independence, the order in which a dataset of training samples $[\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n]^\top$ is fed to the algorithm is irrelevant. For example, in a dataset collecting the features of some flowers, the characteristics of one sample do not influence the characteristics of another sample. However, this is no longer the case when working with a sequence $[\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_s]^\top$, which is a series of ordered data that are not independent of each other. In particular, time series data represent a subcategory of sequential data, where each sample $\boldsymbol{x}_t$ is associated with a specific ordered time step t. For example, stock market data are time series, as each measurement is associated with a specific date and is closely related to the measurements on the previous days; on the other hand, DNA sequences are simple sequences, as the measurements are ordered but are not associated with a time dimension.

Traditional neural networks fail to recognize the order of the past training samples, i.e. they do not have memory: for example, in a Multi-layer Perceptron the samples go through the feedforward and backpropagation steps and the weights are updated, independently of the order in which the samples are processed. Their main limitation is that they take a fixed size vector as input, map it through a fixed amount of computational steps, and return a fixed size vector as output. By contrast, recurrent neural networks (RNN) have memory of the past information and are capable of handling sequences and processing new information accordingly. This is because RNNs allow to operate over sequences of vectors in the input, output, or both. Therefore, RNNs are particularly suited to solving tasks involving time series of financial data.

In order to implement the best architecture for a model, we must distinguish between different sequence modeling tasks. Karpathy (2015) summarized the main types of relationships between input and output data, as illustrated in Figure 7, where the blue circles represent the input vectors, the yellow circles represent the vectors containing the RNN's state, the red circles represent the output vectors and the arrows represent functions.

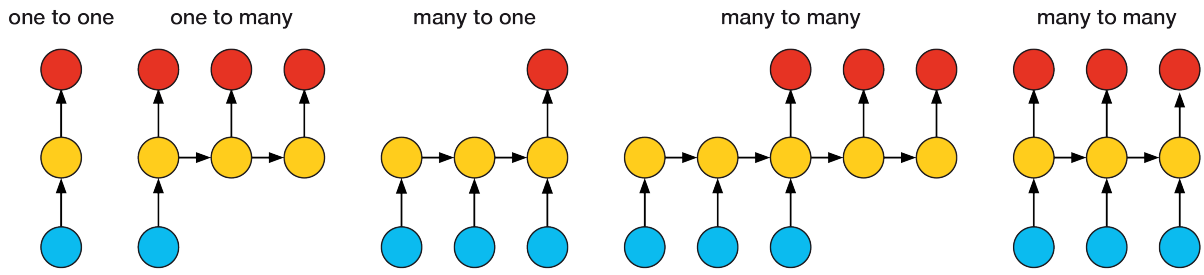

Figure 7: Types of relationships between input and output data

In *one to one*, neither the input nor the output is a sequence (both the input and the output are fixed-size vectors) and the relationship can be modeled in the usual way without a RNN (e.g image classification). All the other cases involve at least one sequence and must be modeled using a RNN: in *one to many*, the input data is a fixed-size vector but the output is a sequence (e.g. image captioning takes an image as input and outputs a sentence); in *many to one*, the output is a fixed-size vector but the input is a sequence (e.g. sentiment analysis takes a sentence as input and outputs a positive or negative sentiment); in *many to many*, both the input and the output are sequences (e.g. machine translation takes a sentence in one language as input and outputs the same sentence in another language), which may be synced (e.g. video classification takes a video as input and outputs a label for each frame of the video).

2.8.2 Memory of Sequential Data

A RNN differs from traditional neural network in that it contains a loop (a structure known as recurrent hedge), which allows information to persist in time. This is illustrated in Figure 8, which shows a traditional feedforward and a recurrent architecture with one hidden layer[4] where, although the units are not displayed, the input, hidden and output layers are vectors that contain many units.

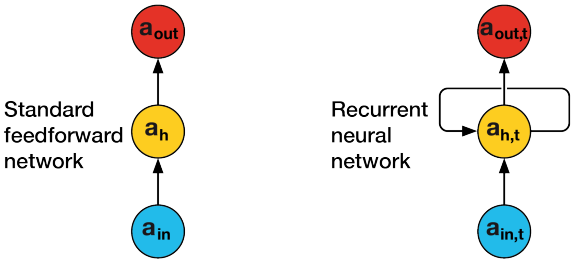

Figure 8: RNNs contain loops

While in traditional neural networks information flows from the input layer to the hidden layer and then from the hidden layer to the output layer, in RNNs information flows to the hidden layer from both the input layer of the current time step and the hidden layer of the previous time step, and then from the hidden layer to the output layer. More precisely, for a single-layer RNN, at the first time step $t = 0$, the units in the hidden layer $\boldsymbol{a}_{(h),0}$ are initialized to zero or small random values; then at all subsequent time steps $t > 0$, the units in the hidden layer $\boldsymbol{a}_{(h),t}$ receive as input both the activations of the input layer at the current time step $\boldsymbol{a}_{(in),t}$ and the activations of the same hidden layer at the previous time step $\boldsymbol{a}_{(h),t-1}$. Similarly, for a

[4]Although a RNN with one hidden layer is called single layer RNN, this does not denote the same idea as a single layer neural network, such as the Perceptron, the Adaline or the logistic regression, which are neural networks that do not have a hidden layer.

two layer RNN, at all time steps $t > 0$, the units in the first hidden layer $\boldsymbol{a}_{(h1),t}$ receive as input both the activations of the input layer at the current time step $\boldsymbol{a}_{(in),t}$ and the activations of the same hidden layer at the previous time step $\boldsymbol{a}_{(h1),t-1}$; the units in the second hidden layer $\boldsymbol{a}_{(h2),t}$ receive as input both the activation of the output layer at the current time step $\boldsymbol{a}_{(out1),t}$ from the previous layer and the activations of the same hidden layer at the previous time step $\boldsymbol{a}_{(h2),t-1}$.

This flow of information in the hidden layer from one time step to the next is what allows a RNN to have a memory. This is illustrated in Figure 9, which shows the compact and the unfolded representations for a single- and a two-layer RNN. In other words, a RNN can be thought of as a series of copies of the same network.

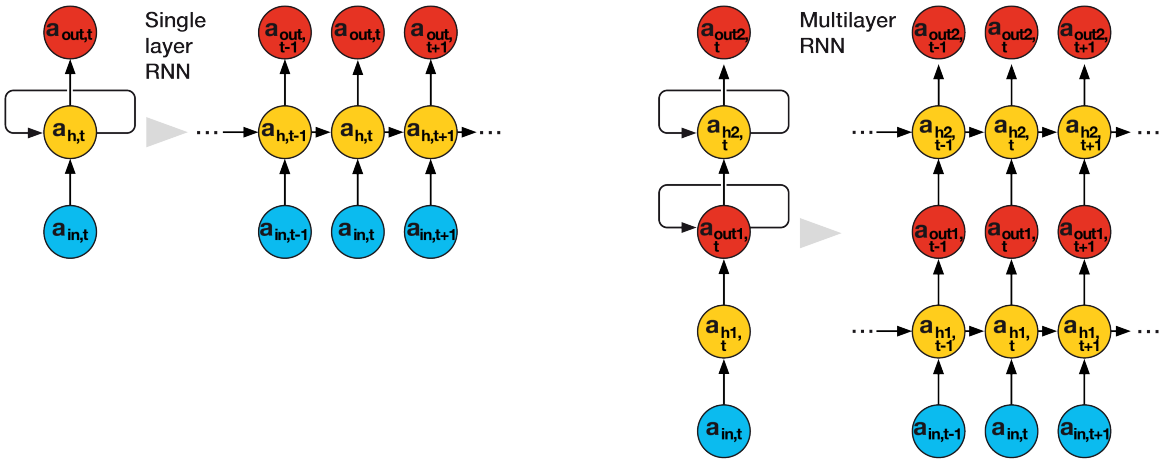

Figure 9: Compact and unrolled representations of a single- and a two-layer RNN

2.8.3 Architecture of a RNN

Let's now consider a single layer RNN. As anticipated, although the units are not displayed in the simplified representation of the RNN, the input, hidden and output layers are vectors that contains many units and are fully connected. The links between the layers (called directed hedge) are associated with a weight matrix, which is time independent. This architecture is illustrated in Figure 10.

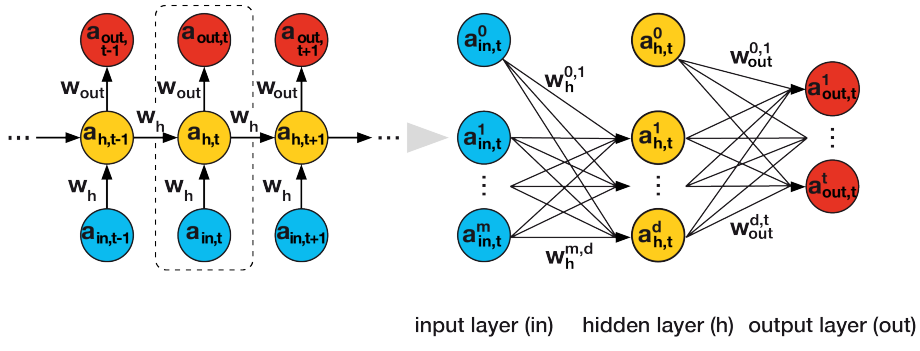

Figure 10: Unrolled and detailed representation of a single-layer RNN

Focusing on the current time step t, the activation units are denoted as $a^j_{(in),t}$, $a^j_{(h),t}$, $a^j_{(out),t}$ for the input, hidden and output layers respectively, where $a^0_{(in),t}$ and $a^0_{(h),t}$ are the unit values to be multiplied by the bias units. Since each activation unit $a^j_{(l),t}$ in a layer (l) is connected to each unit $a^k_{(l+1),t}$ in the next layer $(l+1)$ via a weight coefficient $w^{j,k}_{(l+1)}$, and each unit in the hidden layer $a^j_{(h),t}$ is connected to each unit in the same layer at the previous time step $a^k_{(h),t-1}$ via a weight coefficient $w^{j,k}_{(hh)}$, we can define the weight matrix connecting the input and hidden layers $\boldsymbol{W}_{(h)} \in \mathbb{R}^{(m+1\ x\ d)}$, the weight matrix connecting the hidden and output layers $\boldsymbol{W}_{(out)} \in \mathbb{R}^{(d+1\ x\ t)}$, and the weight matrix connecting the hidden layers at subsequent time steps $\boldsymbol{W}_{(hh)} \in \mathbb{R}^{(d\ x\ d)}$:

$$\boldsymbol{W}_{(h)} = \begin{pmatrix} w^{0,1}_{(h)} & \dots & w^{0,d}_{(h)} \\ \dots & \dots & \dots \\ w^{m,1}_{(h)} & \dots & w^{m,d}_{(h)} \end{pmatrix} \qquad \boldsymbol{W}_{(out)} = \begin{pmatrix} w^{0,1}_{(out)} & \dots & w^{0,t}_{(out)} \\ \dots & \dots & \dots \\ w^{d,1}_{(out)} & \dots & w^{d,t}_{(out)} \end{pmatrix} \tag{40}$$

$$\boldsymbol{W}_{(hh)} = \begin{pmatrix} w^{1,1}_{(hh)} & \dots & w^{1,d}_{(hh)} \\ \dots & \dots & \dots \\ w^{d,1}_{(hh)} & \dots & w^{d,d}_{(hh)} \end{pmatrix} \tag{41}$$

The activation of the units in the input layer is just the input values plus the unit value to be multiplied by the bias unit:

$$\boldsymbol{a}_{(in),t} = \begin{pmatrix} a^0_{(in),t} & a^1_{(in),t} & \dots & a^m_{(in),t} \end{pmatrix} = \begin{pmatrix} 1 & x^1_t & \dots & x^m_t \end{pmatrix} \tag{42}$$

The forward propagation of the patterns of the training data starts with the calculation of the pre-activations and the respective activations of the units of the hidden layer:

$$\boldsymbol{z}_{(h),t} = \boldsymbol{a}_{(in),t}\boldsymbol{W}_{(h)} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{(hh)} = \begin{pmatrix} z^1_{(h),t} & \dots & z^d_{(h),t} \end{pmatrix} \tag{43}$$

$$\boldsymbol{a}_{(h),t} = \begin{pmatrix} a^1_{(h),t} & \dots & a^d_{(h),t} \end{pmatrix} = \phi_{(h)}(\boldsymbol{z}_{(h),t}) = \begin{pmatrix} \phi_{(h)}(z^1_{(h),t}) & \dots & \phi_{(h)}(z^d_{(h),t}) \end{pmatrix} \tag{44}$$

where $\boldsymbol{a}_{(in),t} \in \mathbb{R}^{(1\ x\ m+1)}$, $\boldsymbol{W}_{(h)} \in \mathbb{R}^{(m+1\ x\ d)}$, $\boldsymbol{a}_{(h),t-1} \in \mathbb{R}^{(1\ x\ d)}$ and $\boldsymbol{W}_{(hh)} \in \mathbb{R}^{(d\ x\ d)}$, $\boldsymbol{z}_{(h),t} \in \mathbb{R}^{(1\ x\ d)}$, $\boldsymbol{a}_{(h),t} \in \mathbb{R}^{(1\ x\ d)}$, $\phi(Z)_{(h)}$ is the activation function of the hidden layer, and where, at the first time step $t = 0$, the units in the hidden layer $\boldsymbol{a}_{(h),0}$ have been initialized to zero or small random values. Now, we redefine the vector of activation units $\boldsymbol{a}_{(h),t}$ to add a unit value:

$$\boldsymbol{a}_{(h),t} := \begin{pmatrix} a^0_{(h),t} & a^1_{(h),t} & \dots & a^d_{(h),t} \end{pmatrix} = \begin{pmatrix} 1 & \phi_{(h)}(z^1_{(h),t}) & \dots & \phi_{(h)}(z^d_{(h),t}) \end{pmatrix} \tag{45}$$

and conduct similar calculations for the output layer:

$$\boldsymbol{z}_{(out),t} = \boldsymbol{a}_{(h),t}\boldsymbol{W}_{(out)} = \begin{pmatrix} z^1_{(out),t} & \dots & z^t_{(h),t} \end{pmatrix} \tag{46}$$

$$\boldsymbol{a}_{(out),t} = \begin{pmatrix} a^1_{(out),t} & \dots & a^t_{(out),t} \end{pmatrix} = \phi_{(out)}(\boldsymbol{z}_{(out),t}) = \begin{pmatrix} \phi_{(out)}(z^1_{(out),t}) & \dots & \phi_{(out)}(z^t_{(out),t}) \end{pmatrix} \tag{47}$$

where $\boldsymbol{a}_{(out),t} \in \mathbb{R}^{(1\ x\ 1+d)}$, $\boldsymbol{W}_{(out)} \in \mathbb{R}^{(d+1\ x\ t)}$, $\boldsymbol{z}_{(out),t} \in \mathbb{R}^{(1\ x\ t)}$ and $\boldsymbol{a}_{(out),t} \in \mathbb{R}^{(1\ x\ t)}$.

2.8.4 BPTT and Long-Range Dependencies

Werbos (1990) introduced the learning algorithm used to train RNN, namely the backpropagation through time (BPTT) algorithm. The main idea is that the overall loss L is the sum of all the loss functions:

$$L = \sum_{t=1}^{s} L_t \tag{48}$$

Since the loss at time t is dependent on the units in the hidden layer at all previous time steps $1 \leq k \leq t$, the gradient is computed as follows:

$$\frac{\partial L_t}{\partial \boldsymbol{W}_{(hh)}} = \frac{\partial L_t}{\partial \boldsymbol{a}_{(out),t}} \frac{\partial \boldsymbol{a}_{(out),t}}{\partial \boldsymbol{a}_{(h),t}} \left(\sum_{k=1}^{t} \frac{\partial \boldsymbol{a}_{(h),t}}{\partial \boldsymbol{a}_{(h),k}} \frac{\partial \boldsymbol{a}_{(h),k}}{\partial \boldsymbol{W}_{(hh)}} \right) \quad \text{where} \quad \frac{\partial \boldsymbol{a}_{(h),t}}{\partial \boldsymbol{a}_{(h),k}} = \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{a}_{(h),i}}{\partial \boldsymbol{a}_{(h),i-1}} \tag{49}$$

Unfortunately, as explained in Bengio, Simard, and Frasconi (1994) and Pascanu, Mikolov, and Bengio (2013), there are two widely known issues that arise when training RNN, the presence of the multiplicative factor $\frac{\partial \boldsymbol{a}_{(h),t}}{\partial \boldsymbol{a}_{(h),k}}$ when computing the gradients of the loss function introduces the issues of vanishing and exploding gradients. Since $\frac{\partial \boldsymbol{a}_{(h),t}}{\partial \boldsymbol{a}_{(h),k}}$ is computed as the product of $t-k$ elements, multiplying a weight $w^{j,k}_{(hh)}$ by itself $t-k$ times results in a factor $(w^{j,k}_{(hh)})^{t-k}$. Therefore, given a large $t-k$ (which indicates a long-range dependency), if $|w^{j,k}_{(hh)}| < 1$, then factor becomes very small; if $|w^{j,k}_{(hh)}| > 1$, then factor becomes very large. It is straightforward to conclude that the solution to the vanishing and exploding gradient problems is having $|w^{j,k}_{(hh)}| = 1$. In practice, this can be achieved with at least three solutions: gradient clipping, TBPTT or LSTM.

Gradient clipping consists in specifying a cut-off value for the gradients and assigning this value to all gradient values exceeding this threshold. TBPTT consists in limiting the number of time steps that the signal can backpropagate after each forward pass. While both solutions solve the exploding gradient problem, neither can solve the vanishing gradient one. However, LSTM can successfully target both issues.

2.8.5 Long-Short Term Memory Networks

Hochreiter and Schmidhuber (1997) introduced the long-short term memory network (LSTM), which is a special kind of RNN, capable of handling long-term dependencies and specifically designed to remember information for long periods of time. Like standard RNNs, LSTMs have a chain like structure and can be thought of as a series of copies of the same network. However, their repeating module is a memory cell, which replaces the hidden layer of the standard RNNs: instead of having a single neural network layer, there are four, which interact in a very specific way, as described below. Figure 11 illustrates the internal structure of the memory cell.

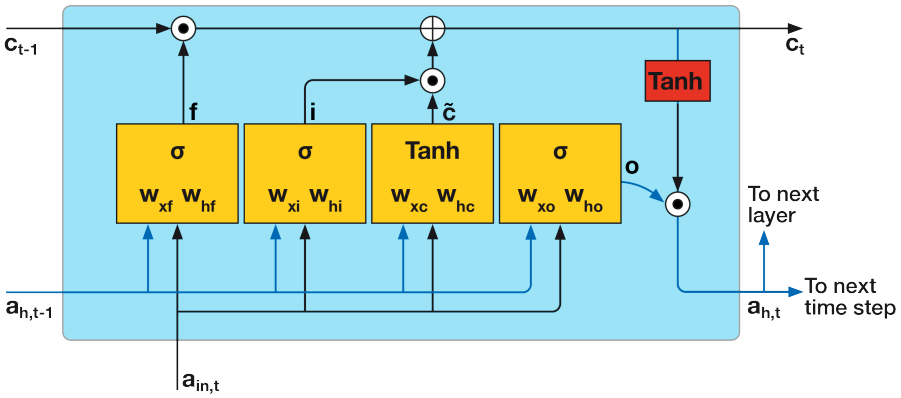

Figure 11: Internal structure of the memory cell of a LSTM

In each memory cell, there is a recurrent edge (the line running through the top of the memory cell) that has the desirable weight $|w_{(hh)}^{j,k}| = 1$, which allows to overcome the vanishing and exploding gradient problems. The collection of values associated with the recurrent edge are called cell state and run straight down the entire memory cell, with only some linear interactions: the cell state from the previous time step C_{t-1} is modified to obtain the cell state at the current time step C_t, without being multiplied directly by any weight; rather, the flow of information added to or removed from the cell state is controlled by three computation units called gates, which are defined in the following paragraph.

