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

March 10, 2020

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

Abstract

This is my abstract

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

Contents

List of Tables

List of Figures

Key Notation tenets

$\hbar$	Reduced constant

List of Symbols

$\hbar$	Reduced constant

1 Introduction

While machine learning models are generally concerned with the prediction of a single value of the output variable y, given the input x, to estimate the conditional mean $E[y|x]$, some situations also require the estimation of the parameters of the conditional distribution $p(y|x)$. This is the case for financial assets, which behave stochastically and have leptokurtic, asymmetric and time-varying tails, both conditionally and unconditionally. Although it is impossible to accurately predict their future returns, it is possible to predict the characteristics of the conditional distribution of their returns. 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 (GARCH) model and its variations. These models describe the conditional distribution by making assumptions on its probability density function, and let the distribution parameters depend on past observations. More precisely, the GARCH models the behavior of financial asset returns with a t-distribution, which manages to capture the heaviness of the tails but fails to account for their asymmetry and their time-varying behavior. Other studies model the time-varying conditional skewness and kurtosis in an autoregressive way, but assume elaborated probability density functions, which complicate the estimation process.

Another method used to predict the characteristics of the conditional distribution is quantile regression, which forecasts the conditional quantiles without making assumptions on the probability density function. Indeed, besides the probability density function, quantiles are another way to describe the shape of the tails: modeling the conditional quantiles for a finite set of probability levels is almost equivalent to modeling the conditional distribution, i.e. estimating the conditional mean, volatility, skewness and kurtosis. 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 when estimating more quantiles and the lack of interpretability. To solve these problems, this paper forecasts conditional quantiles and heavy tails of series of financial asset returns via a parsimonious quantile regression that describes the conditional distribution $p(r_t|r_{t-1}, r_{t-2}, \dots)$ with a parametric conditional quantile function, rather than assuming the probability density function, which may introduce tractability and ill-posedness issues. More precisely, this paper proposes a parametric heavy-tailed quantile function (HTQF) with time-varying parameters, which depend on past information through a Long-Short Term Memory unit (LSTM) and applies it to model a conditional distribution with

time varying leptokurtic and asymmetric tails. The parameters of the LSTM unit are learnt in a quantile regression framework with multiple probability levels. After training, the conditional quantiles and the parameters of the HTQF can be estimated.

The contributions of this paper are: proposing a novel parametric quantile function to represent a distribution with leptokurtic and asymmetric tails and use it to model the conditional distribution of return series of financial assets; implementing a LSTM unit that can be trained in a quantile regression framework to learn the time varying tail behavior and predict conditional quantiles with more accuracy; overcoming the disadvantages of traditional quantile regressions.

2 Literature Review

2.1 The evolution Deep Learning

The first wave of development of DL was known as cybernetics and took place in the 1940s-1960s. DL algorithms were designed to be computational models of biological learning, i.e. to resemble the way learning occurs in a biological brain. For this reason, DL has also been called artificial neural networks (ANN). mcculloch1943's neuron was a model of brain function capable to classify into two categories a set of n input values $x\_1, \ldots, x\_n$, by testing whether $f(\mathbf{x}, \mathbf{w}) = x\_1 w\_1 + \cdots + x\_n w\_n$ was positive or negative, given a set of weights $w\_1, \ldots, w\_n$. rosenblatt1957's Perceptron became the first model that could autonomously learn the set of weights from a training set. widrow1960's adaptive linear neuron simply returned the value of $f(\mathbf{x})$ to predict a real number. These three models belong to the class linear models. Although the latter remain among the most widely used types of ML, their limitations have caused a backlash against biologically inspired learning in general. Moreover, although researchers have successfully drawn from neuroscience to design the architecture of several neural networks, there is too little knowledge about the process of biological learning for them to do the same for the learning algorithms used to train the architectures. For this reason, neuroscience remains an important source of knowledge for ML models but it is no longer the predominant one. Rather, modern DL draws heavily from the fields of statistics and applied mathematics and goes beyond the neuro-scientific perspective to define the principle of hierarchy of concepts, which can be applied also to ML algorithms not inspired by neuroscience.

The second wave of development of DL was known as connectionism and took place in the 1080s-1090s. This time, DL algorithms were developed in the context of cognitive sciences. The main idea was that a large number of simple computational units can achieve an intelligent behavior when connected together. One of the key concepts developed in this period was that

of distributed representation, i.e. the idea that each input to a system should be represented by many features and each feature should be involved in the representation of many possible inputs. Another key concept was that of back-propagation, which is still the dominant approach to train deep models. In this period, hochreiter1997 introduced a long-short term memory network to model long sequences. However, shortly after, the combined failure of DL in meeting the expectations and the advances in other fields of ML led researchers to loose interest in DL models for a second time.

The third and current wave of development of DL started in 2006, when hinton2006 proved that a deep belief neural network could be efficiently trained via a greedy layer-wise pre-training strategy. Before this breakthrough, deep networks were computationally too difficult to train. Now however, researchers are able to train deep neural networks and achieve results superior to those of competing AI systems based on other types of ML. Current research focuses particularly on supervised ML, where DL algorithms are trained on labeled data sets.

The current success of DL can be traced back to two key developments: the increased quantity of available data and the improvements in computing power. In particular, the latter has allowed to construct networks with much larger architectures and better performances. Although biological neurons are not densely connected and ML architectures have had a similar number of connections per neuron for decades, the total number of neurons in neural networks has been very small until recently, when hidden units were first introduced. Despite these technological developments, it will take until 2050 for neural networks to reach the same number of neurons as a human brain. On the top of this, biological neurons may implement functions more complicated than those embedded in artificial neurons.

2.2 What is Deep Learning

In its early days, the field of artificial intelligence (AI) was concerned with the solution of problems that, although intellectually challenging for humans, could be described by a set of formal rules. For example, IBM's Deep Blue chess-playing system became the first computer to defeat then-reigning World Chess Champion Garry Kasparov in 1997. Nowadays instead, AI is mainly concerned with the solution of problems that are easy for people to tackle but difficult to describe formally. For example, humans are able to intuitively recognize the voice of a friend or the content of an image, but cannot easily state the formal rules that they use to come to the conclusion. Many AI projects have attempted to use a knowledge based approach, which consists in hard-coding rules to make such decisions via logical inference, but have turned out to be unsuccessful. The capability of computers to extract features from data and formulate

their own knowledge, rather than relying on hard-coded human inputs, is known as machine learning (ML). More specifically, one approach of AI to these intuitive problems is deep learning (DL), which allows computers to learn from their experience and describe the environment in terms of a hierarchy of concepts, where each concept is defined in terms of its relation to simpler ones.

The performance of ML algorithms is strongly influenced by the representation of the input data. For example, a useful feature for the task of voice recognition may be the size of the speaker's vocal tract, which allows to distinguish a man from a woman and a child from an adult, rather than the color of the speaker's air. Since for most tasks it is difficult to understand what features should be extracted, representation learning applies a ML approach not only to map the features to the output but also to extract the features themselves. More specifically, DL solves the problem by using representations that are expressed in terms of simpler representations.

Due to the absence of a precise measure of the depth of an architecture, there is no specific threshold defining what constitutes DL. However, generally, DL can be regarded as the study of models that imply learning multiple levels of composition.

3 Theory Review

3.1 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.

3.2 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.

$$
\begin{array}{l}
\\
\text{Samples} \downarrow
\end{array}
\quad
\boldsymbol{X}:
\begin{array}{c}
\overrightarrow{\text{Features}} \\
\begin{array}{cccc} \boldsymbol{x}^1 & \boldsymbol{x}^2 & \dots & \boldsymbol{x}^m \end{array} \\
\begin{array}{c} \boldsymbol{x}_1 \\ \boldsymbol{x}_2 \\ \dots \\ \boldsymbol{x}_n \end{array}
\begin{pmatrix} x_1^1 & x_1^2 & \dots & x_1^m \\ x_2^1 & x_2^2 & \dots & x_2^m \\ \dots & \dots & \dots & \dots \\ x_n^1 & x_n^2 & \dots & x_n^m \end{pmatrix}
\end{array}
\qquad
\boldsymbol{y}:
\begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}
$$

3.3 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.