In the figure, $\odot$ indicates the element-wise multiplication, and $\oplus$ indicates the element-wise addition. Moreover, $\boldsymbol{a}_{(h),t-1}$ indicates the activation units in the hidden layer at time step $t-1$ and $\boldsymbol{a}_{(in),t}$ indicates the activation units in the input layer at time step t, i.e. the vector of input data. Four boxes, which are indicated by an activation function (either the sigmoid function σ or the hyperbolic tangent function $tanh$) and some weights, apply a linear combination by performing vector-matrix multiplications on their inputs $\boldsymbol{a}_{(h),t-1}$ and $\boldsymbol{a}_{(in),t}$. The gates controlling the flow of information are the three computation units with the sigmoid activation function and whose output units are passed through $\odot$, whereby the sigmoid layer

outputs numbers between 0 and 1 to define how much of each component should be let through (0 being nothing and 1 being all). More precisely:

- The forget gate[5] $\boldsymbol{f}_t$ determines which information is allowed to go through and which information is removed from the cell state. This is done by a sigmoid layer (with inputs $\boldsymbol{a}_{(in),t}$ and $\boldsymbol{a}_{(h),t-1}$) which returns a number between 0 and 1 for each number in the cell state $\boldsymbol{c}_{t-1}$.

$$\boldsymbol{f}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xf} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hf}\big) \tag{50}$$

- The input gate $\boldsymbol{i}_t$ and the candidate value $\tilde{\boldsymbol{c}}_t$ determine what new information are going to be stored in the cell state. This is done in two steps. First, a sigmoid layer (with inputs $\boldsymbol{a}_{(in),t}$ and $\boldsymbol{a}_{(h),t-1}$) decides which values are going to be updated and a hyperbolic tangent layer (with inputs $\boldsymbol{a}_{(in),t}$ and $\boldsymbol{a}_{(h),t-1}$) creates a vector of new candidate values that could be added to the cell state. Then, the old cell state $\boldsymbol{c}_{t-1}$ is updated into the new cell state $\boldsymbol{c}_t$: the forget gate $\boldsymbol{f}_t$ is multiplied by the old state $\boldsymbol{c}_{t-1}$, thus forgetting the things we decided to forget; the input gate $\boldsymbol{i}_t$ is multiplied by the new candidate values, thus scaling them by how much we decided to update each state value; finally, the two terms are added up.

$$\boldsymbol{i}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xi} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hi}\big) \tag{51}$$

$$\tilde{\boldsymbol{c}}_t = \tanh\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xc} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hc}\big) \tag{52}$$

$$\boldsymbol{c}_t = \big(\boldsymbol{f}_t \odot \boldsymbol{c}_{t-1}\big) \oplus \big(\boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t\big) \tag{53}$$

- The output gate $\boldsymbol{o}_t$ determines what is going to be output. This is done in two steps. First, a sigmoid layer (with inputs $\boldsymbol{a}_{(in),t}$ and $\boldsymbol{a}_{(h),t-1}$) decides what parts of the cell state $\boldsymbol{c}_t$ are going to be output. Then, the cell state is put through a hyperbolic tangent layer, which returns values between +1 and -1, and multiplied by the output gate, so that we output only the parts that we decided to.

$$\boldsymbol{o}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xo} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{ho}\big) \tag{54}$$

$$\boldsymbol{a}_{(h),t} = \boldsymbol{o}_t \odot \tanh\big(\boldsymbol{c}_t\big) \tag{55}$$

Although this subsection has described the standard LSTM used to model long-range dependencies, there are several variants of this architecture. Appendix H introduces some of these variants.

[5]The forget gate was not an original constituent of the LSTM. It was introduced only a few years later by Gers, Schmidhuber, and Cummins (1999) to improve the original model.

3 Models

In this section, we illustrate the three types of benchmark models used in this study (GARCH, EGARCH, GJR and their variations), the traditional quantile regression, the heavy tailed quantile regression and the original machine learning model (Classic LSTM-HTQF) introduced in the study of Yan et al. (2018), and two variations of this machine learning model (Volume LSTM-HTQF, Volatility LSTM-HTQF).

3.1 Benchmark Models

3.1.1 GARCH

A generalized autoregressive conditional heteroscedasticity (GARCH) model is a dynamic model that accounts for the conditional heteroscedasticity in an innovation process and is appropriate when positive and negative shocks of equal magnitude contribute equally to the volatility. This volatility clustering occurs when an innovation process does not have significant autocorrelation, but its variance changes over time. A GARCH model assumes the conditional distribution of the residual to be $\varepsilon_t \sim N(0, \sigma_t^2)$ and posits that the conditional variance σ_t^2 is the sum of three linear processes, with coefficients for each term: an ARCH polynomial, i.e. the past squared innovations $\{\varepsilon_{t-1}^2, \ldots, \varepsilon_{t-q}^2\}$; a GARCH polynomial, i.e. the past conditional variances $\{\sigma_{t-1}^2, \ldots, \sigma_{t-p}^2\}$; and the constants μ and ω for the innovation mean and the conditional variance models, respectively. More precisely, the general form of a GARCH(q, p) model is:

$$r_t = \mu + \varepsilon_t \quad \text{where} \quad \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim N(0,1) \equiv \varepsilon_t|\psi_{t-1} \sim N(0, \sigma_t^2) \tag{56}$$

$$\begin{aligned} \sigma_t^2 &= \omega + \sum_{i=1}^{q} \alpha_i \varepsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2 \\ &= \omega + \alpha_1 \varepsilon_{t-1}^2 + \cdots + \alpha_q \varepsilon_{t-q}^2 + \beta_1 \sigma_{t-1}^2 + \cdots + \beta_p \sigma_{t-p}^2 \end{aligned} \tag{57}$$

where ψ_{t-1} denotes information available up to time $t-1$, $\omega > 0$ and $\alpha_i, \beta_j \geq 0$ guarantees that $\sigma_t^2 > 0$ for all periods, and $\sum_{i=1}^{q} \alpha_i + \sum_{j=1}^{p} \beta_j < 1$ guarantees that σ_t^2 is stationary (and therefore the unconditional variance is finite).

The distributional assumption $\varepsilon_t \sim N(0, \sigma_t^2)$ allows to set up the likelihood function and estimate the model consisting of Equations 56 and 57 using maximum likelihood estimation. The log-likelihood is easily found as the model is conditionally Gaussian:

$$log(L) = \sum_{t=1}^{s} L_t \quad \text{where} \quad L_t = -\frac{1}{2} log(2\pi) - \frac{1}{2} log(\sigma_t^2) - \frac{1}{2} \frac{\varepsilon_t^2}{\sigma_t^2} \tag{58}$$

The estimates of $\omega, \alpha_i, \beta_j$ are found by choosing the parameters that maximize the likelihood function. This is done by a numerical approximation routine, which imposes the constraints introduced above. More precisely, given a time series from 0 to s, we use observation 0 to create the starting value of ε_0^2 and $Var(\varepsilon_t)$ to create the starting value of σ_0^2; we then use observations 1 to s to calculate the value of the likelihood function.

It is sometimes found that the standardized values ε_t/σ_t still have too fat tails compared to the assumed distribution $N(0,1)$. Therefore, a popular variation of the GARCH model is the GARCH-t model, which makes a different assumptions about the distribution of the error term, namely that they follow a a t-distribution. This variation allows to account for the leptokurtosis. More precisely, the general form of a GARCH-t model is:

$$r_t = \mu + \varepsilon_t \quad \text{where} \quad \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim t(v) \tag{59}$$

where v is the degrees of freedom and where Equation 57 is unchanged.

Another popular variation of the GARCH model is the AR-GARCH-t model, which makes the same distributional assumption as the GARCH-t model but additionally assumes that the conditional mean follows an autoregressive process. More precisely, the general form of a AR-GARCH-t model is:

$$\begin{aligned} r_t = \mu_t + \varepsilon_t \quad \text{where} \quad & \mu_t = \delta_0 + \delta_1 r_{t-1} + \cdots + \delta_r r_{t-r} \\ & \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim t(v) \end{aligned} \tag{60}$$

where δ_k is the coefficient of the lag-r regressor and v is the degrees of freedom, and where Equation 57 is unchanged.

3.1.2 EGARCH

Nelson (1991)'s exponential generalized autoregressive conditional heteroskedastic (EGARCH) model is a generalization of the GARCH model that is appropriate when positive and negative shocks of equal magnitude do not contribute equally to volatility. An EGARCH model makes the same distributional assumption as the GARCH model, and posits that the conditional variance σ_t^2 is the sum of four linear processes, each with coefficients for each term: an ARCH polynomial, i.e. magnitudes of past standardized innovations; a GARCH polynomial, i.e. the past logged conditional variances; a leverage polynomial, i.e. the past standardized innovations; and the constants μ and ω for the innovation mean and the conditional variance models, respectively.

More precisely, the general form of an EGARCH(q,p) model is:

$$r_t = \mu + \varepsilon_t \quad \text{where} \quad \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim N(0,1) \equiv \varepsilon_t|\psi_{t-1} \sim N(0,\sigma_t^2) \tag{61}$$

$$log(\sigma_t^2) = \omega + \sum_{i=1}^{q} \alpha_i \left\{ \frac{|\varepsilon_{t-i}|}{\sigma_{t-i}} - E\left[\frac{|\varepsilon_{t-i}|}{\sigma_{t-i}}\right] \right\} + \sum_{j=1}^{p} \beta_j log(\sigma_{t-j}^2) + \sum_{i=1}^{q} \gamma_i \frac{\varepsilon_{t-i}}{\sigma_{t-i}} \tag{62}$$

where, since z_t is Gaussian, then:

$$E\left[\frac{|\varepsilon_{t-i}|}{\sigma_{t-i}}\right] = E[\ |z_{t-i}|\ ] = \sqrt{\frac{2}{\pi}} \tag{63}$$

and where all roots of the GARCH lag operator polynomial $(1 - \beta_1 L - \cdots - \beta_p L^p)$ must lie outside of the unit circle to guarantee the stationarity of the model. Unlike the GARCH (or the GJR), the EGARCH models the logarithm of the variance, which allows to relax the positivity constrains. Lastly, it is important to notice that the forecasts of the conditional variances are biased, because by Jensen's inequality $E(\sigma_t^2) \geq \exp\{\ E[log(\sigma_t^2)]\ \}$.

Equation 62 shows how the effect of the shock ε_{t-i}^2 is symmetric via $|\varepsilon_{t-i}^2|$, i.e. both positive and negative shocks have the same impact on volatility. However, the leverage polynomial modifies this to make the effect asymmetric. In particular, when $\gamma_i < 0$ the negative shock has a bigger impact on volatility than a positive shock of equal magnitude.

Like the GARCH model, variations of the EGARCH include models that make different assumptions about the distribution of the error term, such as the EGARCH-t model, which assumes a t-distribution, in which case, since $z_t \sim t(v)$, then:

$$E\left[\frac{|\varepsilon_{t-i}|}{\sigma_{t-i}}\right] = E[\ |z_{t-i}|\ ] = \sqrt{\frac{v-2}{\pi}}\ \frac{\Gamma\left(\frac{v-1}{2}\right)}{\Gamma\left(\frac{v}{2}\right)} \tag{64}$$

and models that make different assumptions about the dynamics of the conditional mean, such as the AR-EGARCH-t, which assumes it to follow an autoregressive process.

3.1.3 GJR GARCH

Glosten, Jagannathan, and Runkle (1993)'s GARCH (GJR) model is a generalization of the GARCH model that is appropriate to model asymmetric volatility clustering; specifically, when negative shocks contribute more to volatility than positive shocks. A GJR model makes the same distributional assumption as the GARCH model, and posits that the conditional variance σ_t^2 is the sum of four linear processes, with coefficient for each term: an ARCH polynomial, i.e. the past squared innovations $\{\varepsilon_{t-1}^2, \ldots, \varepsilon_{t-q}^2\}$; a GARCH polynomial, i.e. the past conditional

variances $\{\sigma^2_{t-1}, \ldots, \sigma^2_{t-p}\}$; a leverage polynomial, i.e. the past squared negative innovations; and the constants μ and ω for the innovation mean and the conditional variance models, respectively. More precisely, the general form of a GJR(q,p) model is:

$$r_t = \mu + \varepsilon_t \quad \text{where} \quad \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim N(0,1) \equiv \varepsilon_t|\psi_{t-1} \sim N(0, \sigma^2_t) \tag{65}$$

$$\begin{aligned} \sigma^2_t &= \omega + \sum_{i=1}^{q} \alpha_i \varepsilon^2_{t-i} + \sum_{j=1}^{p} \beta_j \sigma^2_{t-j} + \sum_{i=1}^{q} \gamma_i \mathbb{1}_{[\varepsilon_{t-i}<0]} \varepsilon^2_{t-i} \\ &= \omega + \alpha_1 \varepsilon^2_{t-1} + \cdots + \alpha_q \varepsilon^2_{t-q} + \beta_1 \sigma^2_{t-1} + \cdots + \beta_p \sigma^2_{t-p} + \\ &\quad + \gamma_1 \mathbb{1}_{[\varepsilon_{t-1}<0]} \varepsilon^2_{t-1} + \cdots + \gamma_q \mathbb{1}_{[\varepsilon_{t-q}<0]} \varepsilon^2_{t-q} \end{aligned} \tag{66}$$

where ψ_{t-1} denotes information available up to time $t-1$, $\omega > 0$, $\alpha_i, \beta_j \geq 0$ and $\alpha_i + \gamma_i \geq 0$ guarantees that $\sigma^2_t > 0$ for all periods, and $\sum_{i=1}^{q} \alpha_i + \sum_{j=1}^{p} \beta_j + 1/2 \sum_{i=1}^{q} \gamma_i < 1$ guarantees that σ^2_t is stationary (and therefore the unconditional variance is finite).

Equation 66 shows how the effect of the shock ε^2_{t-i} is α_i if the shock is negative and $\alpha_i + \gamma_i$ if the shock is positive. Therefore, when $\gamma_i < 0$ the negative shock has a bigger impact on volatility than a positive shock of equal magnitude. Moreover, if all leverage coefficients are zero, then the GJR model is equivalent to the GARCH model. Finally, the GJR can be estimated with maximum likelihood estimation using Equation 58.

Like the GARCH model, variations of the GJR include models that make different assumptions about the distribution of the error term, such as the GJR-t model, which assumes a t-distribution, and models that make different assumptions about the dynamics of the conditional mean, such as the AR-GJR-t, which assumes it to follow an autoregressive process.

3.2 Traditional Quantile Regression

For a probability level $\tau \in (0,1)$, the τ-quantile of a probability density function $p(y)$ is defined as $q = F^{-1}(\tau)$ where $F(y)$ is the cumulative distribution function of $p(y)$. Quantile regression aims to estimate the τ-quantile q of the conditional distribution $p(y|\boldsymbol{x})$ by means of a parametric function $q = f_\theta(\boldsymbol{x})$, rather than making distributional assumptions. Since q is unobservable, the pinball loss function makes the estimation feasible by returning a value that can be interpreted as the accuracy of a quantile forecasting model. Let y be the real value and q the quantile forecast, then:

$$L_\tau(y,q) = \begin{cases} \tau|y-q| & \text{if } y > q \\ (1-\tau)|y-q| & \text{if } y \leq q \end{cases} \tag{67}$$

The parameter estimate $\hat{\theta}$ is found by minimizing the expected loss function:

$$\min_{\theta} \mathbb{E}[L_\tau(y, q)] = \min_{\theta} \mathbb{E}[L_\tau(y, f_\theta(\boldsymbol{x}))] \tag{68}$$

For a finite a dataset $\{x_i, y_i\}_{i=1}^N$ this is equivalent to minimizing the average loss:

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} L_\tau(y_i, f_\theta(x_i)) \tag{69}$$

To compute multiple conditional quantiles $\{q_1, \ldots, q_K\}$ corresponding to the probability levels $\{\tau_1, \ldots, \tau_K\}$, K different parametric functions $q_k = f_{\theta_k}(\boldsymbol{x})$ are needed and the loss functions are summed up to be minimized simultaneously:

$$\min_{\theta_1,\ldots,\theta_K} \frac{1}{K} \frac{1}{N} \sum_{k=1}^{K} \sum_{i=1}^{N} L_{\tau_k}(y_i, f_{\theta_k}(x_i)) \tag{70}$$

Besides the increasing number of parameters K when estimating many quantiles, the problem with Equation 70 is that it may lead to quantile crossing, i.e. that for some $\boldsymbol{x}$ and $\tau_j < \tau_k$, it may happen that $f_{\theta_j}(\boldsymbol{x}) > f_{\theta_k}(\boldsymbol{x})$. This is caused by the fact that θ_j and θ_k are estimated independently. In fact, the quantile crossing issue could be overcome by adding constraints on the monotonicity of the quantiles during the optimization or by post-processing, i.e. rearranging the estimated quantiles to be monotone. Finally, one last shortcoming is that the mapping from $\boldsymbol{x}$ to the conditional quantile has no interpretability.

3.3 Heavy Tailed Quantile Function

Besides the probability density function and the cumulative distribution function, quantile function is one way to fully describe a continuous distribution. Therefore, in order to overcome the shortcomings of the traditional quantile regression, as well as to properly account for the empirical behavior of financial assets, Yan et al. (2018) introduced a parametric quantile function that allows for leptokurtic, asymmetric and time-varying tails in the distribution of stock returns. The idea behind their model starts with a Q-Q plot, which is a method to determine whether a sample of observations is drawn from a normal distribution or not. Since the τ-quantile of a normal distribution $N(\mu, \sigma^2)$ is $q = \mu + \sigma Z_\tau$, where Z_τ is the τ-quantile of a standard normal distribution, the Q-Q plot forms a straight line when τ takes different values. On the other hand, a Q-Q plot figuring an inverted S shape indicates an heavy tailed distribution.

Yan et al. (2018) construct a parsimonious parametric quantile function, as a function of Z_τ, which allows to control the shape in the Q-Q plot against the standard normal distribution.

In addition the up and down tails of the inverted S shape in the Q-Q plot are controlled by two parameters u and d. The formulation of their heavy tail quantile function (HTQF) is as follows:

$$Q(\tau|\mu,\sigma,u,d) = \mu + \sigma Z_\tau \left(\frac{e^{uZ_\tau}}{A} + 1\right)\left(\frac{e^{-dZ_\tau}}{A} + 1\right) \tag{71}$$

where μ and σ are the location and scale parameters, Z_τ is the τ-quantile of a standard normal distribution, and $f_u(Z_\tau)$ and $f_d(Z_\tau)$ are two factors, where A is a relatively large positive constant, $u \geq 0$ controls the up tail of the inverted S shape in the Q-Q plot (i.e. the right tail of the distribution) and $d \geq 0$ controls the down tail (i.e. the left tail of the distribution). The larger the parameters u and d, the heavier the tails and if $u = d = 0$, the HTQF becomes the quantile function of the normal distribution. More precisely, the factor $f_u(Z_\tau)$ in monotonically increasing and convex in Z_τ, and satisfies $f_u(Z_\tau) \to 1$ as $Z_\tau \to -\infty$, so that $Z_\tau f_u(Z_\tau)$ exhibits the up tail of the inverted S shape and, by a similar reasoning, $Z_\tau f_d(Z_\tau)$ exhibits the down tail. It follows that $Z_\tau f_u(Z_\tau) f_d(Z_\tau)$ exhibits the entire inverted S shape in the Q-Q plot. The constant A is used to keep the value of $f_u(0)$ and $f_d(0)$ close to 1 and ensure that the HTQF is monotonically increasing in Z_τ. Figure 12 illustrates the Q-Q plots of a student-t distribution ($v = 2$) and two HTQFs ($u = 1.0$ and $d = 0.1$; $u = 0.6$ and $d = 1.2$; $A = 4$) against a standard normal distribution; for all three distributions, $\mu = 1$ and $\sigma = 1.5$.

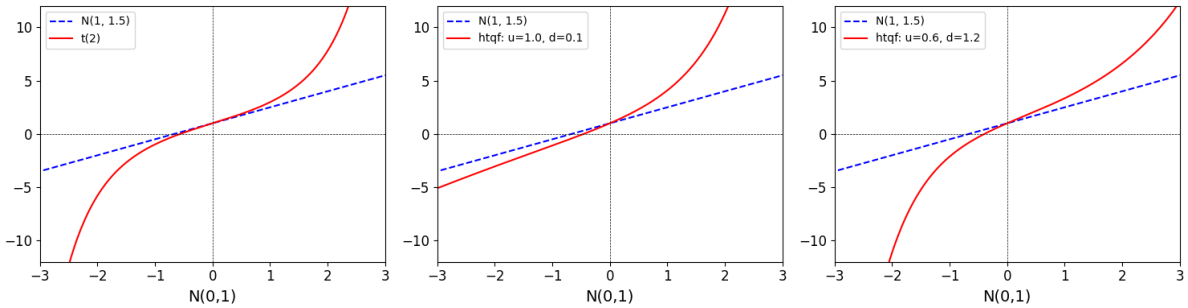

Figure 12: Q-Q plots of a student-t distribution and two HTQFs

3.4 Quantile Regression with HTQF

3.4.1 Classic LSTM-HTQF

As explained in the previous paragraphs, while the benchmark models make a specific distributional assumption for the density function of a conditional distribution $p(y|\boldsymbol{x})$, quantile regression is a method to forecast the quantiles of a conditional distribution by means of a parametric quantile function, without making any distributional assumption. Yan et al. (2018) predict the conditional quantiles of series of stock returns in a quantile regression framework,

that describes the conditional distribution $p(r_t|r_{t-1}, r_{t-2}, \dots)$ using a heavy tail quantile function, denoted by $Q(\tau|\mu_t, \sigma_t, u_t, d_t)$, where μ_t, σ_t, u_t and d_t are the time varying parameters of the distribution. Thus, the conditional τ_k-quantile of r_t can be calculated by substituting τ_k into the HTQF:

$$q_k^t = Q(\tau_k|\mu_t, \sigma_t, u_t, d_t) \quad \text{for} \quad k = 1, \dots, K \tag{72}$$

The time varying parameters should depend on the past series $\{r_{t-1}, r_{t-2}, \dots\}$. To model this, a subsequence $\{r_{t-1}, \dots, r_{t-L}\}$ of fixed length L is extracted from the series of past returns to construct a feature matrix $\boldsymbol{X}_t$ and apply a LSTM unit to it:

$$\boldsymbol{X}_t = \begin{bmatrix} r_{t-L} & (r_{t-L} - \bar{r}_t)^2 & (r_{t-L} - \bar{r}_t)^3 & (r_{t-L} - \bar{r}_t)^4 \\ \dots & \dots & \dots & \dots \\ r_{t-1} & (r_{t-1} - \bar{r}_t)^2 & (r_{t-1} - \bar{r}_t)^3 & (r_{t-1} - \bar{r}_t)^4 \end{bmatrix} \tag{73}$$

where $\bar{r}_t = \frac{1}{L}\sum_{i=1}^{L} r_{t-i}$. This feature matrix contains information for the first four central moments of the past L returns. It is now possible to model the HTQF parameters μ_t, σ_t, u_t and d_t as the output of a LSTM unit with input $\boldsymbol{X}_t$:

$$\big[\mu_t, \sigma_t, u_t, d_t\big]^\top = tanh(\boldsymbol{a}_{(h),t}\boldsymbol{W}_{(out)} + \boldsymbol{b}_{(out)}) \quad \text{where} \quad \boldsymbol{a}_{(h),t} = LSTM_\Theta(\boldsymbol{X}_t) \tag{74}$$

where $\boldsymbol{W}_{(out)}, \boldsymbol{b}_{(out)}$ are the output layer parameters, $\boldsymbol{a}_{(h),t}$ is the last hidden state of the LSTM and Θ represents the parameters of the LSTM.

Lastly, for multiple probability levels $\{\tau_1, \dots, \tau_K\}$, the loss functions between r_t and the conditional quantile $q_k^t = Q(\tau_k|\mu_t, \sigma_t, u_t, d_t)$ are summed up to be minimized simultaneously, like it is done in traditional quantile regression:

$$\min_{\Theta, \boldsymbol{W}_{(out)}, \boldsymbol{b}_{(out)}} \frac{1}{K}\frac{1}{s-L}\sum_{k=1}^{K}\sum_{t=L+1}^{s} L_{\tau_k}\big[r_t, Q(\tau_k|\mu_t, \sigma_t, u_t, d_t)\big] \tag{75}$$

The complete quantile regression model with an heavy tail quantile function and a long short term memory unit, denominated Classic LSTM-HTQF, combines Equations 71, 73, 74, 75. After training, given a new series $\{r_{t'}\}_{t'=1}^{s'}$, the time varying parameters $\mu_{t'}, \sigma_{t'}, u_{t'}$ and $d_{t'}$ of the HTQF can be calculated directly with the learned model parameters $\hat{\Theta}, \hat{\boldsymbol{W}}_{(out)}, \hat{\boldsymbol{b}}_{(out)}$. Conditional quantiles $q_k^{t'}$ can then be predicted and the summed loss can be calculated again to test the performance of the new series.

3.4.2 Volume LSTM-HTQF

The first variation of the Classic LSTM-HTQF is the Volume LSTM-HTQF model. The latter deviates from the former only in the construction of the feature matrix $\boldsymbol{X}_t$, which is expanded to include data on the trading volume v_t associated with each return r_t:

$$\boldsymbol{X}_t = \begin{bmatrix} r_{t-L} & (r_{t-L} - \bar{r}_t)^2 & (r_{t-L} - \bar{r}_t)^3 & (r_{t-L} - \bar{r}_t)^4 & v_{t-L} \\ \dots & \dots & \dots & \dots & \dots \\ r_{t-1} & (r_{t-1} - \bar{r}_t)^2 & (r_{t-1} - \bar{r}_t)^3 & (r_{t-1} - \bar{r}_t)^4 & v_{t-1} \end{bmatrix} \tag{76}$$

where we consider v_t to be the volume associated with the price p_t that produces a return $r_t = log(p_t/p_{t-1})$.

The rationale behind this variation is that the feature matrix $\boldsymbol{X}_t$ should be designed to contain as much information as possible related to the conditional distribution $p(r_t|r_{t-1}, r_{t-2}, \dots)$. Besides the trading volume, potential candidates would include the correlated assets or metrics of fundamentals of the stock. However, unlike the latter, the trading volume is easily accessible via the database employed in this study.

3.4.3 Volatility LSTM-HTQF

The second variation of the Classic LSTM-HTQF is the Volatility LSTM-HTQF model. Like the Volume LSTM-HTQF model, the latter differs from the former only in the construction of the feature matrix $\boldsymbol{X}_t$, which is expanded to include data on the realized volatility σ_t of the returns over a period of four weeks (generally equivalent to 20 trading days):

$$\boldsymbol{X}_t = \begin{bmatrix} r_{t-L} & (r_{t-L} - \bar{r}_t)^2 & (r_{t-L} - \bar{r}_t)^3 & (r_{t-L} - \bar{r}_t)^4 & \sigma_{t-L} \\ \dots & \dots & \dots & \dots & \dots \\ r_{t-1} & (r_{t-1} - \bar{r}_t)^2 & (r_{t-1} - \bar{r}_t)^3 & (r_{t-1} - \bar{r}_t)^4 & \sigma_{t-1} \end{bmatrix} \tag{77}$$

where $\sigma_t = \frac{1}{L'} \sum_{i=1}^{L'} (r_{t-i} - \bar{r})^2$ with $\bar{r} = \frac{1}{L'} \sum_{i=0}^{L'} r_{t-i}$ and where L' represents the length of the sequence of returns in the past four weeks.

The rationale behind this variation is to create a hybrid model between the Volume LSTM-HTQF and the Classic LSTM-HTQF, that adds an additional feature (the volatility), as well as additional information (the returns from further in the past), but still uses only information that can be derived from the time series of prices (or equivalently returns).

4 Data

In this section, we describe the database used in the study, the samples of stocks adopted and the data cleaning and data manipulation processes applied to obtain a dataset suitable to the ensuing analysis.

4.1 Trade and Quote Database

The Trade and Quote (TAQ) database contains historical tick-by-tick trade and quote data for all securities listed and traded on the New York Stock Exchange, the Nasdaq Stock Market and all other U.S. equity exchanges which are part of the U.S. National Market System.

The TAQ database contains a TAQ Daily Product and a TAQ Monthly Product, which are nearly identical: the former covers the period 10/09/2003 - present, is delivered day-by-day and has time stamps with precision of a millisecond through March 2015 and of a microsecond starting in April 2015; the latter covers the period 01/01/1993 - 31/12/2014, is delivered one month at the time and has time stamps with precision of a second. The time stamps are based on the New York Eastern Time and include changes between standard and daylight savings time. Transactions reported outside of the consolidated tape hours [6] and transactions on NYSE listed securities between 8:00 AM and 9:30 AM by other markets are not included.

Because of the higher precision of its time stamps and its more convenient distribution, we used the TAQ Daily Product, which is derived from the output of the Consolidated Tape Association (CTA) and the Unlisted Trading Privileges (UTP) Securities Information Processors (SIPs), and which is available though the Wharton Research Data Services (WRDS) Cloud server. Due to the large volume of data, the TAQ series is divided by year and month into distinct directories. Each folder contains the following SAS files: the trade files, which contain data on the orders executed on an exchange; the quote files, which contain data on the best trading conditions available on an exchange; the national best bid and offer files, which contain data on the highest bid and the lowest offer from all prevailing quotes for each stock; the corresponding index files, which are used to increase the computational speed; and the master files, which contain data on all securities in the TAQ datasets. A more thorough description of the TAQ Daily Product is available on the WRDS website (see also Nasdaq (2015)).

Focusing on the trade files of the Daily TAQ Product, the tables contain the following data for every trade reported in the consolidated tape by all CTA and UTP participants: date of the trade, time at which the trade was published by the SIP, exchange where the trade occurred,

[6] As of August 2000, consolidated tape hours were from 8:00 AM until 6:30 PM; as of 04/03/2004 from 4:00 AM until 6:30 PM.

root and suffix of the ticker, sale condition, number of shares traded, dollar price per share, correction indicator, message sequence number, and other information. All tables are sorted by symbol root and suffix, time, and sequence number. Table 1 displays a few sample records for Apple from the TAQ trade files.

date	time_m	ex	sym_root	sym_suffix	tr_scond	size	price	tr_corr	tr_seqnum
2019-03-28	09:38:00.273000	Q	AAPL	None	@	100	188.8200	00	86742
2019-03-28	09:38:00.515000	D	AAPL	None	@ I	9	188.8189	00	86781
2019-03-28	09:38:00.971000	B	AAPL	None	@	305	188.7900	00	86830

Table 1: Sample Trade Records for Apple

4.2 Samples

The sample adopted for the core part of this study is made up of tick-by-tick traded prices and volumes for ten highly liquid stocks listed on the Nasdaq Stock Market (NASDAQ) and the New York Stock Exchange (NYSE): Apple (AAPL), Advanced Micro Devices (AMD), Amazon.com (AMZN), Cisco Systems (CSCO), Facebook (FB), Intel Corporation (INTC), JPMorgan Chase & Co (JPM), Microsoft Corporation (MSFT), Nvidia Corporation (NVDA) and Tesla (TSLA).

The NASDAQ trading schedule sets the regular trading hours from 9:30 a.m. to 4.00 p.m., and the pre- and after-market trading hours from 4:00 a.m to 9:30 a.m. and from 4.00 p.m. to 8.00 p.m. respectively. Similarly, the NYSE core trading session is from 9:30 a.m. to 4.00 p.m., and the pre- and after-market trading hours are from 7:00 a.m to 9:30 a.m. and from 4.00 p.m. to 8.00 p.m. respectively. As visible in Figure 13, which illustrates 20 seconds of trade records for Apple at the opening and closing of the regular trading hours, trades are present both before and after this time, although with different frequencies. More interestingly, the plot also shows the opening and closing trades, which are marked with a square. Although one would expect them to be the first and last trades within the regular trading hours, this is clearly not the case: in fact, trading starts and ends some time after the official opening and closing times, making the the length of the trading session random. Moreover, the first and last five minutes of the

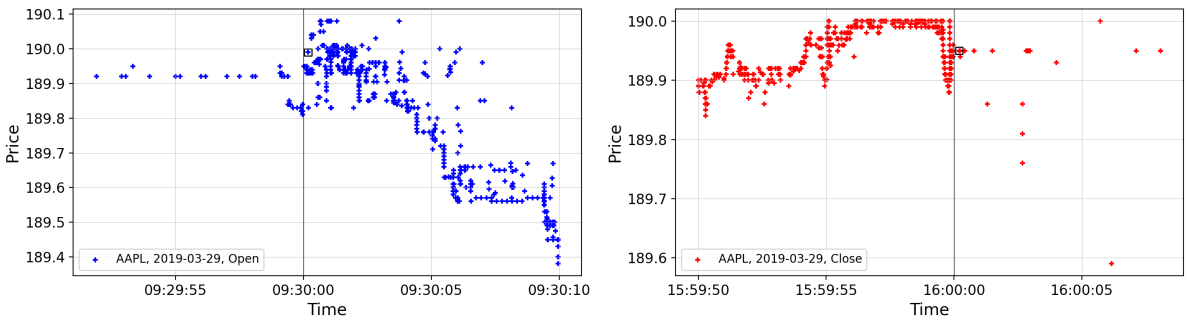

Figure 13: Opening and closing of regular trading hours for Apple

trading of each session contain a high number of trades which would be discarded by the data cleaning process introduced in the next section. To obviate these issues and ensure that the full extent of trading day is covered, we decided to consider a fixed time range from 9.35 a.m. to 15.55 p.m. Lastly, for each security, we consider a time span of 20 weeks, from March 4, 2019 to July 19, 2019 to execute the Classic LSTM-HTQF and the Volume LSTM-HTQF models, and a time span of 24 weeks, from February 4, 2019 to July 19, 2019 to execute the Volatility LSTM-HTQF model.

For comparative purposes, we also implemented the Classic LSTM-HTQF on the daily prices of five stock market indexes used in the original paper by Yan et al. (2018): the S&P 500 Index (SPX500), the NASDAQ-100 Index (NASDAQ100), the Nikkei 225 Index (NIKKEI225), the Deutscher Aktienindex Index (DAX) and the Hang Seng Index (HSI). For each stock, the authors use all prices available up until July 2, 2018, thus obtaining time series of different lengths with 17'235 observations for SPX500, 8'257 observations for NIKKEI225, 7'704 observations for DAX and 13'165 observations for NIKKEI225.

4.3 Data Cleaning and Data Manipulation

As explained in Brownlees and Gallo (2006), the structure of tick data is affected by several factors, such as the regulatory framework (e.g. the length of the trading day), the procedures of the institution that collects and produces the information (e.g. physical vs hybrid markets), and the technological improvements (e.g. changes in the precision of the tick). Furthermore, the series of tick data might contain errors, which are not identified as such by the data provider, might not be time ordered, and might be affected by unusual market conditions (e.g. opening and closing time or trading halts). Therefore, tick data on financial markets require the application of standard filters to remove bad record, as well as methods to remove outliers, which do not correspond to plausible market activity, and to aggregate data into time series suitable to conduct an analysis. When studying ultra-high-frequency measures of volatility, these data cleaning and data manipulation processes are particularly relevant to correct the impact of unwanted observations, which would otherwise result in wrong estimates of the coefficients and erroneous behaviors of the models. Nonetheless, these procedures involve subjective decisions, which result in different trade-offs between the loss of information contained in the discarded data and the noise reduction.

The data preparation process starts with the data cleaning process. As a first step, we keep only regular trades, which have not been corrected, changed or marked as cancel or error, i.e. all records with a correction indicator (tr_corr) field equal to 00. We remove all original

trades which have later been corrected (01), outright cancelled (08), cancelled due to error (07) or cancelled due to symbol correction (09), and the corresponding correction (12), cancel (10) and error (11) records, and symbol corrections (02) (see NYSE (2012), pg. 19 and NYSE (2017), pg. 19). As a second step, we keep only trades with a regular timing, i.e. all records with a sale condition (tr_scond) equal to @, A to F, H to K, M to O, Q to S, V, W, Y or 1 to 9. We remove all trades which have been reported late (G, L, Z), executed outside the regular trading hours (T, U, X) or that refer to conditions at an earlier point in the trading session (P) (see NYSE (2012), pg. 20-23, NYSE (2017), pg. 17-19 and Nasdaq (2018), pg. 41-43). As a third step, we remove all observations that are inconsistent with the current market activity. To identify these outliers, data series other than the tick-by-tick price are of little help. For example, while quotes cannot be used because it is very difficult to match trades and quotes with precision, as explained in Vergote (2005), volumes cannot be used either because their plausibility cannot be assessed beyond the plausibility of the corresponding price. Therefore, we adopt a variation of the heuristic procedure outlined in Brownlees and Gallo (2006). This identifies ouliers by focusing on prices and calculating their relative distance from a neighborhood of closest valid observations. Let $\{p_t^i\}_{t=0}^s$ be the time-ordered tick-by-tick price series of stock i. The rule used to identify outliers is:

$$|p_t^i - \bar{p}_t^i(k)| < 3s_t^i(k) + \gamma = \begin{cases} \text{if true, then observation } p_t^i \text{ is kept} \\ \text{if false, then observation } p_t^i \text{ is removed} \end{cases} \tag{78}$$

where $\bar{p}_t^i(k)$ and $s_t^i(k)$ are the δ-trimmed sample mean and sample standard deviation of a neighborhood of k observations around p_t^i and γ is a granularity parameter. More precisely, the neighborhood is defined so that every price observation is compared with observations from the same trading day: the neighborhood of the first/last $(k-1)/2$ observations are the first/last k observation of the day and the neighborhood of a generic observation in the middle of the day is made up of the preceding and following $(k-1)/2$ observations. The granularity parameter ensures that the right-hand side of the inequality is not zero in case the whole neighborhood contains equal prices. Brownlees and Gallo (2006) argue that the lower the level of trading activity, the smaller k should be, so that the rolling window does not contain too distant prices, and that γ should be a multiple of the minimum price variation. They then test different combinations of k and γ and choose the one that yields better results from a visual inspection of the plot. Following their methodology, we set the trimming parameter constant to $\delta = 10\%$ and we identify the outliers using the heuristic procedure outlined above, for all possible combinations of $k \in \{41, 61, 81, 101\}$ and $\gamma \in \{0.02, 0.04, 0.06\}$. For each stock, we then select the

combination of k and γ that identifies the minimum number of outliers. Indeed, after testing it on a sample of ten stocks and ten time spans of different length, this rule proves to be more robust than, say, selecting the combination of k and γ that identifies the maximum number of outliers or the one that yields the highest or lowest average distance between the observations identified as outliers and the corresponding trimmed sample mean. Moreover, this rule allows to automate the parameter selection process and removes the need for a visual inspection.

Following the three steps of the data cleaning process, we implement the data manipulation process. Indeed, the time series still display an undesired feature, namely the presence of transactions executed at the same time but at different prices. This can be explained by the fact that some securities can trade simultaneously on different exchanges, that the execution of one trade can sometimes generate multiple transaction reports, or that the execution of non-simultaneous trades can be reported at the same time due to reporting approximations. Therefore, as a fourth step, we aggregate blocks of simultaneous observations retaining the median value of the prices and the sum of all the corresponding volumes, so to obtain one observation per time stamp. Finally, the time series displays one last undesired feature, namely the presence of irregular time intervals between subsequent observations, as dictated by variations in the trading frequency. Therefore, as a fifth step, we resample the data at lower frequency, retaining the last observation in the interval and again adding up all the corresponding volumes, so to obtain time series with equally spaced time intervals. To choose the appropriate resampling frequency, we computed the ratio of the number of time intervals with no observation over the total number of time intervals for a sample of highly liquid stocks. We found 5 seconds to be an appropriate frequency to balance the need for interpolation and the information content provided by high frequency data. Moreover, although some exceptionally liquid stocks would allow to push the resampling frequency up to 2 seconds with no significant increase in the need for interpolation, choosing 5 second intervals allows to make the data extraction process more robust. For example, even for highly liquid stocks such as Amazon or Tesla, a resampling frequency of 2 seconds would result in 10% - 20% of the intervals not containing an observation on some days with a low trading volume, while a resampling frequency of 5 seconds drastically reduces this number to 1% on such days. Finally, we applied a linear interpolation (coupled with next and previous point interpolation for the beginning and end of the daily sequence respectively) to fill in the missing data.

These five sequential steps allow us to obtain a time series of prices and volumes for each stock, that can be processed by the models presented in the following sections. Figure 14 shows 10 minutes of trade records for Apple, before and after the five step of the data cleaning

and data manipulation processes. It is clearly visible that our process successfully removes the outliers and creates a smoother sequence of evenly spaced price data.

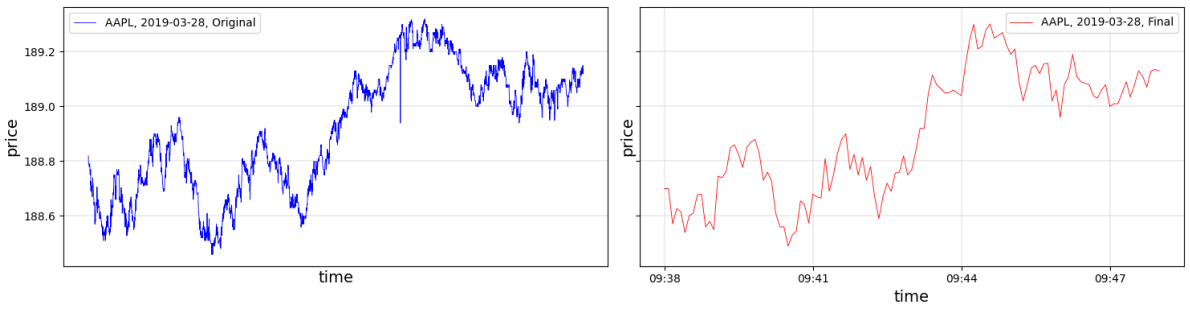

Figure 14: Sample trade records for Apple before and after the cleaning and data manipulation processes

We would like to point out that the choices of the parameters made in the data cleaning and data manipulation processes do not excessively impact the time series creation. Indeed, for very liquid stocks, a neighborhood of high frequency observations is very dense and different choices would yield similar results.