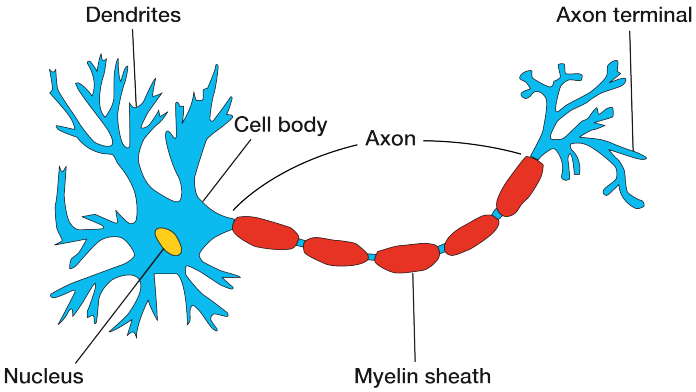


Figure 1: Neuron

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.

3.4 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:

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

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

$$z_i = \boldsymbol{x}_i \boldsymbol{w} = x_i^1 w_1 + \cdots + x_i^m w_m \tag{2}$$

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:

$$\phi(Z) = \begin{cases} 1 & \text{if } Z \geq \theta \\ -1 & \text{otherwise} \end{cases} \tag{3}$$

For convenience, Equations 2 and 3 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 $x_i^0 = 1$:

$$z_i = x_i^0 w_0 + \boldsymbol{x}_i \boldsymbol{w} = x_i^0 w_0 + x_i^1 w_1 + \cdots + x_i^m w_m \tag{4}$$

$$\phi(Z) = \begin{cases} 1 & \text{if } Z \geq 0 \\ -1 & \text{otherwise} \end{cases} \tag{5}$$

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:

$$w_j := w_j + \Delta w_j \quad \text{where} \quad \Delta w_j = \eta(y_i - \hat{y}_i)x_i^j \tag{6}$$

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.

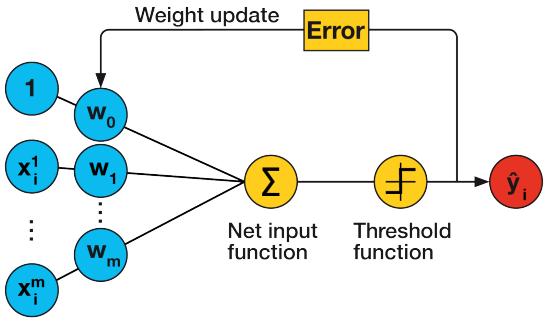


Figure 2: Perceptron learning rule

3.5 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:

$$\boldsymbol{X} = \begin{pmatrix} x_1^1 & \dots & x_1^m \\ \dots & \dots & \dots \\ x_n^1 & \dots & x_n^m \end{pmatrix} \qquad \boldsymbol{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_m \end{pmatrix} \tag{7}$$

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:

$$\boldsymbol{z} = \boldsymbol{w}_0 + \boldsymbol{X}\boldsymbol{w} \tag{8}$$

Note that, differently from Equation 4, 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:

$$\phi(Z) = Z \tag{9}$$

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 5 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:

$$J(\boldsymbol{w}) = \frac{1}{2}\sum_i \Big(y_i - \phi(z_i)\Big)^2 \tag{10}$$

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:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w} \quad \text{where} \quad \Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w}) \tag{11}$$

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

$$\Delta w_j = \eta\sum_i \Big(y_i - \phi(z_i)\Big)x_i^j \tag{12}$$

By comparing Equations 6 and 12 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.

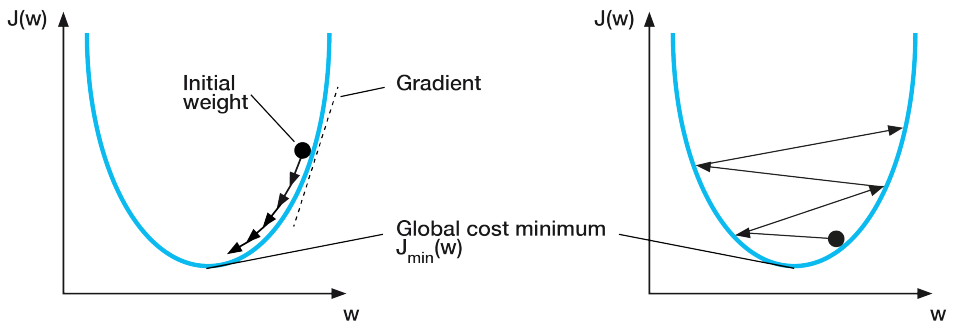


Figure 3: Gradient descent algorithm and overshooting

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:

$$\boldsymbol{x}^j := \frac{\boldsymbol{x^j} - \mu^j}{\sigma^j} \tag{13}$$

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:

$$w_j := w_j + \Delta w_j \quad \text{where} \quad \Delta w_j = \eta \sum_i \Big(y_i - \phi(z_i)\Big) x_i^j \tag{14}$$

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.

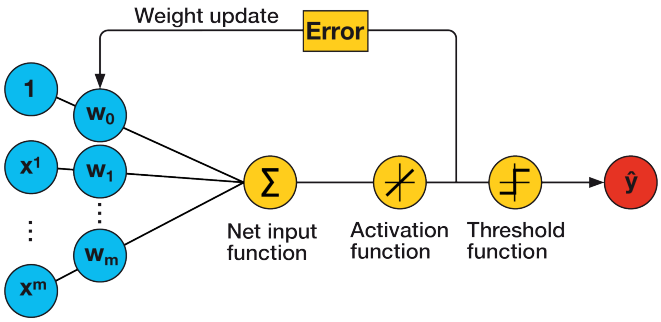


Figure 4: Adaline learning rule

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.

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

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.

3.6 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 9) 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:

$$\phi(Z) = \frac{1}{1 + e^{-Z}} \tag{16}$$

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:

$$\hat{Y} = \begin{cases} 1 & \text{if } \phi(Z) \geq 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{17}$$

which is equivalent to:

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

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:

$$J(\boldsymbol{w}) = \sum_i \Big[-y_i \, log\Big(\phi(z_i)\Big) - (1-y_i) \, log\Big(1-\phi(z_i)\Big)\Big] \tag{19}$$

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:

$$\boldsymbol{w} := \boldsymbol{w} + \Delta \boldsymbol{w} \quad \text{where} \quad \Delta \boldsymbol{w} = -\eta \nabla J(\boldsymbol{w}) \tag{20}$$

which results in:

$$\Delta w_j = \eta \sum_i \Big(y_i - \phi(z_i)\Big) x_i^j \tag{21}$$

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.

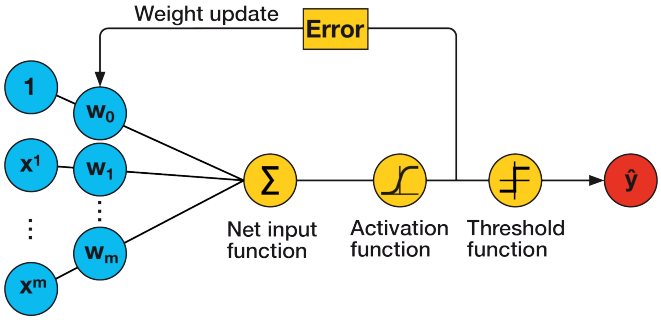


Figure 5: Logistic Regression learning rule

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:

$$J(\boldsymbol{w}) = \sum_i \Big[-y_i \, log\Big(\phi(z_i)\Big) - (1-y_i) \, log\Big(1-\phi(z_i)\Big)\Big] - \frac{\lambda}{2}||\boldsymbol{w}||^2 \tag{22}$$

3.7 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 6 illustrates the concept of margin and support vectors.