As a final note, we suggest to use price data, rather than the best bid and ask quotes, as the data on the latter (contained in the TAQ dataset) do not include information on the quality of the reported data. This would make the data cleaning process presented in this section unfeasible.

5 Empirical Study

In this section we illustrate the data preprocessing process used to prepare the data for the machine learning and the benchmark models, we explain the implementation of the three machine learning models (Classic LSTM-HTQF, Volume LSTM-HTQF, Volatility LSTM-HTQF) and the nine benchmark models (GARCH, EGARCH, GJR and variations), and we discuss the results.

5.1 Data Preprocessing

The data preprocessing processes implemented for the Classic LSTM-HTQF, the Volume LSTM-HTQF and Volatility LSTM-HTQF are nearly identical and only vary in the number and type of features considered when building the feature matrices. Therefore, for the sake of simplicity, we illustrate this process for the Classic LSTM-HTQF model.

As a first step, for each stock i we use the time series of prices $\{p_t^i\}_{t=0}^s$ to create a time series of continuous returns $\{r_t^i\}_{t=1}^s$ where $r_t^i = log(p_t^i/p_{t-1}^i)$, and where the first return of the day is computed using the first two price observations of the same day. In other words, we only compute intra-day returns, and purposefully disregard the overnight return, i.e. the return between the first price observation of one day and the last price observation of the previous day. Indeed, if we assume that the stock price follows the geometric Brownian motion:

$$dP = \mu P dt + \sigma P dz \quad \text{where} \quad dz = \epsilon\sqrt{dt} \tag{79}$$

then by Itô's lemma it follows that the distribution of the continuously compounded rate of return earned between time t' and t is:

$$log\Big(\frac{p_{t'}}{p_t}\Big) \sim N\Big[\Big(\mu - \frac{\sigma^2}{2}\Big)(t'-t), \sigma\sqrt{t'-t}\Big)\Big] \tag{80}$$

This shows how the overnight return has a different distribution than the 5 second return we are considering and, for this reason, we disregard it (see Hull (2015), ch. 12).

As a second step, for each asset i we create the vector of targets $\boldsymbol{y}^i \in \mathbb{R}^{(s\text{-}L\ x\ 1)}$, i.e. the time series of continuous returns $\{r_t^i\}_{t=L+1}^s$ for $L = \{100, 200\}$:

$$\boldsymbol{y}^i = \begin{pmatrix} r_{L+1}^i \\ r_{L+2}^i \\ \cdots \\ r_s^i \end{pmatrix} \tag{81}$$

We then use the past L returns of each target return r_t^i in $\boldsymbol{y}^i$ to construct its feature matrix $\boldsymbol{X}_t^i \in \mathbb{R}^{(L\ x\ 4)}$ as explained in Equation 73, and we create a three-dimensional array $\boldsymbol{X}^i \in \mathbb{R}^{(s\text{-}L\ x\ L\ x\ 4)}$ that stacks all feature matrices:

$$\boldsymbol{X}^i = \begin{pmatrix} \boldsymbol{X}_{L+1}^i \\ \boldsymbol{X}_{L+2}^i \\ \cdots \\ \boldsymbol{X}_s^i \end{pmatrix} \tag{82}$$

Figure 15 shows a visual representation of the three-dimensional array containing the feature matrices.

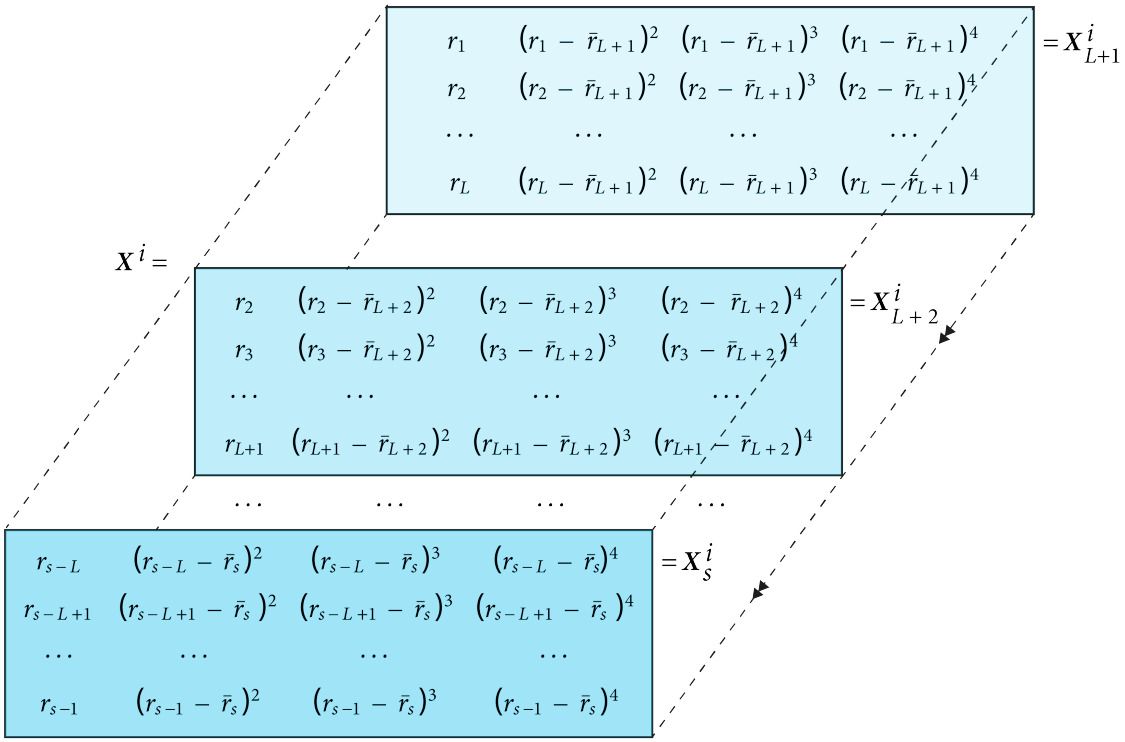

Figure 15: 3D array of feature matrices

As a third step, for each stock i, we split the vector of targets $\boldsymbol{y}^i$ and the array of feature matrices $\boldsymbol{X}^i$ into training, validation and test subsets, with the proportion 8:1:1, thus producing the subsets $\boldsymbol{X}_{train}^i \in \mathbb{R}^{(0.8(s\text{-}L)\ x\ L\ x\ 4)}$, $\boldsymbol{X}_{valid}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ L\ x\ 4)}$, $\boldsymbol{X}_{test}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ L\ x\ 4)}$, $\boldsymbol{y}_{train}^i \in \mathbb{R}^{(0.8(s\text{-}L)\ x\ 1)}$, $\boldsymbol{y}_{valid}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$ and $\boldsymbol{y}_{test}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$.

Finally, we standardize all three subsets by z-scoring, i.e. demeaning and scaling them using the first two moments of the training subset. More precisely, we compute the mean and the standard deviation of each feature in the training subset and we use these same values to individually standardize each feature in the training, validation and test subsets (see Equation 14 for more details).

The data preparation process for the Volume LSTM-HTQF and the Volatility LSTM-HTQF only differ in that their feature matrices contain an additional feature, so that $\boldsymbol{X}_t^i \in \mathbb{R}^{(L\ x\ 5)}$ and $\boldsymbol{X}^i \in \mathbb{R}^{(s\text{-}L\ x\ L\ x\ 5)}$. More precisely, for the Volume LSTM-HTQF model, the additional feature is the volume v_t^i associated with return r_t^i as explained in Equation 76; while for the Volatility LSTM-HTQF model, the additional feature is the volatility σ_t^i realized over the four weeks preceding return r_t^i, as explained in Equation 77

5.2 Implementation of the Machine Learning Models

As for the data preprocessing process, the implementation of the machine learning model for the Classic LSTM-HTQF, the Volume LSTM-HTQF and the Volatility LSTM-HTQF are very similar, and only differ in the size and type of the feature matrices. Therefore, once again, we illustrate this process for the Classic LSTM-HTQF model.

Our machine learning model implements the LSTM recurrent neural network introduced in Section 3.4.1. Like Yan et al. (2018)'s original model, our model has two hyperparameters, namely the length L of the past series of returns $\{r_{t-1}, \ldots, r_{t-L}\}$ on which the time-t HTQF parameters μ_t, σ_t, u_t and d_t depend, and the number of units h in the hidden state of the LSTM unit. Moreover, we add an additional hyperparameter, namely the size of the batch bs used to feed the data to the machine learning model. The tuning of the hyperparameters is done on the following sets: $L \in \{100, 200\}$ (as already explained in section 5.1), $h \in \{16, 32\}$, and $bs \in \{100, 1/10$ of the size of the validation subset$\}$. Therefore, this amounts to running 8 models for each of the ten stocks in our sample.

As explained in section 5.1, for each stock i, we generated the training, validation and test subsets of the feature matrices and the target returns: $\boldsymbol{X}_{train}^i \in \mathbb{R}^{(0.8(s\text{-}L)\ x\ L\ x\ 4)}$, $\boldsymbol{X}_{valid}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ L\ x\ 4)}$, $\boldsymbol{X}_{test}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ L\ x\ 4)}$, $\boldsymbol{y}_{train}^i \in \mathbb{R}^{(0.8(s\text{-}L)\ x\ 1)}$, $\boldsymbol{y}_{valid}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$ and $\boldsymbol{y}_{test}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$. As a first step of the implementation of the machine learning model, we randomly shuffle the $0.8(s\text{-}L)$ elements in $\boldsymbol{X}_{train}^i$ and $\boldsymbol{y}_{train}^i$, maintaining the association between feature matrix and target return, and we create batches of size bs. This allows to build a training dataset made up of several batches, each containing bs feature matrices and bs corresponding target returns. We also repeat the batching (but not the shuffling) process and the dataset creation process on the validation and test subsets.

As a second step, we build the machine learning model in Equation 74 as a two layer model. The first layer is an LSTM recurrent neural network with h units, which is set up to return only the last output rather than the entire sequence of outputs. Since the model is trained using batches, we denote this output as $\boldsymbol{A}_{(h),t}^i \in \mathbb{R}^{(bs\ x\ h)}$, where the index t denotes the general

time step corresponding to the target return for each tuple of feature matrix and corresponding target return contained in the batch[7]. More precisely, adapting Equations 50 to 55, the output of the LSTM in the first layer is computes as follows:

$$\boldsymbol{F}_t^i = \sigma\big(\boldsymbol{A}_{(in),t}^i \boldsymbol{W}_{xf}^i + \boldsymbol{A}_{(h),t-1}^i \boldsymbol{W}_{hf}^i + \boldsymbol{B}_f^i\big) \tag{83}$$

$$\boldsymbol{I}_t^i = \sigma\big(\boldsymbol{A}_{(in),t}^i \boldsymbol{W}_{xi}^i + \boldsymbol{A}_{(h),t-1}^i \boldsymbol{W}_{hi}^i + \boldsymbol{B}_i^i\big) \tag{84}$$

$$\tilde{\boldsymbol{C}}_t^i = \tanh\big(\boldsymbol{A}_{(in),t}^i \boldsymbol{W}_{xc}^i + \boldsymbol{A}_{(h),t-1}^i \boldsymbol{W}_{hc}^i + \boldsymbol{B}_c^i\big) \tag{85}$$

$$\boldsymbol{O}_t^i = \sigma\big(\boldsymbol{A}_{(in),t}^i \boldsymbol{W}_{xo}^i + \boldsymbol{A}_{(h),t-1}^i \boldsymbol{W}_{ho}^i + \boldsymbol{B}_o^i\big) \tag{86}$$

$$\boldsymbol{C}_t^i = \big(\boldsymbol{F}_t^i \odot \boldsymbol{C}_{t-1}^i\big) \oplus \big(\boldsymbol{I}_t^i \odot \tilde{\boldsymbol{C}}_t^i\big) \tag{87}$$

$$\boldsymbol{A}_{(h),t}^i = \boldsymbol{O}_t^i \odot \tanh\big(\boldsymbol{C}_t^i\big) \tag{88}$$

where $\boldsymbol{A}_{(in),t}^i \in \mathbb{R}^{(bs\ x\ 4)}$ is the input at the current time step, $\boldsymbol{A}_{(h),t-1}^i \in \mathbb{R}^{(bs\ x\ h)}$ is the output of the LSTM at the previous time step, $\boldsymbol{W}_{xf}^i$, $\boldsymbol{W}_{xi}^i$, $\boldsymbol{W}_{xc}^i$, $\boldsymbol{W}_{xo}^i \in \mathbb{R}^{(4\ x\ h)}$ are the kernel weights, $\boldsymbol{W}_{hf}^i$, $\boldsymbol{W}_{hi}^i$, $\boldsymbol{W}_{hc}^i$, $\boldsymbol{W}_{ho}^i \in \mathbb{R}^{(h\ x\ h)}$ are the recurrent kernel weights, and $\boldsymbol{B}_f^i$, $\boldsymbol{B}_i^i$, $\boldsymbol{B}_c^i$, $\boldsymbol{B}_o^i \in \mathbb{R}^{(bs\ x\ h)}$ are the bases. By simple matrix multiplication it follows that $\boldsymbol{F}_t^i$, $\boldsymbol{I}_t^i$, $\tilde{\boldsymbol{C}}_t^i$, $\boldsymbol{O}_t^i$, $\boldsymbol{C}_t^i$, $\boldsymbol{A}_{(h),t}^i \in \mathbb{R}^{(bs\ x\ h)}$. Moreover, this shows how the recurrent neural network progressively learns a total of $4[(4\ x\ h)(h\ x\ h)(h)]$ parameters[8] each time it reads in a new batch during training.

The output of the first layer $\boldsymbol{A}_{(h),t}^i$ is then fed to the second layer, which contains a dense neural network with an hyperbolic tangent activation function:

$$\boldsymbol{A}_{(out),t}^i = tanh\big(\boldsymbol{A}_{(h),t}^i \boldsymbol{W}_{(out)}^i + \boldsymbol{B}_{(out)}^i\big) \tag{89}$$

where $\boldsymbol{A}_{(h),t}^i \in \mathbb{R}^{(bs\ x\ h)}$ is the output of the LSTM, $\boldsymbol{W}_{(out)}^i \in \mathbb{R}^{(h\ x\ 4)}$ is the weight matrix, $\boldsymbol{B}_{(out)}^i \in \mathbb{R}^{(bs\ x\ 4)}$ is the the basis matrix, and $\boldsymbol{A}_{(out),t}^i \in \mathbb{R}^{(bs\ x\ 4)}$ is the output of the dense neural network in the second layer. Here the number 4 represents the number of HTQF parameters rather than the number of features as in the previous layer. Moreover, this shows how the densely connected layer progressively learns $(h\ x\ 4) + (4)$ parameters. Finally, since the hyperbolic tangent function returns values between -1 and $+1$ we add the constants $[0, 1, 1, 1]$ to the outputs to obtain the time-t HTQF parameters μ_t^i, σ_t^i, u_t^i and d_t^i in the right range.

Now, the output matrix $\boldsymbol{A}_{(out),t}^i \in \mathbb{R}^{(bs\ x\ 4)}$ can be decomposed into four vectors that contain the estimated parameters for each of the bs tuples of feature matrix and target return: $\boldsymbol{\mu}_t^i \in$

[7]Note that this notation is consistent with the notation in Equation 73.

[8]The basis is computed once for all samples in the batch.

$\mathbb{R}^{(bs\ x\ 1)}$, $\boldsymbol{\sigma}_t^i \in \mathbb{R}^{(bs\ x\ 1)}$, $\boldsymbol{u}_t^i \in \mathbb{R}^{(bs\ x\ 1)}$, $\boldsymbol{d}_t^i \in \mathbb{R}^{(bs\ x\ 1)}$. Furthermore, we define a vector $\boldsymbol{\tau} \in \mathbb{R}^{(1\ x\ 21)}$ of probability levels and a vector $\boldsymbol{z} \in \mathbb{R}^{(1\ x\ 21)}$ of the corresponding quantiles for the standard normal distribution:

$$\boldsymbol{\tau} = \begin{pmatrix} 0.01 & 0.05 & 0.10 & \dots & 0.90 & 0.95 & 0.99 \end{pmatrix} \tag{90}$$

$$\boldsymbol{z} = \begin{pmatrix} z_{0.01} & z_{0.05} & z_{0.10} & \dots & z_{0.90} & z_{0.95} & z_{0.99} \end{pmatrix} \tag{91}$$

Then the matrix of predicted quantiles $\boldsymbol{Q}_t^i \in \mathbb{R}^{(bs\ x\ 21)}$ for each tuple in the batch and for each probability level is calculated according to Equations 71 and 72 as follows[9]:

$$\boldsymbol{Q}_t^i = \boldsymbol{\mu}_t^i + \boldsymbol{\sigma}_t^i \odot \boldsymbol{z} \odot \left(\frac{e^{\boldsymbol{u}_t^i \boldsymbol{z}}}{A} + 1 \right) \odot \left(\frac{e^{\boldsymbol{d}_t^i \boldsymbol{z}}}{A} + 1 \right) \tag{92}$$

where A is arbitrarily set equal to 1.

The loss function used to train the model is the pinball loss function in Equation 75 and its value is computes as follows[10]:

$$l_t^i = \frac{1}{21} \frac{1}{bs} \sum_{k=1}^{21} \sum_{j=1}^{bs} \max \left[\boldsymbol{\tau} \odot (\boldsymbol{y}_t^i - \boldsymbol{Q}_t^i)\ ;\ (\boldsymbol{\tau} - 1) \odot (\boldsymbol{y}_t^i - \boldsymbol{Q}_t^i) \right] \tag{93}$$

where $\boldsymbol{y}_t^i \in \mathbb{R}^{(bs\ x\ 1)}$ is the vector of target returns in each batch, and where the loss value is a scalar for each batch fed to the machine learning model.

After building the machine learning model, we train it on the training subset $\boldsymbol{X}_{train}^i$ using batches and Adam optimization, which is a stochastic gradient descent method based on the adaptive estimation of first-order and second-order moments. As explained in Kingma and Ba (2014) this method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters". More precisely, we use the following suggested parameters: $\alpha = 0.001$ (the learning rate), $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-7}$. For every stock i, we run 10 epochs and we use the validation subset $\boldsymbol{X}_{valid}^i$ to fine tune the hyperparameters and to stop the training when the loss on the validation subset begins to increase irreversibly[11], to prevent overfitting.

Finally, we evaluate the performance of the model on the test set $\boldsymbol{X}_{test}^i$ using the same

[9] The use of the notation in Equation 92 is mathematically incorrect and assumes broadcasting of the values, a procedure commonly used by several programming languages: $\boldsymbol{z}$ is assumed to be broadcast bs times and $\boldsymbol{\mu}_t^i$ and $\boldsymbol{\sigma}_t^i$ are assumed to be broadcast 21 times so that $\mathbb{R}^{(bs\ x\ 1)} + \mathbb{R}^{(bs\ x\ 1)} \odot \mathbb{R}^{(1\ x\ 21)} \odot \mathbb{R}^{(bs\ x\ 21)} \odot \mathbb{R}^{(bs\ x\ 21)} \rightarrow \mathbb{R}^{(bs\ x\ 21)}$.

[10] Similarly to Equation 92, Equation 93 assumes broadcasting of the values: $\boldsymbol{y}_t^i$ is broadcast 21 times and $\boldsymbol{\tau}$ is broadcast bs times so that $\mathbb{R}^{(1\ x\ 21)} \odot (\mathbb{R}^{(bs\ x\ 1)} - \mathbb{R}^{(bs\ x\ 21)}) \rightarrow \mathbb{R}^{(bs\ x\ 21)}$.

[11] In order to do so, we run all machine learning models twice: the first time we run ten epochs and we observe when the validation loss starts to increase irreversibly; we then re-run the model with the preferred number of epochs.

pinball loss function. Like in the original study by Yan et al. (2018) we do so on two $\boldsymbol{\tau}$ sets: the full $\boldsymbol{\tau}$ set of Equation 90 and a set $\boldsymbol{\tau'} = \{0.01, 0.05, 0.1\}$, the quantiles of which are the VaR representing the downside risk.

The implementation of the machine learning model for the Volume LSTM-HTQF and the Volatility LSTM-HTQF only differ in that their feature matrices contain an additional feature, so that $\boldsymbol{X}_t^i \in \mathbb{R}^{(L\ x\ 5)}$, $\boldsymbol{X}^i \in \mathbb{R}^{(s\text{-}L\ x\ L\ x\ 5)}$, and $\boldsymbol{X}_{train}^i \in \mathbb{R}^{(0.8(s\text{-}L)\ x\ L\ x\ 5)}$, $\boldsymbol{X}_{valid}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ L\ x\ 5)}$, $\boldsymbol{X}_{test}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ L\ x\ 5)}$.