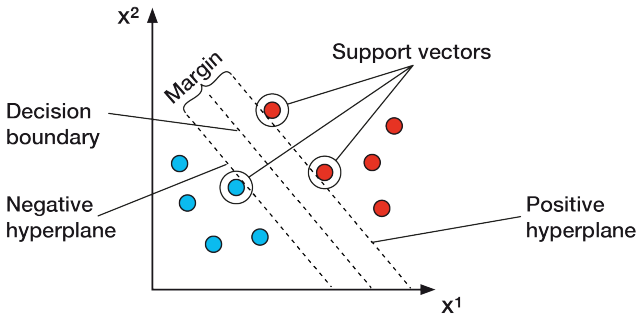


Figure 6: Maximization of the margin

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

$$\text{positive hyperplane: } w_0 + \boldsymbol{x}^{(+)}\boldsymbol{w} = 1 \tag{23}$$

$$\text{negative hyperplane: } w_0 + \boldsymbol{x}^{(-)}\boldsymbol{w} = -1 \tag{24}$$

Subtracting Equation 24 from Equation 23:

$$\left(\boldsymbol{x}^{(+)} - \boldsymbol{x}^{(-)}\right)\boldsymbol{w} = 2 \tag{25}$$

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

$$\frac{\left(\boldsymbol{x}^{(+)} - \boldsymbol{x}^{(-)}\right)\boldsymbol{w}}{||\boldsymbol{w}||} = \frac{2}{||\boldsymbol{w}||} \quad \text{where} \quad ||\boldsymbol{w}|| = \sqrt{\sum_j w_j^2} \tag{26}$$

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 26, subject to the constraint that the samples are correctly classified:

$$\begin{aligned} \max_{\boldsymbol{w}} \quad & \frac{2}{||\boldsymbol{w}||} \\ \text{s.t.} \quad & w_0 + \boldsymbol{x}_i \boldsymbol{w} \geq 1 \quad \text{if } y_i = 1 \\ & w_0 + \boldsymbol{x}_i \boldsymbol{w} \leq -1 \quad \text{if } y_i = -1 \end{aligned} \tag{27}$$

However, in practice, it is easier to minimize $\frac{||\boldsymbol{w}||^2}{2}$ using quadratic programming. Appendix F briefly explains how Support Vector Machine algorithms can be extended to deal with non-linearly separable data, using soft-margin classification.

3.8 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 7 illustrates the concept of a Decision Tree.

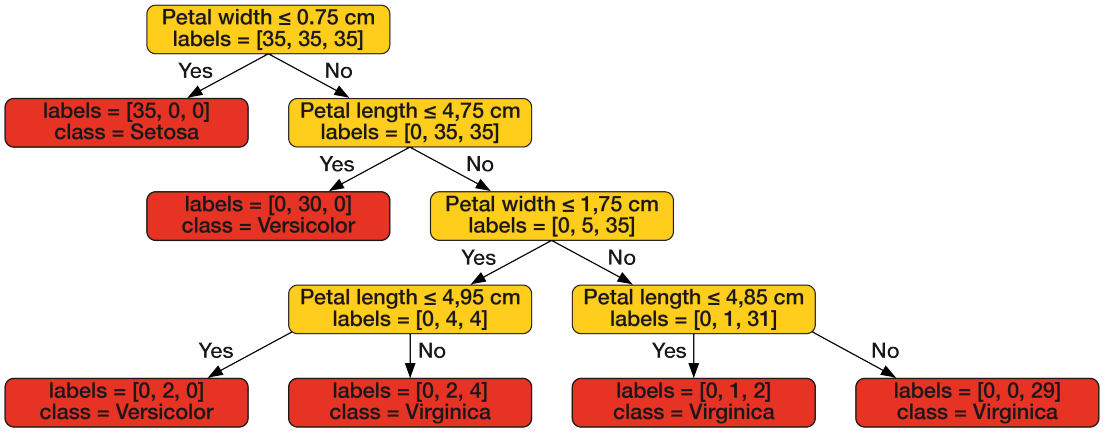


Figure 7: Decision Tree Learning

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

$$IG(D_p, f) = I(D_p) - \sum_j \frac{N_j}{N_p} I(D_j) \tag{28}$$

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:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}) \tag{29}$$

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:

$$I_H(t) = -\sum p(i|t) log_2 p(i|t) \tag{30}$$

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:

$$I_G(t) = \sum_i p(i|t)(1 - p(i|t)) = 1 - \sum_i p(i|t)^2 \tag{31}$$

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

The classification error is defined as follows:

$$I_E(t) = 1 - max\{p(i|t)\} \tag{32}$$

3.9 K-Nearest Neighbors

K-Nearest Neighbors is a type of lazy learner [3] 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 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 [4]. 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:

$$d(\boldsymbol{x}_{i1}, \boldsymbol{x}_{i2}) = \sqrt{\sum_j |x_{i1}^j - x_{i2}^j|^2} \tag{33}$$

Figure 8 illustrates the learning rule.

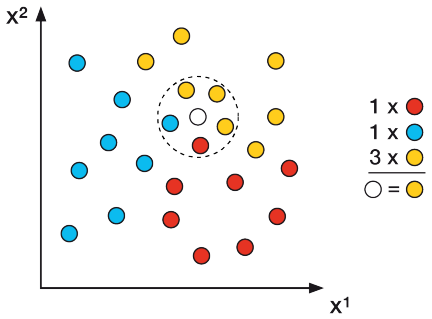


Figure 8: K-Nearest Neighbors learning rule

[3] 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.

[4] 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.

3.10 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 [5]. Figure 9 illustrates the architecture of the Multilayer Perceptron.

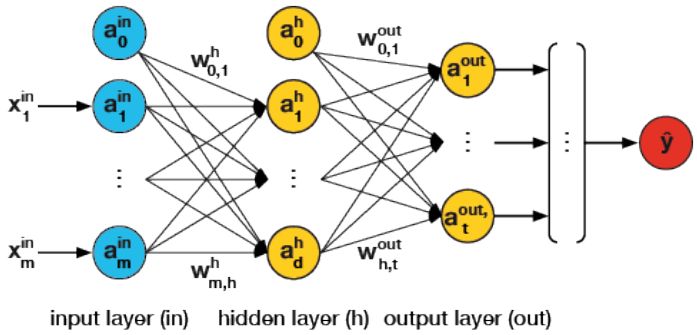


Figure 9: Multilayer Perceptron

Since a Multilayer Perceptron is generally trained via mini-batch learning, the mathematics behind the descriptions of the network's architecture and learning rule is more complicated that ideal for pedagogical reasons. Therefore, before moving forward, the reader is advised to go through Appendix G, which illustrates the architecture and the learning rule, assuming that the algorithm processes one training sample at the time (like in the case of the Perceptron). The remainder of this Section repeats the explanation in the realistic case of mini-batch learning.

Let $\boldsymbol{X}^{mb} \in \mathbb{R}^{(bs\ x\ m)}$ be the matrix of features for a batch of samples mb:

$$\boldsymbol{X}^{mb} = \begin{pmatrix} x_1^1 & \dots & x_1^m \\ \dots & \dots & \dots \\ x_{bs}^1 & \dots & x_{bs}^m \end{pmatrix} \tag{34}$$

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 $\boldsymbol{a}^0_{(in)}$, $\boldsymbol{a}^0_{(h)}$ are the bias units, set to 1. Then the activation of the units in the input layer is just the inputs values plus

[5] 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).

the bias unit:

$$\boldsymbol{a}_{(in)} = \begin{pmatrix} a^0_{(in)} & a^1_{(in)} & \dots & a^m_{(in)} \end{pmatrix} = \begin{pmatrix} \mathbf{1} & \boldsymbol{x}^1 & \dots & \boldsymbol{x}^m \end{pmatrix} \tag{35}$$

Since each activation unit $a^j_{(l)}$ in a layer l is connected to each unit $a^k_{(l+1)}$ in the next layer $(l+1)$ via a weight coefficient $w^{j,k}_{(l+1)}$, 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 $\boldsymbol{W}_{(out)} \in \mathbb{R}^{(d+1\ x\ t)}$:

$$\boldsymbol{W}_{(h)} = \begin{pmatrix} w^{0,1}_{(h)} & \dots & w^{0,d}_{(h)} \\ \dots & \dots & \dots \\ w^{m,1}_{(h)} & \dots & w^{m,d}_{(h)} \end{pmatrix} \qquad \boldsymbol{W}_{(out)} = \begin{pmatrix} w^{0,1}_{(out)} & \dots & w^{0,t}_{(out)} \\ \dots & \dots & \dots \\ w^{d,1}_{(out)} & \dots & w^{d,t}_{(out)} \end{pmatrix} \tag{36}$$

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 categorical variables can be represented with the one-hot representation: for example, the class labels $Setosa = 0$, $Versicolor = 1$, $Virginica = 2$ can be encoded as follows:

$$0 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad 1 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \qquad 2 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{37}$$

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

1. Forward propagate the patterns of the training data through the network to generate the output.
2. From the output, calculate the error to be minimized using a cost function.
3. Backpropagate the error, calculate its derivative with respect to each weight and update the model. Appendix H explains the intuition behind the backpropagation algorithm.

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 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:

$$\boldsymbol{z}_{(h)} = \boldsymbol{a}_{(in)}\boldsymbol{W}_{(h)} = \boldsymbol{X}^{mb}\boldsymbol{W}_{(h)} = \begin{pmatrix} z^1_{(h),1} & \dots & z^d_{(h),1} \\ \dots & \dots & \dots \\ z^1_{(h),bs} & \dots & z^d_{(h),bs} \end{pmatrix} \tag{38}$$

$$\boldsymbol{A}^h = \phi(\boldsymbol{Z}^h) \tag{39}$$

where $\boldsymbol{A}^{in} \in \mathbb{R}^{(bs\ x\ m+1)}$ is the matrix of features and bias units 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 (logistic) function, which maps a real input value into a continuous range between 0 and 1:

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

Similarly, for the output layer:

$$\boldsymbol{Z}^{out} = \boldsymbol{A}^h\boldsymbol{W}^{out} \tag{40}$$

$$\boldsymbol{A}^{out} = \phi(\boldsymbol{Z}^{out}) \tag{41}$$

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 $\boldsymbol{A}^{out} \in \mathbb{R}^{(bs\ x\ t)}$.

Therefore, a MLP designed to solve a multi-class classification task 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 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:

$$\boldsymbol{\delta}^{out} = \boldsymbol{A}^{out} - \boldsymbol{Y}^{oh} \tag{42}$$

Next, it is possible to back propagate the error and calculate the matrix $\boldsymbol{\delta}^h \in \mathbb{R}^{(bs\ x\ d)}$ of errors of the hidden layer as follows (Appendix I shows the derivation of the formula):

$$\boldsymbol{\delta}^h = \boldsymbol{\delta}^{out}\big(\boldsymbol{W}^{out}\big)^\top \odot \big(\boldsymbol{A}^h \odot (1 - \boldsymbol{A}^h)\big) \tag{43}$$

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 product. After calculating the error matrices, the weights can be updated. 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 MLP returns a vector

of length t for each sample, the cost function needs to be generalized to all t activation units. Moreover, an L2-regularization term is added to avoid overfitting (see Appendix E for a more detailed explanation):

$$J(\boldsymbol{W}) = \sum_{i=1}^{n}\sum_{j=1}^{t}\Big[-y_j^{oh,i}\ log(a_j^{out,i}) - (1-y_j^{oh,i})\ log(1-a_j^{out,i})\Big] + \frac{\lambda}{2}\sum_{l\in\{h,out\}}\sum_{j=1}^{u_{l-1}}\sum_{j=1}^{u_l}\Big(w_{j,j}^{l}\Big)^2 \quad (44)$$

where $y_j^{oh,i}$ is an element of the one-hot representation matrix $\boldsymbol{Y}^{oh} \in \mathbb{R}^{(n\ x\ t)}$ for the whole sample, and $a_j^{out,i}$ is an element from the activation matrix $\boldsymbol{A}^{out} \in \mathbb{R}^{(n\ x\ t)}$ for the whole sample, u is the number of units in a layer, and where the regularization term calculates the squared sum of all weights (excluding the bias term) in the weight matrices. The weight updates are as follows:

$$\boldsymbol{W}^h := \boldsymbol{W}^h - \eta\Delta\boldsymbol{W}^h \quad \text{where} \quad \Delta\boldsymbol{W}^h = (\boldsymbol{A}^{in})^\top\boldsymbol{\delta}^h + \frac{\lambda}{2}\boldsymbol{W}^h \quad (45)$$

$$\boldsymbol{W}^{out} := \boldsymbol{W}^{out} - \eta\Delta\boldsymbol{W}^{out} \quad \text{where} \quad \Delta\boldsymbol{W}^{out} = (\boldsymbol{A}^{h})^\top\boldsymbol{\delta}^{out} + \frac{\lambda}{2}\boldsymbol{W}^{out} \quad (46)$$

where the correction term is added to all weights except the bias unit [6].

3.11 Recurrent Neural Network

3.11.1 Sequential Data

Typical machine learning algorithms for supervised learning assume that the input data is independently and identically distributed. In other words, given n samples $[\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^n]^\top$, the order in which this data is presented to an algorithm is irrelevant. However, this is no longer the case when working with a sequence $[\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots, \boldsymbol{x}^t]^\top$, which is a series of ordered data. In particular, time series data represent a subset of sequential data, where each sample $\boldsymbol{x}^t$ belongs to the specific ordered time t.

Traditional neural networks fail to recognize the order in the input samples: for example, in a Multi-layer Perceptron the samples go through the feedforward and backpropagation steps and the weights get updated, independently of the order in which the samples are processed. Differently, recurrent neural networks (RNN) have memory of the past information and are capable of handling sequences and process new information accordingly. In other words, while traditional neural networks process a fixed-size input vector through a fixed number of computational steps to produce a fixed-size output vector, RNNs allow to operate with sequences of vectors at the input level, output level, or both. This idea is clarified by karpathy2015,

[6] $\boldsymbol{W}^{out}$ in $\frac{\lambda}{2}\boldsymbol{W}^{out}$ has therefore a first row of zeros

who showed that there are different types of relationships between input and output data, as illustrated in Figure 10, where the blue circles represent the input vectors, the green circles represent the vectors containing the RNN's state, the yellow circles represent the output vectors and the arrows represent operations.

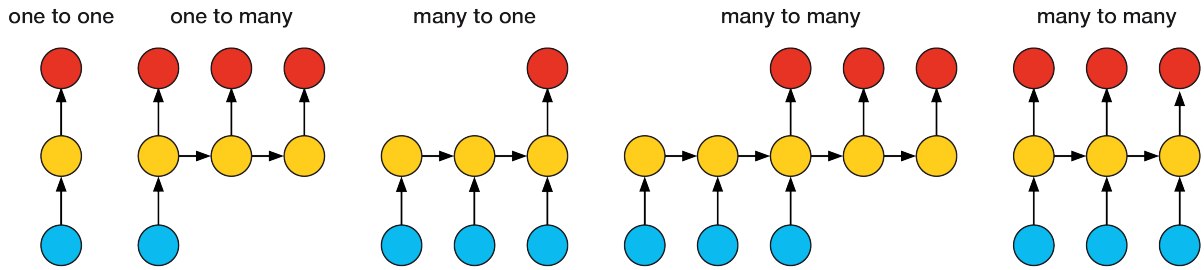


Figure 10: Types of relationships between input and output data

The *one to one* relationship represents the case of non-sequential data, where both the input and the output are fixed-size vectors, which can be modeled by traditional neural networks (e.g image classification). All the other cases involve at least one sequence and must be modeled by 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 an 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).

3.11.2 Memory of Sequential Data

RNNs differ from traditional NNs in that they contain loops (a structure knows as recurrent hedge), as shown in Figure 11, which allow information to persist in time. More precisely,

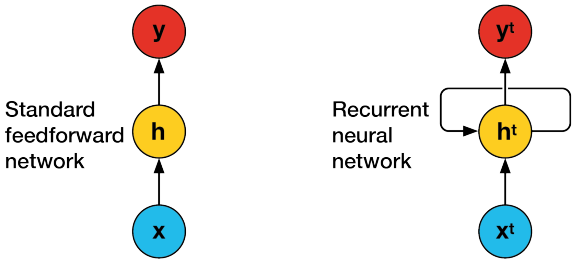


Figure 11: RNNs contain loops

while in the former information flows from the input layer to the hidden layer and then from the hidden layer to the output layer, in the latter information flows to the hidden layer from both the input layer and the hidden layer of the previous time step and then from the hidden

layer to the output layer. At the first time step $t = 0$, the hidden units are initialized to small random values; then at all subsequent time steps $t > 0$, the hidden units receive as input both the preactivation (net input) of the current data $\boldsymbol{x}^t$ from the input layer and the activation of the previous time step $\boldsymbol{h}^{t-1}$ from the same hidden layer. This flow of information in the hidden layer from one time stamp to the next is what allows a RNN to have a memory. This is illustrated in Figure 12, which shows the recurrent hedge and the unfolded representations for a single layer and a two-layer RNN. In other words, a RNN ca be thought of as a series of copies of the same network.