5.3 Implementation of the Benchmark Models

The implementation of the benchmark models for the GARCH, EGARCH and GJR (and their respective variations) is very similar. Therefore, for the sake of simplicity, we illustrate this process for the GARCH type models.

Our first class of benchmarks includes the GARCH, the GARCH-t and the AR-GARCH-t, which have been introduced in section 3.1.1. These three models have two hyperparameters, namely the values q and p of the maximum non-zero lags in the ARCH and GARCH polynomials, respectively. In addition, the AR-GARCH-t has a third hyperparameter, namely the number of largs r associated with the nonseasonal autoregressive polynomial coefficients. The tuning of the hyperparameters is done on the following sets: $q \in \{1, 2, 3, 4\}$, $p \in \{1, 2, 3, 4\}$, and $r \in \{1, 2, 3, 4\}$. To be precise, the length L of the past series of returns $\{r_{t-1}, \ldots, r_{t-L}\}$ represents an additional variable, as it influences the dimension of the vectors of target returns $\boldsymbol{y}_{train}^i \in \mathbb{R}^{(0.8(s\text{-}L)\ x\ 1)}$, $\boldsymbol{y}_{valid}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$ and $\boldsymbol{y}_{test}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$, on which the model is estimated and tested. Therefore, although the results change just slightly (given the small size L of the series of past returns compared to the size s of the whole series of returns), we still re-estimate all three models for $L \in \{100, 200\}$.

Differently from the machine learning models, the implementation of the benchmark models focuses only on the time series of target returns in the train, validation and test subsets, which we generated in section 5.1.

Starting with the GARCH model, the first step of the implementation of the benchmark models is the hyperparameter tuning. For each stock i and each combination of q and p, we create a new GARCH(q,p) model as defined in Equations 56 and 57, and we use maximum likelihood to estimate its parameters ω, $\{\alpha_i\}_{i=1}^q$, $\{\beta_j\}_{i=1}^p$ (setting $\mu = 0$), using the target returns $\boldsymbol{y}_{train}^i$ in the training subset. We then use the newly estimated model to infer the conditional variances $(\boldsymbol{\sigma}_{valid}^i)^2$ on the target returns $\boldsymbol{y}_{valid}^i$ in the validation set. By taking the square root of the values we obtain a vector of inferred conditional volatilities $\boldsymbol{\sigma}_{valid}^i \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$. Now,

we can compute the matrix of estimated quantiles $\boldsymbol{Q}^i_{valid} \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 21)}$ as follows:

$$\boldsymbol{Q}^i_{valid} = \boldsymbol{\sigma}^i_{valid}\boldsymbol{z} \tag{94}$$

where $\boldsymbol{z} \in \mathbb{R}^{(1\ x\ 21)}$ is the vector of quantiles for the standard normal distribution defined in Equation 91. We cal also calculate the pinball loss function as follows[12]

$$l^i_{valid} = \frac{1}{21}\frac{1}{0.1(s\text{-}L)}\sum_{k=1}^{21}\sum_{t=1}^{0.1(s\text{-}L)} \max\left[\boldsymbol{\tau} \odot (\boldsymbol{y}^i_{valid} - \boldsymbol{Q}^i_{valid})\ ;\ (\boldsymbol{\tau} - 1) \odot (\boldsymbol{y}^i_{valid} - \boldsymbol{Q}^i_{valid})\right] \tag{95}$$

where $\boldsymbol{\tau}$ is the vector of probability levels corresponding to the quantiles in $\boldsymbol{z}_\tau$. By re-evaluating Equation 95 for every combination of q and p, we are able to select the best combination of these hyperparameters p^* and q^* that minimizes the loss value on the validation set.

As a second step, we create a new GARCH(p^*, q^*) model, estimate it on the training subset $\boldsymbol{y}^i_{train}$, and use it to infer the conditional volatilities $\boldsymbol{\sigma}^i_{test} \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$ of the target returns $\boldsymbol{y}^i_{test}$ in the test subset. We then compute the matrix of estimated quantiles $\boldsymbol{Q}^i_{test} \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 21)}$ and the loss value l^i_{test}, exactly how it was done during the hyperparameter selection process. Finally, consistently with what we did during the implementation of the machine learning models, we compute the loss value both on $\boldsymbol{\tau}$ and $\boldsymbol{\tau}'$.

The implementation of the GARCH-t model differs in that it assumes that the error term follows a t-distribution, rather than a normal distribution, as explained in Equation 59. Therefore, the vector of inferred conditional volatilities $\boldsymbol{\sigma}^i_{valid}$ in the validation subset is computed as follows:

$$\boldsymbol{\sigma}^i_{valid} = \sqrt{\frac{(\boldsymbol{\sigma}^i_{valid})^2(v-2)}{v}} \tag{96}$$

where v represents the estimated degrees of freedom. The same holds for $\boldsymbol{\sigma}^i_{test}$. Moreover, the vector $\boldsymbol{z}$ now contains the quantiles of the t-distribution, rather than the normal distribution.

Lastly, the AR-GARCH-t model differs from the GARCH-t model in that it additionally assumes that the conditional mean follows an autoregressive process, as explained in Equation 60. To choose the best hyperparameters for this model, we create a new model for all possible combinations of r, p, and q, and we use maximum likelihood to estimate its parameters ω, $\{\alpha_i\}_{i=1}^q$, $\{\beta_j\}_{i=1}^p$, δ_0 and $\{\delta_k\}_{k=1}^r$, using the target returns $\boldsymbol{y}^i_{train}$ in the train subset. We then use the newly estimated model to infer the conditional volatilities $\boldsymbol{\sigma}^i_{valid}$ in the validation subset using Equation 96. However here we additionally infer the vector of residuals $\boldsymbol{\varepsilon}^i_{valid} \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$ so

[12] Similarly to Equation 93, Equation 95 assumes broadcasting of the values: $\boldsymbol{y}^i_{valid}$ is broadcast 21 times and $\boldsymbol{\tau}$ is broadcast $0.1(s\text{-}L)$ times so that $\mathbb{R}^{(1\ x\ 21)} \odot (\mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)} - \mathbb{R}^{(0.1(s\text{-}L)\ x\ 21))} \rightarrow \mathbb{R}^{(0.1(s\text{-}L)\ x\ 21)}$.

that we can compute the vector of mean values $\boldsymbol{\mu}^i_{valid} \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)}$ of the train set as follows:

$$\boldsymbol{\mu}^i_{valid} = \boldsymbol{y}^i_{valid} - \boldsymbol{\varepsilon}^i_{valid} \tag{97}$$

Now, we can compute the matrix of estimated quantiles $\boldsymbol{Q}^i_{valid} \in \mathbb{R}^{(0.1(s\text{-}L)\ x\ 21)}$ as follows[13]:

$$\boldsymbol{Q}^i_{valid} = \boldsymbol{\mu}^i_{valid} + \boldsymbol{\sigma}^i_{valid}\boldsymbol{z} \tag{98}$$

where the vector $\boldsymbol{z}$ now contains the quantiles of the t-distribution. With these changes, the hyperparameter selection process and the second step of the implementation of the benchmark mirror what we said for the GARCH-t model.

Similarly, the implementation of the EGARCH and GJR benchmark models (and their variations EGARCH-t, AR-EGARCH-t, GJR-t, AR-GJR-t) is identical to the one presented for the GARCH (and their variations GARCH-t, AR-GARCH-t), and is therefore not shown.

5.4 Results

In this section, first we discuss the hyperparameters that achieve the best performance for each of the three types machine learning models; then, we compare the performances of the latter among each other; finally, we compare the performances of the machine learning models against the nine benchmark models.

Tables 2, 3 and 4 show the values of the loss function evaluated on $\boldsymbol{\tau}$ and $\boldsymbol{\tau'}$ for the Classic LSTM-HTQF, the Volume LSTM-HTQF and the Volatility LSTM-HTQF respectively, and for all possible combinations of the original hyperparameters $L \in \{100, 200\}$ and $h \in \{16, 32\}$, and the additional hyperparameter $bs \in \{100, 1/10 \text{ of the size of the validation subset}\}$. For each stock, we mark in bold and highlight in gray the combination of hyperparameters that yields the best performance on $\boldsymbol{\tau}$ and $\boldsymbol{\tau'}$ respectively. In the following three paragraphs, unless specified otherwise, our focus is on the more complete loss function calculated on $\boldsymbol{\tau}$.

For the Classic LSTM-HTQF, six stocks (AAPL, AMZN, FB, JPM, MSFT, TSLA) achieve a better performance when considering a longer series of past returns $L = 200$, while only four stocks (AMD, CSCO, INTC, NVDA) do so with $L = 100$. Moreover, all stocks except two (CSCO, FB) achieve a better result with a larger number of units $h = 32$ in the hidden state of the LSTM. Focusing on $\boldsymbol{\tau'}$, we can see that only for four stocks (AAPL, CSCO, INTC, MSFT) the combination of hyperparameters that yields the best performance is the same as for $\boldsymbol{\tau}$. Finally, for both sets,

[13] Similarly to Equation 92, Equation 98 assumes broadcasting of the values: $\boldsymbol{\mu}^i_{valid}$ is broadcast 21 times so that $\mathbb{R}^{(0.1(s\text{-}L)\ x\ 1)} + \mathbb{R}^{(0.1(s\text{-}L)\ x\ 1))}\mathbb{R}^{(1\ x\ 21)} \rightarrow \mathbb{R}^{(0.1(s\text{-}L)\ x\ 21)}$.

Test loss for τ (%)

		AAPL		AMD		AMZN		CSCO		FB	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	14.8142	14.8703	18.1780	18.2593	17.8597	17.9378	**17.9438**	18.1036	17.0752	17.0906
100	32	14.7669	14.8747	**18.1745**	18.2177	17.8514	17.9372	17.9582	18.0504	17.0746	17.0995
200	16	14.7787	14.8644	18.1781	18.2618	17.8648	17.9187	17.9554	18.0967	**17.0705**	17.0845
200	32	**14.7457**	14.8309	18.1993	18.2349	**17.8508**	17.9195	17.9441	18.0474	17.0710	17.0830

Test loss for τ (%)

		INTC		JPM		MSFT		NVDA		TSLA	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	18.2433	18.3819	17.7608	17.8551	15.7303	16.0292	18.5784	18.5975	16.4104	16.4126
100	32	**18.2078**	18.3345	17.7453	17.8313	15.7152	15.9417	**18.5763**	18.5991	16.3982	16.4208
200	16	18.2432	18.3805	17.7571	17.8494	15.7305	16.0323	18.5855	18.5976	16.4101	16.4188
200	32	18.2205	18.3307	**17.7384**	17.8232	**15.6393**	15.9011	18.5826	18.6094	**16.3971**	16.4161

Test loss for τ' (%)

		AAPL		AMD		AMZN		CSCO		FB	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	7.3338	7.4145	7.5969	7.6466	7.3207	7.3701	**7.5728**	7.7182	7.1909	7.2218
100	32	7.2900	7.2926	**7.6126**	7.5962	7.3002	7.3846	7.5963	7.6333	7.1915	7.2359
200	16	7.3129	7.3920	7.5581	7.6435	7.3206	7.3397	7.5958	7.7151	**7.1891**	7.2162
200	32	**7.2527**	7.2675	7.6132	7.6331	**7.3047**	7.3450	7.5965	7.6370	7.1884	7.2139

Test loss for τ' (%)

		INTC		JPM		MSFT		NVDA		TSLA	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	7.7858	7.9298	7.8084	7.9285	7.3976	8.1143	7.7760	7.8061	6.7283	6.7072
100	32	**7.7622**	7.8554	7.7324	7.8571	7.1661	7.6948	**7.7789**	7.8079	6.6931	6.7243
200	16	7.8012	7.9238	7.7932	7.9210	7.4081	8.1061	7.7763	7.7986	6.7372	6.7147
200	32	7.7699	7.8475	**7.7587**	7.8383	**7.1439**	7.6177	7.7766	7.8232	**6.6945**	6.7245

Table 2: Test loss for the Classic LSTM-HTQF

Test loss for τ (%)

		AAPL		AMD		AMZN		CSCO		FB	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	14.7398	14.8339	18.0713	18.2065	17.6413	17.7867	17.7650	18.0145	17.0384	17.0622
100	32	**14.6970**	14.7867	**18.0599**	18.1455	17.6410	17.7511	**17.7319**	17.9590	17.0345	17.0504
200	16	14.7500	14.8353	18.1393	18.2231	17.6484	17.7871	17.7812	18.0071	**17.0339**	17.0498
200	32	14.6999	14.7814	18.1109	18.1838	**17.6403**	17.7555	17.7500	17.9574	17.0397	17.0497

Test loss for τ (%)

		INTC		JPM		MSFT		NVDA		TSLA	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	18.0848	18.3149	17.7651	17.8352	15.6805	16.0227	18.5354	18.5717	**16.3034**	16.3855
100	32	18.0368	18.2674	**17.7153**	17.8113	15.6979	15.8980	18.5287	18.5586	16.3079	16.3680
200	16	18.0683	18.3202	17.7489	17.8445	15.7537	16.0203	18.5242	18.5792	16.3349	16.4063
200	32	**18.0207**	18.2625	17.7463	17.8119	**15.6318**	15.8984	**18.5153**	18.5504	16.3205	16.3682

Test loss for τ' (%)

		AAPL		AMD		AMZN		CSCO		FB	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	7.3500	7.4181	7.5430	7.6282	7.1996	7.2984	7.5040	7.6897	7.1764	7.2106
100	32	**7.2388**	7.3046	**7.5263**	7.5653	7.1994	7.2674	**7.4805**	7.6101	7.1753	7.1954
200	16	7.3589	7.4090	7.5643	7.6338	7.2463	7.2861	7.5007	7.6872	**7.1713**	7.1929
200	32	7.2219	7.2843	7.6005	7.5988	**7.2218**	7.2615	7.4841	7.6121	7.1882	7.1880

Test loss for τ' (%)

		INTC		JPM		MSFT		NVDA		TSLA	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	7.7548	7.9366	7.8480	7.9180	7.1538	8.1865	7.7659	7.8147	**6.6653**	6.7227
100	32	7.7228	7.8420	**7.7208**	7.8490	7.2231	7.6257	7.7628	7.8007	6.6879	6.7229
200	16	7.7412	7.9299	7.7351	7.9215	7.7130	8.1810	7.7632	7.8184	6.7201	6.7702
200	32	**7.7101**	7.8385	7.7156	7.8382	**7.1215**	7.6837	**7.7528**	7.7907	6.6902	6.6916

Table 3: Test loss for the Volume LSTM-HTQF

Test loss for τ (%)

		AAPL		AMD		AMZN		CSCO		FB	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	14.8287	14.8948	18.2196	18.4834	17.8804	17.9402	17.9543	18.1081	**17.0864**	17.1410
100	32	14.7836	14.8559	**18.1907**	18.2611	17.8901	17.9308	**17.9313**	18.0853	17.0875	17.1154
200	16	14.8043	14.8953	18.2043	18.6325	**17.8752**	17.9875	17.9522	18.1027	17.1001	17.1098
200	32	**14.7412**	14.8601	18.2303	18.4432	17.8903	17.9489	17.9319	18.0695	17.1171	17.1111

Test loss for τ (%)

		INTC		JPM		MSFT		NVDA		TSLA	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	18.2521	18.3815	17.7783	17.8712	15.7677	16.0289	18.5939	18.6110	16.4024	16.4345
100	32	**18.2175**	18.3606	17.7709	17.8428	**15.6800**	15.9272	**18.5851**	18.6090	**16.3908**	16.4236
200	16	18.2495	18.3879	17.7666	17.8617	15.8132	16.0437	18.6017	18.6313	16.4152	16.4364
200	32	18.2196	18.3456	**17.7436**	17.8333	15.6945	15.9022	18.5873	18.6099	16.3993	16.4240

Test loss for τ' (%)

		AAPL		AMD		AMZN		CSCO		FB	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	7.3410	7.4113	7.6337	7.8840	7.3072	7.3474	7.5914	7.7201	**7.2071**	7.2654
100	32	7.2353	7.3505	**7.6367**	7.6508	7.3185	7.3437	**7.5798**	7.6825	7.2051	7.2686
200	16	7.2978	7.4166	7.6235	8.0270	**7.3228**	7.4186	7.6024	7.7204	7.2245	7.2452
200	32	**7.2451**	7.3084	7.6758	7.9312	7.3540	7.3447	7.5869	7.6791	7.2255	7.2448

Test loss for τ' (%)

		INTC		JPM		MSFT		NVDA		TSLA	
L	H	100	1/10	100	1/10	100	1/10	100	1/10	100	1/10
100	16	7.7995	7.9321	7.7707	7.9531	7.2740	8.0916	7.7898	7.8201	6.6872	6.7331
100	32	**7.7592**	7.8988	7.7793	7.8906	**7.1560**	7.4974	**7.7873**	7.8144	**6.7021**	6.7438
200	16	7.8202	7.9382	7.7513	7.9295	7.4728	8.0921	7.7769	7.8704	6.7360	6.7280
200	32	7.8076	7.8755	**7.7862**	7.8681	7.0446	7.4586	7.7800	7.8169	6.6934	6.7196

Table 4: Test loss for the Volatility LSTM-HTQF

the smaller batch size $bs = 100$ consistently finds a smaller loss function, suggesting that the size of the batch is indeed a relevant hyperparameter to consider.

For the Volume LSTM-HTQF, exactly half of the stocks (AAPL, AMD, CSCO, JPM, TSLA) achieve a better performance when $L = 100$, while the other half (AMZN, FB, INTC, MSFT, NVDA) do so when $L = 200$. Like for the previous model, all stocks except two (FB, TSLA) achieve a better performance when $h = 32$. Focusing on $\boldsymbol{\tau'}$, we can see that now for all stocks except three (AAPL, AMZN, JPM) the combination of hyperparameters that yields the best performance is the same as for $\boldsymbol{\tau}$. Again, for both sets, we find that the smaller batch size $bs = 100$ consistently finds a smaller loss function.

For the Volatility LSTM-HTQF, seven stocks (AMD, CSCO, FB, INTC, MSFT, NVDA, TSLA) achieve a better performance when $L = 100$, while only 3 (AAPL, AMZN, JPM) do so when $L = 200$. Like the previous models, all stocks except two (AMZN, FB) achieve a better performance when $h = 32$. Focusing on $\boldsymbol{\tau'}$, we can see that, only for two stocks (CSCO, INTC) the combination of hyperparameters that yields the best performance is the same as for $\boldsymbol{\tau}$. Again, for both sets, we find that the smaller batch size $bs = 100$ consistently finds a smaller loss function.

Tables 5 and 6 show the values of the loss function evaluated on $\boldsymbol{\tau}$ and $\boldsymbol{\tau'}$ respectively, for the nine benchmark models and the three types of machine learning models. More precisely,

since we re-estimated each benchmark model twice for $L \in \{100, 200\}$ as explained in section 5.3, the tables show the smallest between these two losses, along with the corresponding hyperparameters. Instead, for the tree types of machine learning models we report the result for the combination of hyperparameters that achieves the minimum loss on the complete τ set, i.e. the result marked in bold in tables 2, 3 and 4. For each stock, we mark in bold the model (either benchmark or machine learning) that yields the best performance and we highlight in gray the best benchmark model.