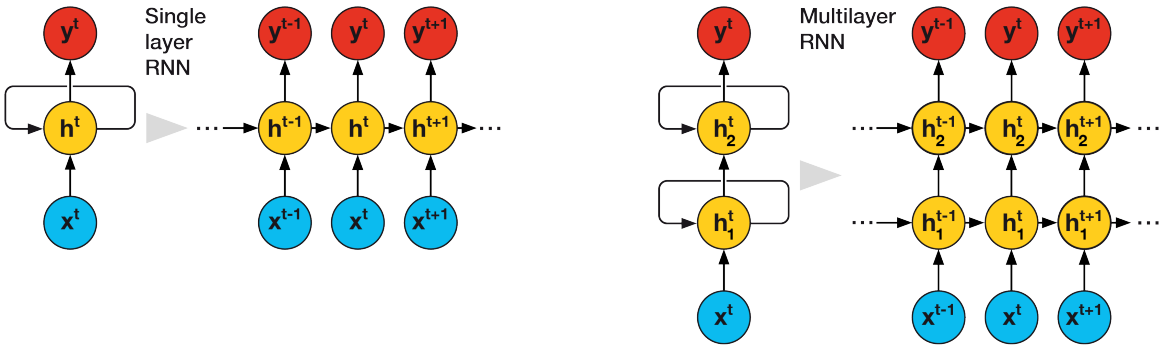


Figure 12: Recurrent hedge and unrolled representations of a single layer and a two-layer RNN

3.11.3 Activation Functions

Let's consider the single layer RNN shown in Figure 13. Each arrow (called directed edge) is associated with a weight matrix, which is independent of time: $\boldsymbol{W}_{xh}$ is the weight matrix between the input and the hidden layers, $\boldsymbol{W}_{hh}$ is the weight matrix of the recurrent hedge, and $\boldsymbol{W}_{xy}$ is the weight matrix between the hidden and the output layers.

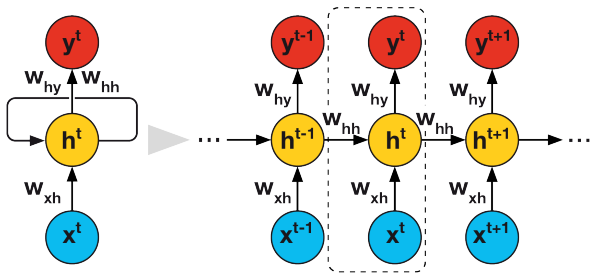


Figure 13: Weight matrix in a RNN

The activations of a RNN are similar to those of the Perceptron presented above. Let

4 Machine Learning Algorithms

To implement a machine learning algorithm can be divided into three main steps.

4.1 Data Preprocessing

Raw data rarely comes in a form that is suited to the optimal performance of a learning algorithm. Therefore, it is first necessary to select and extract meaningful features from the raw data and transform them on the same scale for optimal performance (usually in the range [0,1] or a standard normal distribution). Furthermore, if some of the selected features are highly correlated, and therefore redundant, the features can be compressed onto a lower dimensional subspace with dimensionality reduction techniques, so to lower the storage space requirements, increase the learning speed and improve the predictive performance. Finally, in order to assess whether the model performance generalizes beyond the training set to new data, it is necessary to randomly divide the dataset into a training and a test set. The former is used to train the algorithm, while the latter is used to evaluate the final model.

4.2 Model Training and Selection

As stated by the No Free Lunch theorem in Wolpert (1996), each algorithm has its inherent biases and specific assumptions, and no single model has a superior performance across all possible scenarios. Therefore, it is essential to compare a handful of algorithms using a performance metric to select the best performing one. Moreover, since the test set is used only for the final model evaluation and not for the model selection, it is not possible to know which model performs better on the test set itself. To obviate this issue, cross-validation techniques are used, such as further splitting the training set into a training and a validation subset, to assess the generalization performance of the models. Finally, hyperparameter optimization techniques are used to optimize the performance of the model by fine-tuning the parameters of the algorithms.

4.3 Model Evaluation and Output Prediction

After having trained and selected the best model, its generalization error can be calculated by testing its performance on the unseen data contained in the test set. If the result is satisfactory, the model can then be used to make predictions on unseen data. It is important to notice that, although the parameters used in the previous procedures (such as feature scaling or dimensionality reduction) are obtained exclusively from the training set, the same parameters must be applied also to the validation and test sets, as well as to new data.

Notwithstanding the big impact of the Perceptron and the the Adaline models, many researchers started to lose interest in neural networks, due to the challenge of training a network with multiple layers. A major breakthrough came with rumelhart1988, who popularized the use of the backpropagation algorithm to train these neural networks efficiently. This and other advancements made in deep learning over the last two decades have paved the way for the current popularity of neural networks and, in particular, of deep architectures.

5 Models

5.1 GARCH

A GARCH model is used to model the time-varying volatility of a series of a financial asset returns. It assumes that the conditional distribution of the residual $\varepsilon_t = r_t - \mu_t$ (where μ_t is the conditional mean of r_t) is normal $N(0, \sigma_t^2)$ and that the time-varying volatility σ_t depends on past residuals $\{\varepsilon_{t-1}, \ldots, \varepsilon_{t-q}\}$ and past volatilities $\{\sigma_{t-1}, \ldots, \sigma_{t-p}\}$. The general form of a GARCH(p, q) model is:

$$r_t = \mu_t + \varepsilon_t \quad \text{where} \quad \varepsilon_t | \psi_{t-1} \sim N(0, \sigma_t^2) \tag{47}$$

$$\sigma_t^2 = \omega + \alpha_1 \varepsilon_{t-1}^2 + \cdots + \alpha_q \varepsilon_{t-q}^2 + \beta_1 \sigma_{t-1}^2 + \cdots + \beta_p \sigma_{t-p}^2 \tag{48}$$

where ψ_{t-1} denotes information available up to time $t-1$. The parameters $\omega, \alpha_i, \beta_i$ can be estimated with maximum likelihood.

Variations of the GARCH include: models that make different assumptions about the distribution of the error term ε_t, such as the GARCH-t model, which assumes a t-distribution ($\varepsilon_t = \sigma_t z_t, z_t | \psi_{t-1} \sim t(v)$) and accounts for the leptokurtosis; models that make different assumptions about the dynamics of the conditional mean μ_t, such as the AR-GARCH, which assumes it to follow an autoregressive process ($\mu_t = \gamma_0 + \gamma_1 r_{t-1} + \cdots + \gamma_s r_{t-s}$), or others that assume it to be constant ($\mu_t = \mu$); and models that make different assumptions about the dynamics of the variance term σ_t^2, such as EGARCH, GJR-GARCH, TGARCH, etc. GARCH models were initially designed to model and forecast conditional volatility but can be applied to predict conditional quantiles (such as VaR) because they fully describe the conditional distribution.

A similar type of models are stochastic volatility (SV) models, which are applied when volatility contains independent risk drivers. However, in continuous time, if the driver is a Brownian Motion, then the process is Markov and may fail to model serial dependencies in volatility. Another similar type of models are long-memory volatility (LMV) models, with drivers such as a fractional Brownian Motion or an Hawkes process. Finally, CAViaR is a type of model to estimate conditional quantiles separately for different levels of probability, rather than making assumptions on the full conditional distribution.

5.2 Traditional Quantile Regression

For a probability level $\tau \in (0, 1)$, the τ-quantile of a continuous distribution 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 $p(y|x)$ by means of a parametric function $q = f_\theta(x)$, 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:

$$L_\tau(y, q) = \begin{cases} \tau|y - q| & \text{if } y > q \\ (1 - \tau)|y - q| & \text{if } y \leq q \end{cases} \tag{49}$$

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

$$\min_\theta \mathbb{E}[L_\tau(y, f_\theta(x))] \tag{50}$$

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

$$\min_\theta \frac{1}{N} \sum_{i=1}^{N} L_\tau(y_i, f_\theta(x_i)) \tag{51}$$

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

$$\min_{\theta_1, \ldots, \theta_K} \frac{1}{K} \frac{1}{N} \sum_{k=1}^{K} \sum_{i=1}^{N} L_{\tau_k}(y_i, f_{\theta_k}(x_i)) \tag{52}$$

The problem with Equation 52 is that it may lead to quantile crossing, i.e. that for some x and $\tau_{k_j} < \tau_{k_i}$, it may happen that $f_{\theta_{k_j}}(x) > f_{\theta_{k_i}}(x)$. This is caused by the fact that θ_{k_j} and θ_{k_i} are estimated independently. The quantile crossing issue can be overcome by adding constraints on the monotonicity of the quantiles in the optimization or by post-processing, i.e. rearranging the estimated quantiles to be monotone. Furthermore, the number of parameters grows when estimating many K quantiles. Finally, the mapping from x to the conditional quantile has no interpretability.