Test loss for τ (%)

Model Asset	AAPL				AMD				AMZN				CSCO				FB			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	15.7795	1	1		18.8361	1	1		18.5933	1	1		18.9552	1	1		17.2739	1	1	
GARCH-t	15.4795	1	1		18.7395	1	1		18.5535	1	1		18.6359	1	1		17.1712	1	1	
AR-GARCH-t	15.3599	1	1	1	18.3694	1	1	1	18.0146	1	1	1	18.2014	1	1	1	17.1038	1	1	1
EGARCH	15.4184	1	1		18.8364	1	1		18.8881	3	1		18.9366	1	1		17.4192	4	4	
EGARCH-t	15.6874	2	4		18.7880	3	3		18.7198	3	1		18.5968	1	1		17.4106	3	1	
AR-EGARCH-t	15.0469	1	1	1	18.3812	1	2	1	18.0157	1	1	1	18.1800	1	1	1	17.1111	1	1	1
GJR	15.7795	1	1		18.8361	1	1		18.5933	1	1		18.9552	1	1		17.2739	1	1	
GJR-t	15.4795	1	1		18.7395	1	1		18.5535	1	1		18.6359	1	1		17.1712	1	1	
AR-GJR-t	15.3599	1	1	1	18.3694	1	1	1	18.0146	1	1	1	18.2014	1	1	1	17.1038	1	1	1
Classic LSTM-HTQF	14.7457				18.1745				17.8508				17.9438				17.0705			
Volume LSTM-HTQF	**14.6970**				**18.0599**				**17.6403**				**17.7319**				**17.0339**			
Volatility LSTM-HTQF	14.7412				18.1907				17.8752				17.9313				17.0864			

Test loss for τ (%)

Model Asset	INTC				JPM				MSFT				NVDA				TSLA			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	19.1592	1	1		18.8684	1	1		17.0130	1	1		18.8571	1	1		16.8480	1	1	
GARCH-t	18.8279	1	1		18.2631	1	1		16.4800	1	1		18.7752	1	1		16.7669	1	1	
AR-GARCH-t	18.5043	1	1	1	18.1582	1	1	1	16.3252	1	1	1	18.6190	1	1	1	16.4540	1	1	1
EGARCH	19.0738	1	1		18.8415	1	2		16.2973	1	1		19.2300	3	1		16.8566	1	1	
EGARCH-t	18.7832	1	1		18.1629	1	1		16.2146	1	1		18.9896	3	1		17.0068	3	1	
AR-EGARCH-t	18.5016	1	3	1	18.0771	1	1	1	16.0885	1	1	1	18.8374	3	1	1	16.4589	1	1	1
GJR	19.1592	1	1		18.8684	1	1		17.0130	1	1		18.8571	1	1		16.8480	1	1	
GJR-t	18.8279	1	1		18.2631	1	1		16.4800	1	1		18.7752	1	1		16.7669	1	1	
AR-GJR-t	18.5043	1	1	1	18.1582	1	1	1	16.3252	1	1	1	18.6190	1	1	1	16.4540	1	1	1
Classic LSTM-HTQF	18.2078				17.7384				15.6393				18.5763				16.3971			
Volume LSTM-HTQF	**18.0207**				**17.7153**				**15.6318**				**18.5153**				**16.3034**			
Volatility LSTM-HTQF	18.2175				17.7436				15.6800				18.5851				16.3908			

Table 5: Test loss for the benchmark models and the machine learning models on τ

Starting with the benchmarks, the tables show that the simplest models GARCH, GARCH-t, EGARCH, EGARCH-t, GJR and GJR-t, never achieve the best performance, except once for the EGARCH model (MSFT on τ'). Instead, the benchmarks that model the conditional mean in an autoregressive way show a significantly better ability to capture the dynamics of the conditional distribution: the AR-GARCH-t model achieves the best performance six times (AMZN, NVDA, TSLA on τ and AMD, AMZN, INTC on τ'), the AR-EGARCH-t model seven times (AAPL, CSCO, INTC, JPM, MSFT on τ and AAPL, JPM on τ') and the AR-GJR-t six times (AMD, FB on τ and CSCO, FB, NVDA, TSLA on τ').

Moving on to the machine learning models, our most important result is the superior performance of the Volume LSTM-HTQF, that consistently outperforms the other two machine learning models, as well as each of the nine benchmark models across τ and τ'. This does

Test loss for τ' (%)

Model Asset	AAPL				AMD				AMZN				CSCO				FB			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	7.4941	1	1		7.8857	1	1		7.5461	1	1		8.3327	1	1		7.2865	1	1	
GARCH-t	7.4105	1	1		7.8110	1	1		7.5211	1	1		8.0935	1	1		7.2395	1	1	
AR-GARCH-t	7.3739	1	1	1	7.7107	1	1	1	7.3832	1	1	1	7.8508	1	1	1	7.2322	1	1	1
EGARCH	7.3648	1	1		7.9018	1	1		7.9091	3	1		8.3393	1	1		7.4507	4	4	
EGARCH-t	7.7287	2	4		7.8996	3	3		7.8041	3	1		8.0835	1	1		7.5865	3	1	
AR-EGARCH-t	7.3628	1	1	1	7.7363	1	2	1	7.3902	1	1	1	7.8554	1	1	1	7.2515	1	1	1
GJR	7.4941	1	1		7.8857	1	1		7.5461	1	1		8.3327	1	1		7.2865	1	1	
GJR-t	7.4105	1	1		7.8110	1	1		7.5211	1	1		8.0935	1	1		7.2395	1	1	
AR-GJR-t	7.3739	1	1	1	7.7107	1	1	1	7.3832	1	1	1	7.8508	1	1	1	7.2322	1	1	1
Classic LSTM-HTQF	**7.2527**				7.6126				7.3047				7.5728				7.1891			
Volume LSTM-HTQF	**7.2388**				**7.5263**				**7.2218**				**7.4805**				**7.1713**			
Volatility LSTM-HTQF	**7.2451**				7.6367				7.3228				7.5798				7.2071			

Test loss for τ' (%)

Model Asset	INTC				JPM				MSFT				NVDA				TSLA			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	8.4305	1	1		8.6585	1	1		7.9102	1	1		7.8993	1	1		6.8836	1	1	
GARCH-t	8.2171	1	1		8.2300	1	1		7.6291	1	1		7.8522	1	1		6.8274	1	1	
AR-GARCH-t	8.0472	1	1	1	8.1567	1	1	1	7.5513	1	1	1	7.8076	1	1	1	6.7233	1	1	1
EGARCH	8.4048	1	1		8.6635	1	2		7.5147	1	1		8.2986	3	1		6.9026	1	1	
EGARCH-t	8.2229	1	1		8.1347	1	1		7.9489	1	1		8.1650	3	1		7.2174	3	1	
AR-EGARCH-t	8.1271	1	3	1	8.0726	1	1	1	7.8499	1	1	1	8.1216	3	1	1	6.7347	1	1	1
GJR	8.4305	1	1		8.6585	1	1		7.9102	1	1		7.8993	1	1		6.8836	1	1	
GJR-t	8.2171	1	1		8.2300	1	1		7.6290	1	1		7.8522	1	1		6.8274	1	1	
AR-GJR-t	8.0472	1	1	1	8.1567	1	1	1	7.5513	1	1	1	7.8076	1	1	1	6.7233	1	1	1
Classic LSTM-HTQF	7.7622				7.7587				7.1439				7.7789				6.6945			
Volume LSTM-HTQF	**7.7101**				**7.7208**				**7.1215**				**7.7528**				**6.6653**			
Volatility LSTM-HTQF	7.7592				7.7862				7.1560				7.7873				6.7021			

Table 6: Test loss for the benchmark models and the machine learning models on τ'

confirm the idea that the feature matrix used in the construction of the machine learning model should be expanded to contain any information related to the conditional distribution of the returns. Moreover, we find that the Volatility LSTM-HTQF outperforms the Classic LSTM-HTQF only for three stocks on the τ set (AAPL, CSCO, TSLA) and only for two stocks on the τ' set (AAPL, INTC). This results is not totally unexpected. As we explained in section 3.4.3, the idea behind this variation was to create a hybrid model between the Volume LSTM-HTQF and the Classic LSTM-HTQF, that adds an additional feature, but still uses only information that can be derived from the time series prices. Therefore, this inferior performance is likely due to the length L' set to 20 trading days on which the realized volatility is calculated. Given the high frequency of the data (at 5 second intervals), it may be possible that such a long time span creates and excessive aggregation, or that the machine learning model needs more data to learn the dynamics of the conditional distribution. Finally, although they underperform the more powerful Volume LSTM-HTQF, both the Classic LSTM-HTQF and the Volatility LSTM-HTQF still consistently outperform all nine benchmark models.

For comparison purposes, we also present the results that we obtain when we apply our Classic LSTM-HTQF to the daily data of five stock market indexes used in the original study by Yan et al. (2018). Table 7 shows the values of the loss function evaluated on τ and τ' for all possible combinations of the original hyperparameters $L \in \{100, 200\}$ and $h \in \{16, 32\}$. To

be consistent with Yan et al. (2018), we consider a batch size of one fifth of the size of the validation set, but we also test a smaller value of this parameter, namely one tenth of the size of the validation set. For each stock, we mark in bold and highlight in gray the combination of hyperparameters that yields the best performance on τ and τ' respectively.

Focusing on the τ set, we can see that four indexes (SPX500, NASDAQ100, NIKKEI225, DAX) achieve a better performance when considering a longer series of past returns $L = 200$, while only one index (HSI) does so with $L = 100$. Moreover, all indexes except one (NIKKEI225) achieve a better result with a larger number of units $h = 32$ in the hidden state of the LSTM. Focusing on τ', we can see that for three indexes (NIKKEI225, DAX, HSI) the combination of hyperparameters that yields the best performance is the same as for τ. Finally, the smaller batch size ($bs = 1/10$ of the size of the validation set) finds a smaller loss function for all indexes except one (DAX) in the τ set and two (NASDAQ100, DAX) in the τ' set, confirming what we found in the case of high frequency data, namely that the size of the batch is indeed a relevant hyperparameter to consider and that a smaller batch size can lead to performance improvements.

Test loss for τ (%)

		SPX500		NASDAQ100		NIKKEI225		DAX		HSI	
L	H	1/5	1/10	1/5	1/10	1/5	1/10	1/5	1/10	1/5	1/10
100	16	23.0720	23.0877	13.9132	13.8312	28.1159	28.1095	19.4313	19.4393	15.9652	15.9478
100	32	23.0884	23.0503	13.8687	13.8508	28.1023	28.1362	19.4661	19.4477	15.9369	**15.9222**
200	16	23.0148	23.0257	13.7944	13.8105	28.0497	**28.0164**	19.1871	19.1407	16.7224	16.7222
200	32	22.9632	**22.9449**	13.8281	**13.7924**	28.0545	28.0348	**19.1347**	19.2147	16.7133	16.7216

Test loss for τ' (%)

		SPX500		NASDAQ100		NIKKEI225		DAX		HSI	
L	H	1/5	1/10	1/5	1/10	1/5	1/10	1/5	1/10	1/5	1/10
100	16	10.2308	10.2475	6.5766	6.5058	12.6282	12.5937	8.2760	8.2824	7.0199	6.9839
100	32	10.3530	10.2717	6.5473	6.5135	12.7061	12.6524	8.3011	8.2821	6.9937	**6.9742**
200	16	10.2073	10.1917	6.4814	6.4813	12.6003	**12.5498**	8.2446	8.2166	7.3609	7.3487
200	32	10.2877	**10.2154**	6.4789	**6.4809**	12.6553	12.6567	**8.2109**	8.3344	7.3441	7.3506

Table 7: Test loss for the Classic LSTM-HTQF using daily data

Tables 8 and 9 show the values of the loss function evaluated on τ and τ' respectively, for the nine benchmark models and the Classic LSTM-HTQF, and provide the original results obtained in Yan et al. (2018) for comparative purposes. For the machine learning model we report the result for the combination of hyperparameters that achieves the minimum loss on the complete τ set, i.e. the results marked in bold in table 7. For each index we mark in bold the model (either the benchmark or machine learning) that yields the best performance and we highlight in gray the best benchmark model.

Focusing on the machine learning models, we can see that our implementation of the Classic LSTM-HTQF obtains very similar results to those found in the original study and, in all cases except two (NASDAQ100 and DAX on τ'), our model implementation displays a slight performance improvement over Yan et al. (2018)'s model. Moreover, unlike the authors' orig-

inal model, our model implementation always outperforms the nine benchmark econometric models, both on τ and τ'. Finally, moving on to the benchmark models, we can see that our implementation obtains very similar results. In particular, for three indexes out of five (SPX500, NASDAQ100, NIKKEI225) our best performing benchmarks are the same as those found by Yan et al. (2018), both on τ and τ'.

Test loss for τ (%)

Model Asset	SPX500				NASDAQ100				NIKKEI225				DAX				HSI			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	23.1618	1	2		14.0356	1	1		28.6740	1	2		19.5416	1	3		16.9307	1	1	
GARCH-t	23.0672	1	2		13.9456	1	1		28.5150	1	2		19.3765	1	3		16.8440	1	3	
AR-GARCH-t	23.0283	1	2	2	13.9164	1	1	2	28.4813	1	2	2	19.3940	1	3	2	16.8497	1	3	1
EGARCH	23.1369	1	2		13.9863	1	1		28.4985	1	1		19.3544	1	1		16.9457	1	2	
EGARCH-t	23.0126	1	1		13.9270	1	1		28.4158	1	1		19.2546	1	1		16.8554	1	3	
AR-EGARCH-t	22.9711	1	1	2	13.8989	1	1	2	28.3801	1	1	2	19.2740	1	1	2	16.8598	1	3	1
GJR	23.1618	1	2		14.0356	1	1		28.6740	1	2		19.5416	1	3		16.9307	1	1	
GJR-t	23.0672	1	2		13.9456	1	1		28.5150	1	2		19.3765	1	3		16.8440	1	3	
AR-GJR-t	23.0283	1	2	2	13.9164	1	1	2	28.4813	1	2	2	19.3940	1	3	2	16.8497	1	3	1
Classic LSTM-HTQF	**22.9449**				**13.7924**				**28.0164**				**19.1347**				**15.9222**			

Test loss for τ (%) - Yan et al (2018)

Model Asset	SPX500				NASDAQ100				NIKKEI225				DAX				HSI			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	23.1600	-	-		14.0600	-	-		28.6800	-	-		19.6800	-	-		16.2300	-	-	
GARCH-t	23.1400	-	-		13.9600	-	-		28.5500	-	-		19.6100	-	-		16.1200	-	-	
EGARCH-t	23.0800	-	-		13.9500	-	-		28.5100	-	-		19.5700	-	-		16.1100	-	-	
AR-EGARCH-t	23.0400	-	-	-	13.9100	-	-	-	**28.4700**	-	-	-	19.5200	-	-	-	16.1100	-	-	-
GJR-t	23.1400	-	-		13.9600	-	-		28.5500	-	-		19.6100	-	-		16.1200	-	-	
AR-GJR-t	23.1000	-	-	-	13.9300	-	-	-	28.5200	-	-	-	19.6300	-	-	-	16.1200	-	-	-
Classic LSTM-HTQF	**22.9900**				**13.8700**				28.5400				**19.3200**				**15.9800**			

Table 8: Test loss for the benchmark models and the Classic LSTM-HTQF on τ using daily data

Test loss for τ' (%)

Model Asset	SPX500				NASDAQ100				NIKKEI225				DAX				HSI			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	10.5108	1	2		6.6716	1	1		13.2642	1	2		8.6229	1	3		7.5934	1	1	
GARCH-t	10.4665	1	2		6.6599	1	1		13.2285	1	2		8.4568	1	3		7.4997	1	3	
AR-GARCH-t	10.5048	1	2	2	6.6513	1	1	2	13.2704	1	2	2	8.4992	1	3	2	7.4859	1	3	1
EGARCH	10.4694	1	2		6.6658	1	1		13.2376	1	1		8.2551	1	1		7.5675	1	2	
EGARCH-t	10.3466	1	1		6.6665	1	1		13.1692	1	1		8.2517	1	1		7.4958	1	3	
AR-EGARCH-t	10.3806	1	1	2	6.6778	1	1	2	13.2049	1	1	2	8.2873	1	1	2	7.4870	1	3	1
GJR	10.5108	1	2		6.6716	1	1		13.2642	1	2		8.6229	1	3		7.5934	1	1	
GJR-t	10.4664	1	2		6.6599	1	1		13.2285	1	2		8.4568	1	3		7.4997	1	3	
AR-GJR-t	10.5048	1	2	2	6.6513	1	1	2	13.2704	1	2	2	8.4992	1	3	2	7.4859	1	3	1
Classic LSTM-HTQF	**10.2154**				**6.4809**				**12.5498**				**8.2109**				**6.9742**			

Test loss for τ' (%) - Yan et al (2018)

Model Asset	SPX500				NASDAQ100				NIKKEI225				DAX				HSI			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	10.3900	-	-		6.6900	-	-		13.3900	-	-		8.5300	-	-		7.2900	-	-	
GARCH-t	10.4800	-	-		6.6700	-	-		13.3000	-	-		8.5000	-	-		7.1900	-	-	
EGARCH-t	10.3700	-	-		6.6800	-	-		13.2400	-	-		8.4000	-	-		7.1700	-	-	
AR-EGARCH-t	10.4100	-	-	-	6.6600	-	-	-	**13.2700**	-	-	-	8.3400	-	-	-	7.1500	-	-	-
GJR-t	10.4800	-	-		6.6700	-	-		13.3000	-	-		8.5000	-	-		7.1900	-	-	
AR-GJR-t	10.5200	-	-	-	6.6600	-	-	-	13.3300	-	-	-	8.5400	-	-	-	7.1700	-	-	-
Classic LSTM-HTQF	**10.2500**				**6.4600**				**12.8900**				**8.1000**				**7.0200**			

Table 9: Test loss for the benchmark models and the Classic LSTM-HTQF on τ' using daily data

6 Conclusion

The returns of financial asset are stochastic and their distributions have leptokurtic, asymmetric and time-varying tails, both conditionally and unconditionally. Although it is impossible to accurately predict future returns, it is possible to predict with precision the characteristics of their conditional distributions $p(r_t|r_{t-1}, r_{t-2}, \dots)$, a task which is essential in the context of asset pricing and risk management.

Quantiles are one way to describe the shape of a distribution, as modeling the conditional quantiles for a finite set of probabilities is almost equivalent to modeling the entire conditional distribution. Therefore, the straightforward approach to this task in discrete-time Econometrics is to use the Generalized Autoregressive Conditional Heteroskedasticity model and its variations to predict the conditional quantiles of a distribution. The problem with this type of models is that they make strong assumptions about the probability distribution of the conditional distribution, which may introduce tractability and ill-posedness issues. Another approach to predict the characteristics of the conditional distribution is quantile regression, which forecasts the quantiles of a conditional by means of a parametric function, rather than making distributional assumptions. However, the traditional quantile regression has three major shortcomings, namely quantile crossing, the increasing number of parameters required to estimate more quantiles and the lack of interpretability. To overcome the need for a distributional assumption, as well as the issues of traditional quantile regression, Yan et al. (2018) introduced a novel parametric heavy-tail quantile function to represent a distribution with leptokurtic, asymmetric and time-varying tails.

Leveraging this model, in this paper, we forecast the conditional quantiles and heavy tails of series of stock returns measured at high frequency, in a parsimonious quantile regression framework, that describes the conditional distribution using Yan et al. (2018)'s parametric heavy tail quantile function. More precisely, we implement three variations of the same machine learning model to forecast the parameters of this quantile function. The Classic LSTM-HTQF, which replicates the author's original model, is a densely connected network, stacked on top of a recurrent neural network with a long-short term memory unit, and takes as input the sequence of the first four central moments of a series of past returns. The Volume LSTM-HTQF adds a fifth feature, namely the trading volume associated with each observation in the sequence of past returns. The Volatility LSTM-HTQF is a hybrid model between the previous two that adds a fifth feature, namely the realized volatility of the returns over the past four weeks, but still uses only information that can be derived from the time series of prices. Finally, we compare the performance of the three machine learning models against nine econometric benchmarks,

namely the GARCH, EGARCH, GJR and their variations.

Our study is run on tick-by-tick traded prices and volumes for ten highly liquid stocks listed on the Nasdaq Stock Market and the New York Stock Exchange. For each security, we consider a fixed time range from 9.35 a.m. to 15.55 p.m. for each trading day over a time span of 20 weeks for the Classic LSTM-HTQF and the Volume LSTM-HTQF and a time span of 24 weeks for the Volatility LSTM-HTQF. After implementing the data cleaning and data manipulation processes, we obtain time series at 5 second intervals for each of the ten stocks in our sample.

Starting with the econometric models used as benchmarks, we find that the benchmarks that model the conditional mean in an autoregressive way (AR-GARCH-t, AR-EGARCH-t, AR-GJR-t) show a significantly better ability to capture the dynamics of the conditional distribution of returns that the simpler counterparts. Moving on to the machine learning models, our most important result is the superior performance of the Volume LSTM-HTQF, that consistently outperforms the other two machine learning models, as well as each of the nine benchmark models across the complete τ set of probability levels and the smaller τ' set, the quantiles of which are the value at risk levels representing the downside risk of the distribution. This result proves that the model performance can be improved by expanding the feature matrix to include all information that can aid in the prediction of the conditional distribution of the returns. Moreover, although we do not find evidence of a consistent under- or over-performance of the Volatility LSTM-HTQF with respect to the Classic LSTM-HTQF, the frequent underperformance of the former is likely due to the excessive aggregation created by the length of the time span on which the realized volatility is calculated. It is worth noting that, notwithstanding the underperformance of the Classic LSTM-HTQF and the Volatility LSTM-HTQF against the Volume LSTM-HTQF, they both still consistently outperform all nine benchmark models. Lastly, we find that the size of the batch has a strong impact on the results and, in particular, that a smaller batch size improves the performance of all machine learning models.

Overall, we conclude that the Yan et al. (2018)'s parametric heavy tail quantile function is more suited to model the leptokurtic and asymmetric behavior of the tails of a conditional distribution of high frequency stock returns than the econometric models, which must assume a particular specification of the probability density function, such as the t-distribution. Moreover, this parametric function has time dependent parameters, which allow to account for the time varying behavior of the tails; on the other hand, in order to do so, the GARCH based models would need to specify the density function in a complicated analytical form, which would make the model estimation unfeasible if we assume time varying degrees of freedom. Lastly, while the non-linear activation functions used in the LSTM allow to learn the non-linear dependencies

on the past data, the linear specification of the GARCH based models does not allow to do so.

The main limitation of this study is related to the length of the period used to compute the realized volatility in the Volatility LSTM-HTQF. Since we tried only one specification (four weeks, or twenty trading days), we strongly believe that the mediocre performance of this model is due to the excessive level of aggregation. Future implementations should test shorter lengths of this time span, or feed significantly more data to the machine learning model.

In our view the most interesting direction for future studies on this topic is the implementation of machine learning models with a Transformer architecture. Recently presented in Vaswani et al. (2017), this new specification has been shown to outperform the simple recurrent neural network in several natural language processing tasks. The architecture is based on the concept of self-attention, which means that the model is able to learn to focus on the parts of the input sequence that are most relevant to the prediction of the output. Finally, other interesting developments include comparing the performance of our machine learning models against that of the traditional quantile regression coupled with a LSTM unit, as well as expanding the feature matrix to include other information, such as the return of correlated assets or the fundamentals of the stock.

Appendices

A Weight Updates in the Adaline Model

In the Adaline model, the weight update via gradient descent is defined as follows:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w} \quad \text{where} \quad \Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w}) \tag{99}$$

The following is the derivation of the gradient of the cost function $\nabla J(\boldsymbol{w})$:

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j} \frac{1}{2} \sum_i \Big(y_i - \phi(z_i) \Big)^2 \\
&= \frac{1}{2} \frac{\partial}{\partial w_j} \sum_i \Big(y_i - \phi(z_i) \Big)^2 \\
&= \frac{1}{2} \sum_i 2 \Big(y_i - \phi(z_i) \Big) \frac{\partial}{\partial w_j} \Big(y_i - \phi(z_i) \Big) \\
&= \sum_i \Big(y_i - \phi(z_i) \Big) \frac{\partial}{\partial w_j} \Big(y_i - \sum_j (w_j x_i^j) \Big) \\
&= \sum_i \Big(y_i - \phi(z_i) \Big) (-x_i^j) \\
&= -\sum_i \Big(y_i - \phi(z_i) \Big) x_i^j
\end{aligned} \tag{100}$$

So that the weight update becomes:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \Big(y_i - \phi(z_i) \Big) x_i^j \tag{101}$$

B Logistic Sigmoid Function

Let's start by introducing the odds ratio, which quantifies the odds in favor of a particular event, and which is defined as follows:

$$f(P) = \frac{P}{1 - P} \tag{102}$$

where P is the probability of the event taking place (e.g. the probability of a class label being $y = 1$ in the context of binary classification).

Then, let the logit function be:

$$logit(P) = log\Big(f(P) \Big) = log\left(\frac{P}{1 - P} \right) \tag{103}$$

which takes a value $P \in [0, 1)$ and maps it into the continuous range of real numbers.

Let's now write the linear relationship between the logit function and the vector of features for sample i:

$$logit(P) = logit\Big(p(y=1|\boldsymbol{x}_i)\Big) = x_i^0 w_0 + \boldsymbol{x}_i \boldsymbol{w} = x_i^0 w_0 + x_i^1 w_1 + \cdots + x_i^m w_m = z_i \tag{104}$$

where $p(y=1|\boldsymbol{x}_i)$ is the conditional probability that sample i belongs to the class $y=1$, given its vector of features $\boldsymbol{x}_i$.

However, what we are interested in is the probability that sample i belongs to the class $y=1$, given its features. This can be calculated by inverting the logit function:

$$p(y=1|\boldsymbol{x}_i) = logit^{-1}(z_i) \tag{105}$$

which can be explicited as follows:

$$\begin{aligned} log\left(\frac{P}{1-P}\right) &= Z \\ \frac{P}{1-P} &= e^Z \\ P &= e^Z(1-P) \\ P &= e^Z - e^Z P \\ P(1+e^Z) &= e^Z \\ P &= \frac{e^Z}{1+e^Z} \\ P &= \frac{e^Z e^{-Z}}{(1+e^Z)e^{-Z}} \\ P &= \frac{1}{e^{-Z}+1} \end{aligned} \tag{106}$$

which is formalized into the logistic sigmoid function:

$$\phi(Z) = \frac{1}{1+e^{-Z}} \tag{107}$$

where Z is the net input calculated as the linear combination of sample features and weights. The function takes a value $Z \in \mathbb{R}$ and maps it into a continuous range between 0 and 1: $\phi(Z)$ tends to 0 and 1 when Z tends to minus and plus infinity respectively.

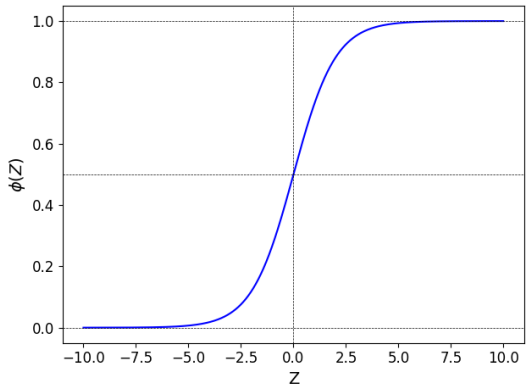

Figure 16: Logistic sigmoid function

C Cost Function in the Linear Regression Model

Let the probability distribution function be $f(y_i|\boldsymbol{x}_i;\boldsymbol{w})$ and assume that the samples are mutually independent. Then the joint probability distribution function is given by:

$$f(\boldsymbol{y}|\boldsymbol{X};\boldsymbol{w}) = \prod_{i=1}^{n} f(y_i|\boldsymbol{x}_i;\boldsymbol{w}) \tag{108}$$

The likelihood function of the samples is then given by:

$$\begin{aligned} L(\boldsymbol{w}|\boldsymbol{y},\boldsymbol{X}) &= \prod_{i=1}^{n} L_i(\boldsymbol{w}|y_i,\boldsymbol{x}_i) \\ &= \prod_{i=1}^{n} f(y_i|\boldsymbol{x}_i;\boldsymbol{w}) \\ &= \prod_{i=1}^{n} \Big(\phi(z_i)\Big)^{y_i} \Big(1-\phi(z_i)\Big)^{1-y_i} \end{aligned} \tag{109}$$

Let's define the log-likelihood function by applying a monotonous transformation, which simplifies the calculation of the first order condition:

$$logL(\boldsymbol{w}|\boldsymbol{y},\boldsymbol{X}) = \sum_{i=1}^{n} \Big[y_i \, log\Big(\phi(z_i)\Big) + (1-y_i) \, log\Big(1-\phi(z_i)\Big)\Big] \tag{110}$$

Let's now turn the log-likelihood function into a cost function, so that it can be minimized via gradient descent:

$$J(\boldsymbol{w}) = \sum_{i=1}^{n} \Big[-y_i \, log\Big(\phi(z_i)\Big) - (1-y_i) \, log\Big(1-\phi(z_i)\Big)\Big] \tag{111}$$

To clarify the meaning of this cost function, let's evaluate the cost of a single training sample:

$$J(\boldsymbol{w}) = -y_i \; log\Big(\phi(z_i)\Big) - (1 - y_i) \; log\Big(1 - \phi(z_i)\Big) \tag{112}$$
$$= \begin{cases} - \; log\Big(\phi(z_i)\Big) & \text{if } y_i = 1 \\ - \; log\Big(1 - \phi(z_i)\Big) & \text{if } y_i = 0 \end{cases}$$

As shows in the figure below, which plots the logistic sigmoid activation function on the x-axis and the associated cost function on the y-axis, the cost approaches zero only if the sample is correctly predicted to belong to class 1 (blue line) or class 0 (red line). Instead, mis-classifications are penalized with an increasing cost.

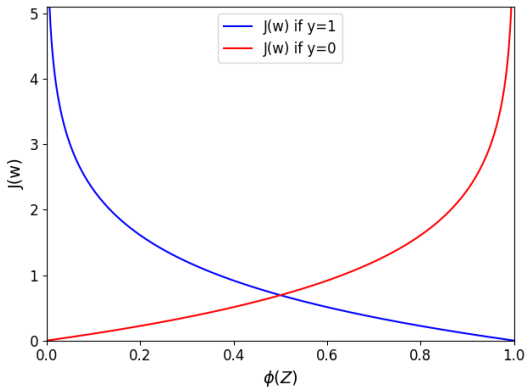

Figure 17: Logistic cost function

D Gradient of the Cost Function in the Linear Regression Model

In the Linear Regression model, the weight update via gradient descent is defined as follows:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w} \quad \text{where} \quad \Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w}) \tag{113}$$

The following is the derivation of the gradient of the cost function $\nabla J(\boldsymbol{w})$:

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \sum_i \Big[- y_i \; log\Big(\phi(z_i)\Big) - (1 - y_i) \; log\Big(1 - \phi(z_i)\Big)\Big] \tag{114}$$
$$= -\sum_i \left\{ \left[y_i \; \frac{\partial log\Big(\phi(z_i)\Big)}{\partial \phi(z_i)} + (1 - y_i) \; \frac{\partial log\Big(1 - \phi(z_i)\Big)}{\partial \phi(z_i)} \right] \frac{\partial \phi(z_i)}{\partial z_i} \frac{\partial z_i}{\partial w_i} \right\}$$
$$= -\sum_i \left\{ \left[\frac{y_i}{\phi(z_i)} - \frac{1 - y_i}{1 - \phi(z_i)} \right] \frac{e^{-z_i}}{(1 + e^{-z_i})^2} x_i^j \right\}$$
$$= -\sum_i \left\{ \left[\frac{y_i}{\phi(z_i)} - \frac{1 - y_i}{1 - \phi(z_i)} \right] \frac{1}{1 + e^{-z_i}} \left(1 - \frac{1}{1 + e^{-z_i}} \right) x_i^j \right\}$$

$$
\begin{aligned}
&= -\sum_i \left\{ \left[\frac{y_i}{\phi(z_i)} - \frac{1-y_i}{1-\phi(z_i)} \right] \phi(z_i)\Big(1-\phi(z_i)\Big) x_i^j \right\} \\
&= -\sum_i \left\{ \Big[y_i - y_i\phi(z_i) - \phi(z_i) + y_i\phi(z_i) \Big] x_i^j \right\} \\
&= -\sum_i \left\{ \Big[y_i - \phi(z_i) \Big] x_i^j \right\}
\end{aligned}
$$

So that the weight update becomes:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \Big(y_i - \phi(z_i) \Big) x_i^j \tag{115}$$

E L2-Regularization to Prevent Overfitting

A model suffers from overfitting (or high variance) when it learns the detail of the training data to the extent that it negatively impacts its performance on unseen (test) data. Similarly, a model can suffer from underfitting (or high bias) when it does not learn the underlying pattern in the training data and this results in the same poor performance on unseen (test) data. More precisely, while variance measures the variability of the model prediction for a particular sample instance when the model is trained on different samples of training data, bias measures how far off this model prediction is from the true value. The following Figure illustrates the concepts of underfitting and overfitting.

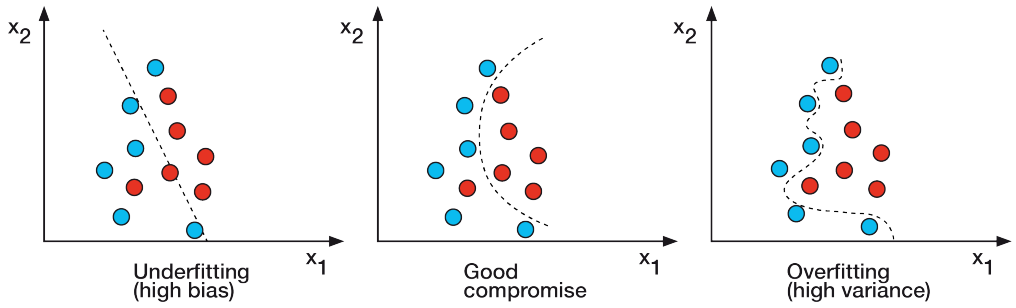

Figure 18: Underfitting and overfitting

Regularization is a technique that allows to find a good tradeoff between overfitting and underfitting, by controlling for collinearity and noise in the training data, and avoid extreme parameter values. The most common type of regularization is L2-regularization:

$$\frac{\lambda}{2}||\boldsymbol{w}||^2 = \frac{\lambda}{2}\sum_j w_j^2 \tag{116}$$

where λ is the regularization parameter, which allows to control how well the model fits the

training data while avoiding extreme weight values. The higher the value of λ, the stronger the regularization. For regularization to work successfully, it is necessary to apply a standardization to the data, so that all features are on a comparable scale.

F Backpropagation Algorithm

Rumelhart, Hinton, Williams, et al. (1988) popularized the backpropagation algorithm, which is a very efficient way to compute the partial derivatives of a cost function in a Multilayer Neural Network to learn the optimal weights. Differently from the cost function of single layer neural networks, such as the Adaline or the Logistic Regression, the error surface of the cost function of a Multilayer Neural Network is neither convex nor smooth with respect to the parameters. Therefore, the challenge with the parametrization of such a network is that the high number of weight coefficients creates a high-dimensional cost surface with many local minima, that need to be overcome in order to find the global minimum.

The idea behind the backpropagation algorithm starts with the chain rule used in calculus to compute the derivatives of nested functions, such as $F(X) = f(g(h(u(v(X)))))$. Then its partial derivative with respect to X is:

$$\begin{aligned} \frac{\partial}{\partial X} F(X) &= \frac{\partial}{\partial X} f(g(h(u(v(X))))) \\ &= \frac{\partial f}{\partial g} \frac{\partial g}{\partial h} \frac{\partial h}{\partial u} \frac{\partial u}{\partial v} \frac{\partial v}{\partial X} \end{aligned} \tag{117}$$

Backpropagation is a special case of automatic differentiation, which is a set of techniques used in computing to efficiently solve the problem of calculating derivatives of nested functions. The idea is that applying the chain rule in a forward manner implies multiplying large matrices for each layer and eventually multiplying them by a vector to find the output. On the other hand, applying the chain rule in a backwards manner implies multiplying a vector by a matrix, which returns a vector, which is then multiplied by the next matrix, and so on. The multiplication of a matrix by a vector is computationally less expensive and it is what makes the backpropagation algorithm so powerful.

G Derivative of the Error Term of the Hidden Layer of a Multilayer Perceptron

The error term of the hidden layer is calculated as follows:

$$\boldsymbol{\Delta}_{(h)} = \boldsymbol{\Delta}_{(out)} (\boldsymbol{W}_{(out)})^\top \odot \frac{\partial \phi(\boldsymbol{Z}_{(h)})}{\partial \boldsymbol{Z}_{(h)}} \tag{118}$$

where $\odot$ indicates an element-wise multiplication and where the derivative of the logistic sigmoid activation function is calculated as follows:

$$\begin{aligned}
\frac{\partial \phi(\boldsymbol{Z}_{(h)})}{\partial \boldsymbol{Z}_{(h)}} &= \frac{\partial}{\partial \boldsymbol{Z}_{(h)}} \left(\frac{1}{1 + e^{-\boldsymbol{Z}_{(h)}}} \right) \\
&= \frac{e^{-\boldsymbol{Z}_{(h)}}}{(1 + e^{-\boldsymbol{Z}_{(h)}})^2} \\
&= \frac{1 + e^{-\boldsymbol{Z}_{(h)}}}{\left(1 + e^{-\boldsymbol{Z}_{(h)}}\right)^2} - \left(\frac{1}{1 + e^{-\boldsymbol{Z}_{(h)}}} \right)^2 \\
&= \frac{1}{1 + e^{-\boldsymbol{Z}_{(h)}}} - \left(\frac{1}{1 + e^{-\boldsymbol{Z}_{(h)}}} \right)^2 \\
&= \phi(\boldsymbol{Z}_{(h)}) - \left(\phi(\boldsymbol{Z}_{(h)})\right)^2 \\
&= \phi(\boldsymbol{Z}_{(h)}) \big(1 - \phi(\boldsymbol{Z}_{(h)})\big) \\
&= \boldsymbol{A}_{(h)} \big(1 - \boldsymbol{A}_{(h)}\big)
\end{aligned} \tag{119}$$

Therefore, we can rewrite the error term of the hidden layer as follows:

$$\boldsymbol{\Delta}_{(h)} = \boldsymbol{\Delta}_{(out)} (\boldsymbol{W}_{(out)})^\top \odot \Big(\boldsymbol{A}_{(h)} \odot \big(1 - \boldsymbol{A}_{(h)}\big) \Big) \tag{120}$$

H Variants of the LSTM

Gers and Schmidhuber (2000) introduced a variant of the traditional LSTM that adds peephole connections, which allow the gates to receive the cell state as an additional input (Figure 19a).

$$\boldsymbol{f}_t = \sigma\big(\boldsymbol{a}_{(in),t} \boldsymbol{W}_{xf} + \boldsymbol{a}_{(h),t-1} \boldsymbol{W}_{hf} + \boldsymbol{c}_{t-1} \boldsymbol{W}_{cf}\big) \tag{121}$$

$$\boldsymbol{i}_t = \sigma\big(\boldsymbol{a}_{(in),t} \boldsymbol{W}_{xi} + \boldsymbol{a}_{(h),t-1} \boldsymbol{W}_{hi} + \boldsymbol{c}_{t-1} \boldsymbol{W}_{ci}\big) \tag{122}$$

$$\boldsymbol{o}_t = \sigma\big(\boldsymbol{a}_{(in),t} \boldsymbol{W}_{xo} + \boldsymbol{a}_{(h),t-1} \boldsymbol{W}_{ho} + \boldsymbol{c}_t \boldsymbol{W}_{co}\big) \tag{123}$$

Another variant uses coupled forget and input gates: instead of separately deciding what to forget and what to add new information to, the LSTM only forgets where new information is added (Figure 19b).

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + (1 - \boldsymbol{f}_t) \odot \tilde{\boldsymbol{c}}_t \tag{124}$$

Cho et al. (2014) introduced the Gated Recurrent Unit, which combines the forget and input gates into an update gate, merges the cell state and the hidden state, and makes some other

minor changes (Figure 19c).

$$\boldsymbol{z}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xz} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hz}\big) \tag{125}$$

$$\boldsymbol{r}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xr} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hr}\big) \tag{126}$$

$$\tilde{\boldsymbol{a}}_{(h),t} = tanh\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xh} + (\boldsymbol{r}_t \odot \boldsymbol{a}_{(h),t-1})\boldsymbol{W}_{ph}\big) \tag{127}$$

$$\boldsymbol{a}_{(h),t} = (1 - \boldsymbol{z}_t) \odot \boldsymbol{a}_{(h),t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{a}}_{(h),t} \tag{128}$$

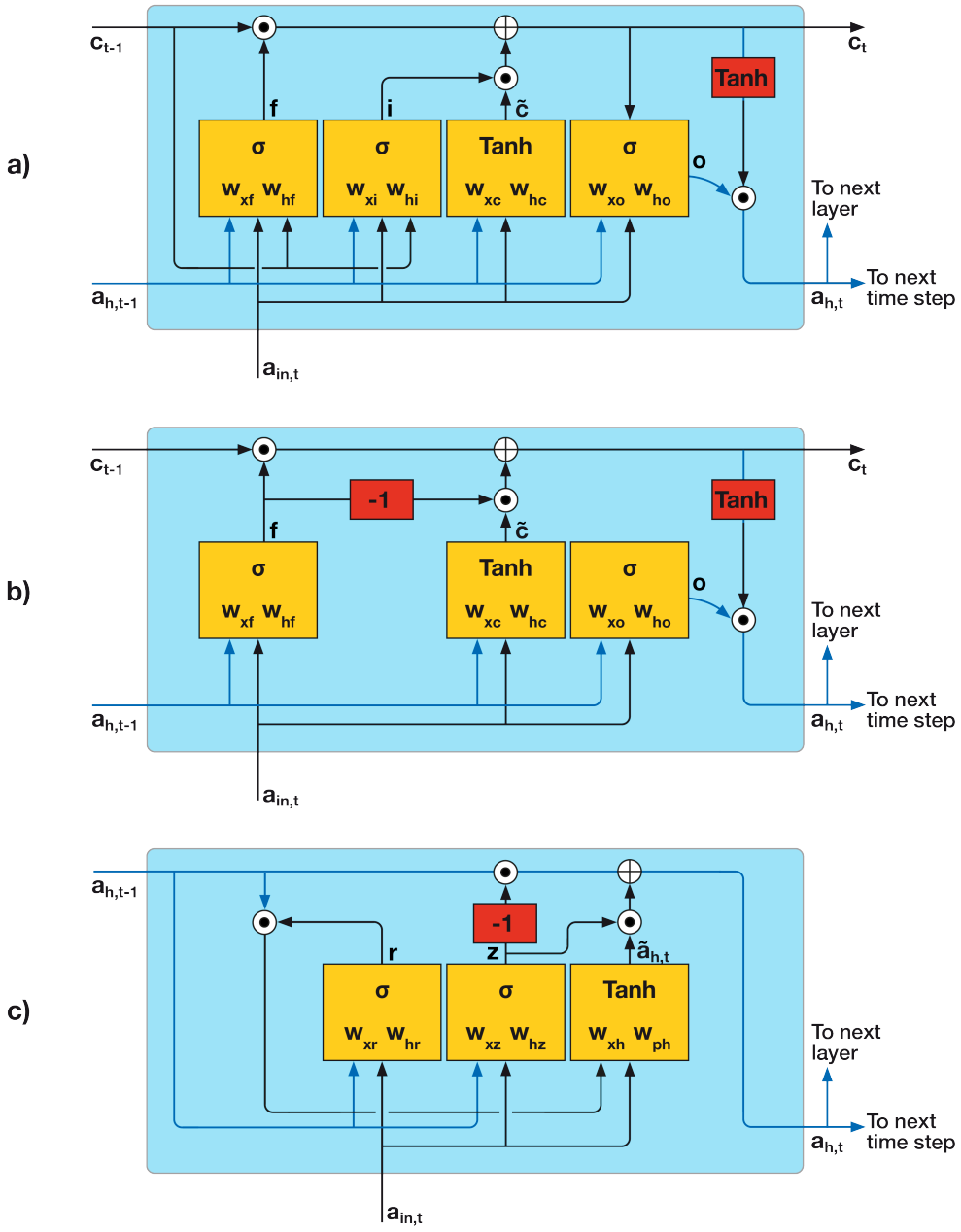

Figure 19: Variants of the LSTM

Besides these notable architectures, there are other variants of the LSTM, such as Chung et al. (2014)'s Gated RNNs, Yao et al. (2015)'s Depth Gated RNNs, as well as completely different approaches to tackling long-term dependencies, like Koutnik et al. (2014)'s Clockwork RNNs. Greff et al. (2016) and Jozefowicz, Zaremba, and Sutskever (2015) offer a comparison of the most popular variants.