5.3 Heavy Tailed Quantile Function

The probability density function (PDF), the cumulative distribution function (CDF) and the quantile function (QF) are three methods to describe a continuous distribution. Since in financial modeling it is important to define a parametric PDF that appropriately accounts for the

empirical behavior of the asset, this model defines a parametric quantile function that allows for leptokurtic, asymmetric and time-varying tails.

The idea for this 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 $\mu + \sigma Z_\tau$, where Z_τ is the τ-quantile of a standard normal distribution, when τ takes different values, their Q-Q plot forms a straight line. On the other hand, a Q-Q plot figuring an inverted S shape indicates that the distribution is heavy tailed.

This model constructs 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. More precisely, the up and down tails of the inverted S shape in the Q-Q plot are controlled by two parameters. The formulation of the heavy tail quantile function is as follows:

$$Q(\tau|\mu, \sigma, u, d) = \mu + \sigma Z_\tau \left(\frac{e^{uZ_\tau}}{A} + 1 \right) \left(\frac{e^{-dZ_\tau}}{A} + 1 \right) \tag{53}$$

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 tail 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.

5.4 Quantile Regression with HTQF

Differently from the GARCH model, which assumes a specific PDF of the conditional distribution $p(r_t|r_{t-1}, r_{t-2}, \dots)$, this paper parametrizes its quantile function as a HTQF, denoted $Q(\tau|\mu_t, \sigma_t, u_t, d_t)$, where μ_t, σ_t, u_t and d_t are the time varying parameters of the distribution. The conditional τ_k-quantile of r_t can be calculated by substituting τ_k into the HTQF:

$$q_k^t = Q(\tau_k|\mu_t, \sigma_t, u_t, d_t) \quad \text{for} \quad k = 1, \dots, K \tag{54}$$

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 past series to construct a feature vector sequence of length L and apply a LSTM unit to it:

$$x_1^t, \dots, x_L^t = \begin{bmatrix} r_{t-L} \\ (r_{t-L} - \bar{r}_t)^2 \\ (r_{t-L} - \bar{r}_t)^3 \\ (r_{t-L} - \bar{r}_t)^4 \end{bmatrix} \dots \begin{bmatrix} r_{t-1} \\ (r_{t-1} - \bar{r}_t)^2 \\ (r_{t-1} - \bar{r}_t)^3 \\ (r_{t-1} - \bar{r}_t)^4 \end{bmatrix} \tag{55}$$

where $\bar{r}_t = \frac{1}{L}\sum_{i=1}^{L} r_{t-i}$. This specification for the feature vector contains information for the first four central moments of the past L samples. 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 inputs $\{x_1^t, \dots, x_L^t\}$:

$$\big[\mu_t, \sigma_t, u_t, d_t\big]^\intercal = tanh(W^o h_t + b^o) \quad \text{where} \quad h_t = LSTM_\Theta(x_1^t, \dots, x_L^t) \tag{56}$$

where W^o, b^o are the output layer parameters, h_t is the last hidden state and Θ are the LSTM parameters. 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:

$$\min_{\Theta, W^o, b^o} \frac{1}{K}\frac{1}{T-L}\sum_{k=1}^{K}\sum_{t=L+1}^{T} L_{\tau_k}\big[r_t, Q(\tau_k|\mu_t, \sigma_t, u_t, d_t)\big] \tag{57}$$

The complete quantile regression model with HTQF and LSTM, denoted HTQF-LSTM, combines Equations 53, 55, 56, 57. After training, for new series $\{r_{t'}\}$, 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 $\hat{\Theta}, \hat{W^o}, \hat{b^o}$, where $\{u_{t'}\}$ and $\{d_{t'}\}$ represent how the tails behave in time. Conditional quantiles $q_k^{t'}$ can be predicted and the summed loss is calculated again to test the performance of the new series.

5.5 Discussion

The HTQF brings improvements over the GARCH approach as it is more suited to model the leptokurtic, asymmetric and time varying behavior of the tails. On the other hand, the GARCH assumes a particular specification of the PDF, such could be a skewed t-distribution, which must have a complicated analytical form to account for the varying behavior of the tails. This makes the model estimation very difficult. Moreover, unlike the GARCH model, the LSTM unit used

by the proposed model can learn non-linear dependences on past information. The empirical study compares the HTQF with the classical GARCH models.

Moreover, compared with the traditional quantile regression, the HTQF brings some improvements as it is a monotonically increasing function in Z_τ and τ, so it is not affected by quantile crossing; it always needs four parameters $(\mu_t, \sigma_t, u_t, d_t)$ to estimate the quantiles, regardless of their number K and it is interpretable. The empirical study also compares the HTQF with the traditional quantile regression, which is also coupled with a LSTM unit. The traditional quantile regression is defined by reformulating Equations 56 and 57 as follows:

$$q_k^t : \left[q_1^t, \ldots, q_K^t\right]^\intercal = tanh(W^o h_t + b^o) \tag{58}$$

$$\min_{\Theta, W^o, b^o} \frac{1}{K} \frac{1}{T-L} \sum_{k=1}^{K} \sum_{t=L+1}^{T} L_{\tau_k}(r_t, q_k^t) \tag{59}$$

For new time t' the predicted quantiles $q_1^{t'}, \ldots, q_K^{t'}$ are sorted to avoid the quantile crossing issue. This model is denoted TQR-LSTM.

Moreover, generally, the feature vector sequence $\{x_1^t, \ldots, x_L^t\}$ should contain all information that is related to the conditional distribution of r_t so that it can help to predict it. However, for comparison purposes with the GARCH model, this series is limited to contain past returns $\{r_{t-1}, r_{t-2}, \ldots\}$.

5.6 Model Specifications

Each time series is divided into three subsequent parts: the training set is 4/5 of the series and the validation and test sets are 1/10 each. All three sets are normalized to have a sample mean of zero and a variance of 1. The validation set is used to tune the hyper-parameters and to stop training to avoid overfitting.

The model presented in this paper, henceforth denoted HTQF-LSTM(L, H), has two hyper-parameters: the fixed length L of the past series of returns, on which the parameters μ_t, σ_t, u_t and d_t depend, and the dimension H of the hidden state of the LSTM unit. Also the traditional quantile regression has two hyperparameters, TQR-LSTM(L, H). Finally, the competing models, namely the GARCH and its variations (such as GARCH-t, EGARCH-t, GJR-GARCH-t, AR-EGARCH-t and AR-CJR-GARCH-t) have three hyper-parameters: the number of lags q for the error term, the number of lags p for the variance term and the number of lags s for the autoregressive process. The tuning of the hyper-parameters is done in the following sets: $L \in \{40, 60, 80, 100\}$, $H \in \{8, 16\}$ and $s, p, q \in \{1, 2, 3\}$.

Moreover, the constant $A = 4$ is chosen arbitrarily. The $K = 21$ quantiles belong to the set $\{0.01, 0.05, 0.10, \ldots, 0.90, 0.95, 0.99\}$. Finally, performance of the models is evaluated using the pinball loss function on the test set. Two different performances over two different τ sets are evaluated: one is the full τ set, the other is $\big[0.01, 0.05, 0.1\big]$, the quantiles of which are VaR representing downside risk.

6 Empirical Study

This section presents the dataset used in the empirical study; then, it outlines the model specifications adopted; finally, it presents the results obtained.

6.1 Dataset

The experiments are conducted on three types of times series: simulated data, daily asset returns (of stock indexes, exchange rates and treasury yields) and intraday 5-minute commodity futures returns [7]. More precisely, for daily asset returns, the data of maximum possible length is used for every time series; for intraday commodity futures return, the recent one year every 5-minute returns are used. Throughout the empirical study, simple returns are used ($r_t = P_t/P_{t-1} - 1$ where P_t is the price or yield).

6.2 Simulated Data

The purpose of the simulated dataset is to verify whether the model can learn the true temporal behavior of the conditional distribution of a given time series. The simulated time series is generated in a way similar to a GARCH-t model but the degrees of freedom v_t are allowed to be time varying. Starting from, $r_0 = 0$ and $\sigma_0 = 1$, 10,000 data points for the time series $\{r_t\}$, the scale parameter $\{\sigma_t\}$ and the tail parameter $\{v_t\}$ are generated according to:

$$v_t = max\{8 - 2\pi_t, 3\} \quad \text{where} \quad \pi_t = \sqrt{0.136 + 0.257 r_{t-1}^2 + 0.717 \pi_{t-1}^2} \tag{60}$$

$$\sigma_t = \sqrt{0.293 + 0.161 r_{t-1}^2 + 0.575 \sigma_{t-1}^2} \tag{61}$$

$$r_t = \sigma_t z_t \quad \text{where } z_t \text{ is sampled from } t(v_t) \tag{62}$$

From Figure 14 it is visible that the learned HTQF scale and tail parameters are strongly linearly correlated with the true ones, on both the training and the test set. This confirms that

[7] Overnight jumps are eliminated.

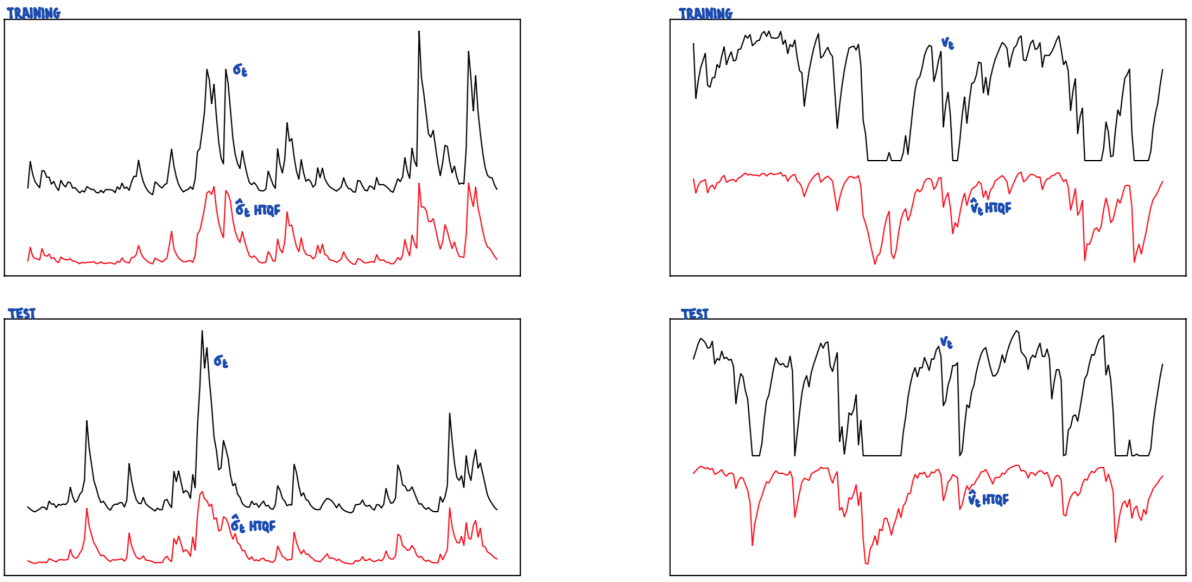


Figure 14: Comparison between true parameters and learnt ones.

the model has properly learned the temporal behavior of the conditional distribution of r_t. In fact, the linear correlation coefficients between the two lines in the four subplots are 0.8751, -0.8974, 0.9548 and -0.8808 respectively. Negative signs are due to the fact that the heavier the tail, the bigger $\hat{u}_t$ but the smaller v_t. After running linear regressions between them, we obtain R-squared of 0.7658, 0.8054, 0.9116, 0.7758. The other learned parameter $\hat{d}_t$ is similar to $\hat{u}_t$ but it is not shown as the t-distribution used to generate data is symmetric.

6.3 Real World

Stock indexes include S&P500, NASDAQ100, HSI, Nikkei225, DAX, FTSE100; exchange rates include USD to EUR/GBP/CHF/JPY/AUD; treasury yields include US 2, 10 and 30 years. TABLE1 reports the pinball loss function for every methods, as computed on the test sets of every asset return series. It shows that on most financial assets the HTQF-LSTM outperforms the other models. In parts (b) and (d) the performance is better than in (a) and (c), which indicates that the model describes the tails better. Note that the loss function is bounded by a positive number, so even a small decrease may actually represent a substantial improvement. (Other tests include Kupiec's unconditional coverage test, Christoffersen's independence test and the mixed conditional coverage test for backtesting the VaR forecasts of various models).

Commodity futures include contracts on steel rebar, natural rubber, soybean, cotton and sugar traded on the Chinese market. TABLE2 reports the pinball loss function for every methods, as computed on the test sets of every asset return series. Also in this case, HTQF-LSTM outperforms the other models for most of the assets.

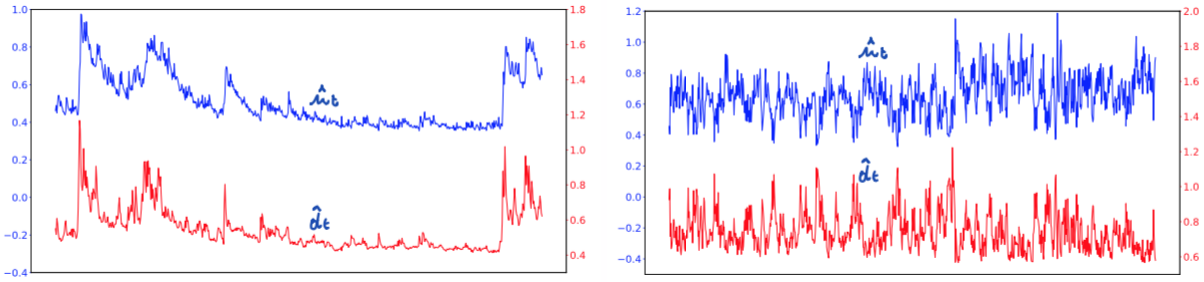


Figure 15: HTQF parameters.

In order to investigate the dynamics captured by the HTQF-LSTM model, Figure 15 plots the HTQF parameters $\hat{u}_t$ and $\hat{d}_t$ on the test set of S&P500: similar patterns with clustering and spikes are visible in both lines but they differ in details. Figure 15 also plots the HTQF parameters on the test set of steel rebar 5-minute data: high frequency data also have time varying leptokurtic tails. The different tail dynamics with the S&P may be attributed to the different time scales of the two time series.

7 Conclusion

The paper proposed a parametric HTQF to model the asymmetric and leptokurtic conditional distribution of financial asset returns. The dependence of the HTQF parameters on past information is modeled by a LSTM unit. The pinball loss function between the observation and conditional quantiles makes the learning of LSTM parameters be in a quantile regression framework, which overcomes the disadvantages of traditional quantile regressions.

Conditional quantiles and VaRs can be predicted with better accuracy. The plotting of HTQF parameters shows the dynamic tail behavior of financial asset returns, some of which show spikes and clustering, with differences between the two tails.

8 Data

In this section, I describe the database used in the study, the data preparation process applied to obtain a dataset suitable to the ensuing analysis, and the sample of stocks adopted.

8.1 Database Description

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 [8] 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, I use 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 physical 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.

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, root and suffix of the ticker, sale condition, number of shares traded, dollar price per share,

[8] 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.

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.

Table 1: Sample Trade Records for Apple

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

8.2 Data Preparation

Ultra high-frequency data on financial markets require the application of methods to remove outliers, which do not correspond to plausible market activity, and to aggregate data into time series suitable for the analysis. The structure of tick data is dependent on several factors, such as the regulatory framework, the procedures of the institution that collects and produces the information, and the technological advancements. Furthermore, the series of tick data might contain errors, which are not identifies as such by the data provider and might be affected by unusual market conditions. When studying ultra-high-frequency based measures of volatility, these data cleaning and data management processes are particularly relevant, as they correct the impact of outliers, which may otherwise signal fictitious movements in the market price. Nonetheless, these procedure involve subjective decisions, which may have an impact on the final results of the analysis.