I Support Vector Machine

Support Vector Machine is an algorithm for binary and multi-class classification, which learns the optimal weights by maximizing the margins, i.e. the distance between the decision boundary and the training samples that are closest to this hyperplane (called support vectors). The rationale behind the maximization of the margins is that such models are less subject to overfitting and tend to generalize better on unseen data. Figure 20 illustrates the concept of margin and support vectors.

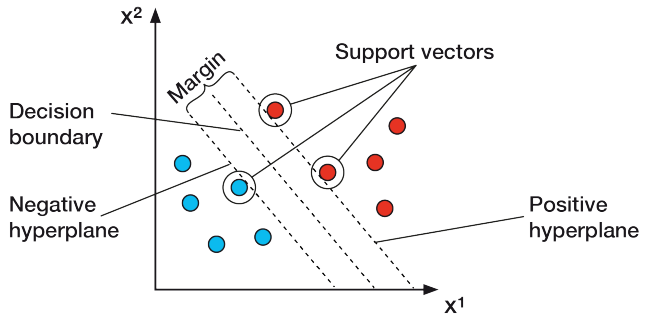

Figure 20: Maximization of the margin

Let the positive $(+)$ and negative $(-)$ hyperplanes be defined as follows:

$$\text{positive hyperplane: } w_0 + \boldsymbol{x}^{(+)}\boldsymbol{w} = 1 \tag{129}$$

$$\text{negative hyperplane: } w_0 + \boldsymbol{x}^{(-)}\boldsymbol{w} = -1 \tag{130}$$

Subtracting Equation 130 from Equation 129:

$$\left(\boldsymbol{x}^{(+)} - \boldsymbol{x}^{(-)}\right)\boldsymbol{w} = 2 \tag{131}$$

Normalizing by the length of the vector of weights $\boldsymbol{w}$:

$$\frac{\left(\boldsymbol{x}^{(+)} - \boldsymbol{x}^{(-)}\right)\boldsymbol{w}}{||\boldsymbol{w}||} = \frac{2}{||\boldsymbol{w}||} \quad \text{where} \quad ||\boldsymbol{w}|| = \sqrt{\sum_j w_j^2} \tag{132}$$

where the left hand side represents the margin to maximize, i.e. the distance between the positive and the negative hyperplanes.

Support Vector Machine maximizes the margin by maximizing the right hand side of Equation 132, subject to the constraint that the samples are correctly classified:

$$\begin{aligned} \max_{\boldsymbol{w}} \quad & \frac{2}{||\boldsymbol{w}||} \\ \text{s.t.} \quad & w_0 + \boldsymbol{x}_i \boldsymbol{w} \geq 1 \quad \text{if } y_i = 1 \\ & w_0 + \boldsymbol{x}_i \boldsymbol{w} \leq -1 \quad \text{if } y_i = -1 \end{aligned} \tag{133}$$

However, in practice, it is easier to minimize $\frac{||\boldsymbol{w}||^2}{2}$ using quadratic programming.

Support Vector Machine algorithms can be extended to deal with non-linearly separable data, using soft-margin classification. When the data are non-linearly separable, the linear constraints need to be relaxed to allow the algorithm to converge to the optimal weights, even though some samples are misclassified. To do so, a slack variable ξ is added to the objective function and its linear constraints:

$$\begin{aligned} \min_{\boldsymbol{w}} \quad & \frac{||\boldsymbol{w}||^2}{2} + C\Big(\sum_i \xi_i\Big) \\ \text{s.t.} \quad & w_0 + \boldsymbol{x}_i \boldsymbol{w} \geq 1 - \xi_i \quad \text{if } y_i = 1 \\ & w_0 + \boldsymbol{x}_i \boldsymbol{w} \leq -1 + \xi_i \quad \text{if } y_i = -1 \end{aligned} \tag{134}$$

where the parameter C allows to control the penalty for misclassification: the larger C, the more strict the algorithm is about misclassifications, i.e. the smaller the width of the margin.

J K-Nearest Neighbors

K-Nearest Neighbors is a type of lazy learner [14] for binary and multi-class classification, which does not learn an optimal decision boundary from the data, but instead memorizes the training data. While such a memory-based algorithms has the advantage that it immediately adapts to new training data, its computational complexity can grows linearly with the size of the training set. Furthermore, since the learning rule does not involve a training step, the training samples

[14] Machine learning algorithms can be grouped into parametric and non-parametric models. The former learn the optimal parameters from the training set to classify unseen data without the need of the original training samples. Examples include the Perceptron, Logistic Regression and Support Vector Machine. The latter cannot be characterized by a fixed set of parameters and the number of parameters grows with the training data. An example is the Decision Tree classifier. K-Nearest Neighbors belongs to a subcategory of non-parametric models called instance-based learning, which are characterized by memorizing the training dataset. More precisely, lazy learning is a special type of instance-based learning, which has zero cost during the learning process.

cannot be discarded and the storage requirement grows with the size of the training set.

The K-Nearest Neighbors learning rule is defined as follows:

1. Choose the number K and a distance metric.
2. Find the K samples in the training set that are closest to the sample to be classified.
3. Assign the class label of the sample by majority vote.

The choice of K is important to find a good trade-off between over- and under-fitting [15]. Moreover, usually, a Euclidead distance metric is used for samples with real-valued features, in which case it is important to standardize the data so that every feature contributes equally to the distance. The Euclidead distance metric is defined as follows:

$$d(\boldsymbol{x}_{i1}, \boldsymbol{x}_{i2}) = \sqrt{\sum_j |x_{i1}^j - x_{i2}^j|^2} \tag{135}$$

Figure 21 illustrates the learning rule.

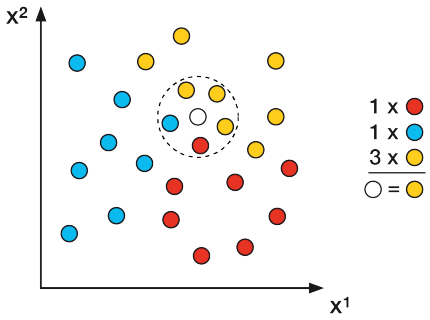

Figure 21: K-Nearest Neighbors learning rule

K Decision Tree

Decision Tree is an algorithms for binary and multi-class classification, which significantly facilitates interpretability by breaking down the data based on a series of questions, which are learnt from the features in the training set. More precisely, the learning algorithm is an iterative process, which starts at the tree root and sequentially splits the data at each node based on the feature that results in the largest information gain, until the leaves are pure, i.e. they all contain samples that belong to the same class. Since this procedure can result in a very deep tree with many sequential nodes (which could potentially result in overfitting), the tree depth is usually capped to a maximum level. Figure 22 illustrates the concept of a Decision Tree.

[15] K-Nearest Neighbors algorithms are prone to over-fitting due to the curse of dimensionality: an increasingly large training set causes the feature space to become increasingly sparse, whereby even the closest neighbors are too far to give good estimates. Since regularization techniques cannot be applied to this model, feature selection and dimensionality reduction techniques can greatly benefit the learning process.

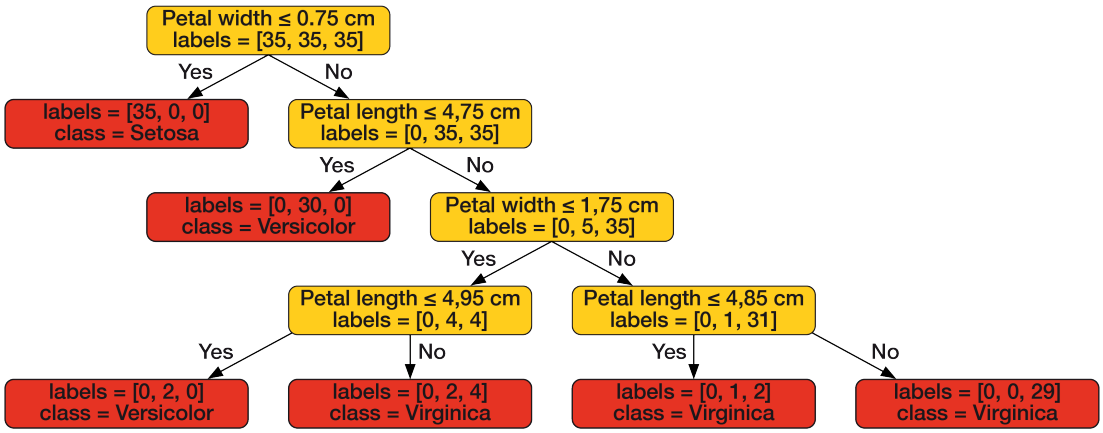

Figure 22: Decision Tree Learning

In order to find the feature that maximizes the information gain at each node, the learning algorithm introduces an objective function:

$$IG(D_p, f) = I(D_p) - \sum_j \frac{N_j}{N_p} I(D_j) \tag{136}$$

where f is the feature to perform the split, I is the impurity measure, D_p and D_j are the datasets of the parent and jth node, and N_p and N_i are respectively the total number of samples. In other words, the information gain is defined as the difference between the impurity of the parent node and the sum of the impurities of the child nodes, whereby the smaller the impurity of the latter, the higher the information gain. In practice, to reduce the computational cost, a binary Decision Tree is normally used, where each parent node is split into just two child nodes:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}) \tag{137}$$

The most common impurity measures used in Binary Decision Trees are gini impurity (I_G), entropy (I_H) and classification error (I_E).

The entropy aims at maximizing the mutual information in the tree and, for all non-empty classes $p(i|t) \neq 0$, it is defined as:

$$I_H(t) = -\sum p(i|t) log_2 p(i|t) \tag{138}$$

where $p(i|t)$ is the proportion of the samples that belong to class i for a particular node t. It follows that entropy is zero if all samples at node t belong to the same class, while it is maximized with a uniform class distribution. In particular, in a Binary Decision Tree, $I_H(t) = 0$

if $p(i=1|t)=1$ or if $p(i=0|t)=0$ and $I_H(t)=1$ if $p(i=1|t)=0.5$ and if $p(i=0|t)=0.5$

The gini impurity measure aims at minimizing the probability of misclassification, and is defined as follows:

$$I_G(t)=\sum_i p(i|t)(1-p(i|t))=1-\sum_i p(i|t)^2 \tag{139}$$

Like for entropy, the gini impurity measure is maximized with a uniform class distribution.

The classification error is defined as follows:

$$I_E(t)=1-max\{p(i|t)\} \tag{140}$$

References

[1] Luc Bauwens and Pierre Giot. *Econometric modelling of stock market intraday activity*. Vol. 38. Springer Science & Business Media, 2013.

[2] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. "Learning long-term dependencies with gradient descent is difficult". In: *IEEE transactions on neural networks* 5.2 (1994), pp. 157–166.

[3] Marshall E Blume and Michael A Goldstein. "Quotes, order flow, and price discovery". In: *The Journal of Finance* 52.1 (1997), pp. 221–244.

[4] Tim Bollerslev. "Generalized autoregressive conditional heteroskedasticity". In: *Journal of econometrics* 31.3 (1986), pp. 307–327.

[5] Christian T Brownlees and Giampiero M Gallo. "Financial econometric analysis at ultra-high frequency: Data handling concerns". In: *Computational Statistics & Data Analysis* 51.4 (2006), pp. 2232–2245.

[6] Xinpeng Chen et al. "Fine-grained video attractiveness prediction using multimodal deep learning on a large real-world dataset". In: *Companion Proceedings of the The Web Conference 2018*. 2018, pp. 671–678.

[7] Kyunghyun Cho et al. "Learning phrase representations using RNN encoder-decoder for statistical machine translation". In: *arXiv preprint arXiv:1406.1078* (2014).

[8] Junyoung Chung et al. "Empirical evaluation of gated recurrent neural networks on sequence modeling". In: *arXiv preprint arXiv:1412.3555* (2014).

[9] Rama Cont. "Empirical properties of asset returns: stylized facts and statistical issues". In: *Quantitative Finance* 1.2 (2001), pp. 223–236.

[10] Robert F Engle. "Autoregressive conditional heteroscedasticity with estimates of the variance of United Kingdom inflation". In: *Econometrica: Journal of the Econometric Society* (1982), pp. 987–1007.

[11] Lijie Fan et al. "End-to-end learning of motion representation for video understanding". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 6016–6025.

[12] Yang Feng et al. "Video re-localization". In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 51–66.

[13] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*. Vol. 1. 10. Springer series in statistics New York, 2001.

[14] Felix A Gers and Jürgen Schmidhuber. “Recurrent nets that time and count”. In: *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*. Vol. 3. IEEE. 2000, pp. 189–194.

[15] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. “Learning to forget: Continual prediction with LSTM”. In: *9th International Conference on Artificial Neural Networks: ICANN ’99* (1999), pp. 850–855.

[16] Lawrence R Glosten, Ravi Jagannathan, and David E Runkle. “On the relation between the expected value and the volatility of the nominal excess return on stocks”. In: *The journal of finance* 48.5 (1993), pp. 1779–1801.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[18] Klaus Greff et al. “LSTM: A search space odyssey”. In: *IEEE transactions on neural networks and learning systems* 28.10 (2016), pp. 2222–2232.

[19] Joel Hasbrouck, George Sofianos, and Deborah Sosebee. “New York Stock Exchange systems and trading procedures”. In: *Director* 212.998 (1993), p. 0310.

[20] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[21] John Hull. *Options, futures and other derivatives, 9th edition*. Pearson, 2015.

[22] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. “An empirical exploration of recurrent network architectures”. In: *International conference on machine learning*. 2015, pp. 2342–2350.

[24] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[25] Roger Koenker and Gilbert Bassett Jr. “Regression quantiles”. In: *Econometrica: journal of the Econometric Society* (1978), pp. 33–50.

[26] Roger Koenker and Kevin F Hallock. “Quantile regression”. In: *Journal of economic perspectives* 15.4 (2001), pp. 143–156.

[27] Jan Koutnik et al. “A clockwork rnn”. In: *arXiv preprint arXiv:1402.3511* (2014).

[28] Charles MC Lee and Mark J Ready. “Inferring trade direction from intraday data”. In: *The Journal of Finance* 46.2 (1991), pp. 733–746.

[29] Ananth Madhavan and George Sofianos. “An empirical analysis of NYSE specialist trading”. In: *Journal of Financial Economics* 48.2 (1998), pp. 189–210.

[30] Warren S McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133.

[31] Nasdaq. *NLS Plus*. Vesion 3.0. Aug. 2018.

[32] Nasdaq. *The UTP Plan Trade Data Feed-SM (UTDF-SM)*. Version 14.4. Nov. 2015.

[33] Daniel B Nelson. “Conditional heteroskedasticity in asset returns: A new approach”. In: *Econometrica: Journal of the Econometric Society* (1991), pp. 347–370.

[34] Paul Newbold, William Carlson, and Betty Thorne. *Statistics for business and economics*. Pearson, 2012.

[35] NYSE. *Daily TAQ Client Specification*. Vesrion 3.0. Nov. 2017.

[36] NYSE. *TAQ User’s Guide*. Version 3.31. Nov. 2012.

[38] Roel C A Oomen. “Properties of realized variance under alternative sampling schemes”. In: *Journal of Business & Economic Statistics* 24.2 (2006), pp. 219–237.

[39] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. “On the difficulty of training recurrent neural networks”. In: *International conference on machine learning*. 2013, pp. 1310–1318.

[40] Sebastian Raschka. *Python machine learning*. Packt Publishing Ltd, 2015.

[41] Frank Rosenbaltt. “The perceptron - a perciving and recognizing automaton”. In: *Report 85-460-1 Cornell Aeronautical Laboratory, Ithaca, Tech. Rep.* (1957).

[42] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. “Learning representations by back-propagating errors”. In: *Cognitive modeling* 5.3 (1988), p. 1.

[43] George Sofianos and Ingrid M Werner. “The trades of NYSE floor brokers”. In: *Journal of Financial Markets* 3.2 (2000), pp. 139–176.

[44] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems*. 2017, pp. 5998–6008.

[45] Olivier Vergote. “How to Match Trades and Quotes for NYSE Stocks?” In: *Available at SSRN 808984* (2005).

[46] Bairui Wang et al. “Reconstruction network for video captioning”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 7622–7631.

[47] Jingwen Wang et al. “Bidirectional attentive fusion with context gating for dense video captioning”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 7190–7198.

[48] Paul J Werbos. “Backpropagation through time: what it does and how to do it”. In: *Proceedings of the IEEE* 78.10 (1990), pp. 1550–1560.

[49] B Widrow. *An adaptive'ADALINE'neuron using chemical'memistors', 1553-1552*. 1960.

[50] David H Wolpert. “The lack of a priori distinctions between learning algorithms”. In: *Neural computation* 8.7 (1996), pp. 1341–1390.

[51] Xing Yan et al. “Parsimonious Quantile Regression of Financial Asset Tail Dynamics via Sequential Learning”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 1582–1592.

[52] K Yao et al. “Depth-Gated Recurrent Neural Networks”. In: *arXiv preprint arXiv:1508.03790* 9 (2015).

Internet Resources

[23] Andrej Karpathy. *The Unreasonable Effectiveness of Recurrent Neural Networks*. May 21, 2015. URL: `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`.

[37] Christopher Olah. *Understanding LSTM Networks*. Aug. 27, 2015. URL: `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`.

Declaration of authorship

You have to append the following declaration to your thesis and put your signature to it:

"I hereby declare

- that I have written this thesis without any help from others and without the use of documents and aids other than those stated above;
- that I have mentioned all the sources used and that I have cited them correctly according to established academic citation rules;
- that I have acquired any immaterial rights to materials I may have used such as images or graphs, or that I have produced such materials myself;
- that the topic or parts of it are not already the object of any work or examination of another course unless this has been explicitly agreed on with the faculty member in advance and is referred to in the thesis;
- that I will not pass on copies of this work to third parties or publish them without the University's written consent if a direct connection can be established with the University of St.Gallen or its faculty members;
- that I am aware that my work can be electronically checked for plagiarism and that I hereby grant the University of St.Gallen copyright in accordance with the Examination Regulations in so far as this is required for administrative action;
- that I am aware that the University will prosecute any infringement of this declaration of authorship and, in particular, the employment of a ghostwriter, and that any such infringement may result in disciplinary and criminal consequences which may result in my expulsion from the University or my being stripped of my degree."

Date and signature

18/05/2020, Nicolo Cenedo

By submitting this academic term paper, I confirm through my conclusive action that I am submitting the Declaration of Authorship, that I have read and understood it, and that it is true.