The data preparation starts with the data cleaning process. As a first step, I keep only regular trades, which have not been corrected, changed or marked as cancel or error, i.e. all records with a correction indicator field equal to 00. I remove all original trades, which have later been corrected (01), marked as erroneous (07) or cancelled (08) and the corresponding correction (12), error (11) and cancel (10) records. As a second step, I keep only trades with a regular timing, i.e. all records with a sale condition equal to @, A-F, H-K, M-O, Q-S, V, W, Y or 1-9. I remove 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). As a third step, I 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, both volumes and quotes cannot be used because their plausibility cannot be assessed beyond the plausibility of the corresponding price and because it is very difficult to match trades and quotes with precision. Therefore, I adopted a variation of the heuristic procedure outlined by brownlees2006. This identifies ouliers by calculating their relative distance from a

neighborhood of closest valid observations.

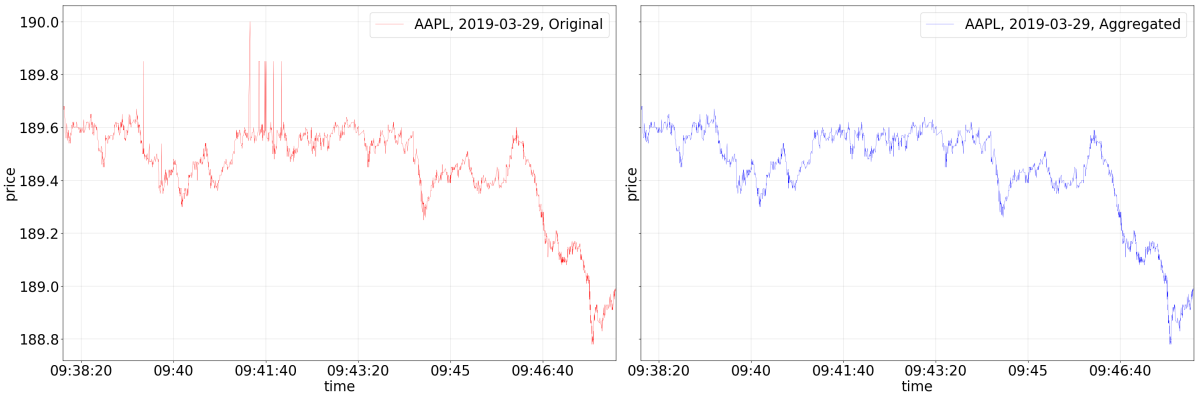
Let $\{p_j^i\}_{j=1}^N$ be the time-ordered tick-by-tick price series for stock i. The rule to identify outliers is:

$$|p_j^i - \bar{p}_j^i(k)| < 3s_j^i(k) + \gamma = \begin{cases} \text{observation } p_j^i \text{ is kept} \\ \text{observation } p_j^i \text{ is removed} \end{cases}$$

where $\bar{p}_j^i(k)$ and $s_j^i(k)$ are the δ-trimmed sample mean and sample standard deviation of a neighborhood of k observations around p_j^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. brownlees2006 argue that the higher 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, I set the trimming parameter constant to $\delta = 10\%$ and I identify the outliers using the heuristic procedure outlined above, for all possible combinations of $k \in \{41, 61, ..., 121\}$ and $\gamma \in \{0.02, 0.04, ..., 0.08\}$. For each stock, I then select the combination of k and γ that identifies the highest 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, select the combination of k and γ that identifies the least number of outliers or that yields the highest average distance between the observations identified as outliers and the corresponding trimmed sample mean. Moreover, this rule allows to automate the parameter selection procedure and removes the need for a visual inspection of the plot.

Following these three steps, the price series still shows an undesired feature, namely the presence of transaction executed at the same time but at different prices. This is explained by the fact that securities can trade simultaneously on different exchanges and by reporting errors. Therefore, as a fourth step, I aggregated blocks of simultaneous observations calculating the median value of the prices and the sum of the corresponding sizes.

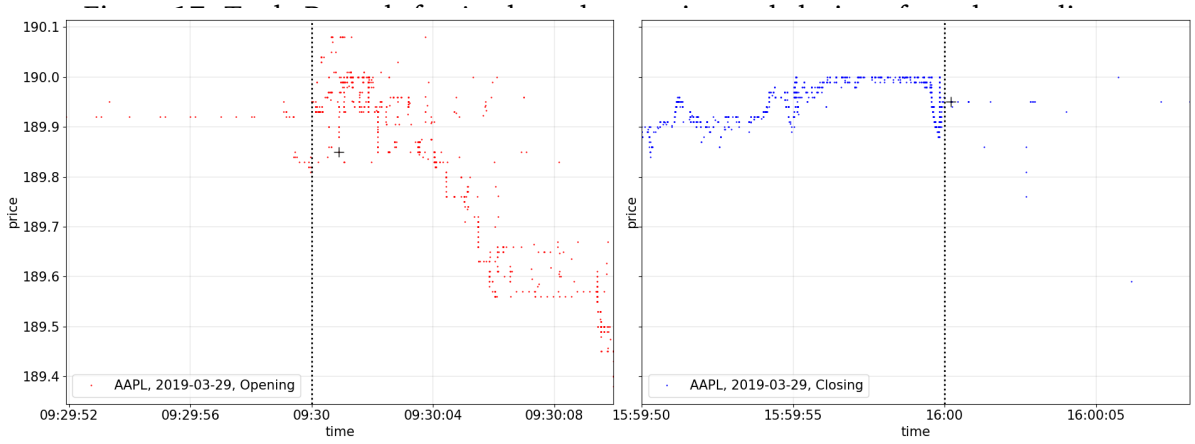
Figure 8.2 shows 10 minutes of trade records for Apple, before and after the four-step cleaning process just outlined.



8.3 Sample

The sample adopted in this study is made up of the components of the NASDAQ-100 Index, which includes 100 of the largest domestic and international non-financial securities listed on the Nasdaq Stock Market. For the companies that are included with two share classes (namely, Alphabet Inc, Fox Corporation, Liberty Global plc), only the share class with voting rights is considered. The dates considered are from the 01/01/2018 to 12/31/2018.

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. As visible in Figure 8.3, which shows 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 black cross. 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, even though other data fields are helpful in the identification of the opening and closing trades, these are not always reported correctly. To obviate these issues and to ensure that the full extent of trading prices is captured, I choose to consider a fixed time range from 9.30 a.m. to 16.05 p.m. for each stock.



primary listing on NASDAQ (`ex = T`)

FILTERS: -Common stocks -Primary listing on NYSE

TODO: -The addition of dividens and/or split/adjustment information

Sequence number: Market Data System sequence number. (Applies only to NYSE trades. It will contain a zero if the trade is not a NYSE trade or if the sequence number is intermediate.) Correction: Since corrections of corrections are possible, only the original trade and the final correction is shown; no interim corrections are shown.

– Good trades: - 0: regular trade that was not corrected, changed or signified as cancel or error; - 1: original trade which was later corrected. This record contains the original time and the corrected data for the trade - 2: Symbol correction (out of time sequence)

– Original trade records: - 7: Trade cancelled due to error - 8: Trade cancelled - 9: Trade cancelled due to symbol correction

– Correction instructions: - 10: Cancel record (associated with 8) - 11: Error record (associated with 7) - 12: correction record (associated with 1; contains corrected time and original data).

A detailed explanation of the programming and computing setup required to enable the access routine to retrieve and manipulate the trade records is available in the online appendix.

Valid exchanges: NYSE (N), NASD (T), AMEX (A), Boston (B), Pacific (P), Philadelphia (X), CBOE (W), Midwest (M), Cincinnati (C)

Valid security types: common (C), preferred (P), warrants (W), rights (R), other (O).

- When there is an imbalance in Market-On-Close buy or sell orders, the imbalance is executed at the close of trading against the offer or bid on the NYSE, thereby setting the closing price. The remaining buy and sell orders are then matched at the closing price set by the imbalance trade.

9 Nando de Freitas

Machine learning deals with the problem of extracting features from data so as to solve many different predictive tasks. Common applications of machine learning include forecasting, imputing missing data, detecting anomalies, classifying, ranking, summarizing and decision making. Machine learning is particularly useful when no human expertise is available, when human are not able to explain their expertise, when the solution keeps changing over time, when the solution needs to be adapted to particular cases or when the problem is too vast for human reasoning capabilities.

A Weight Updates in the Adaline Model

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

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

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

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

So that the weight update becomes:

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

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:

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

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:

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

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:

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

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:

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

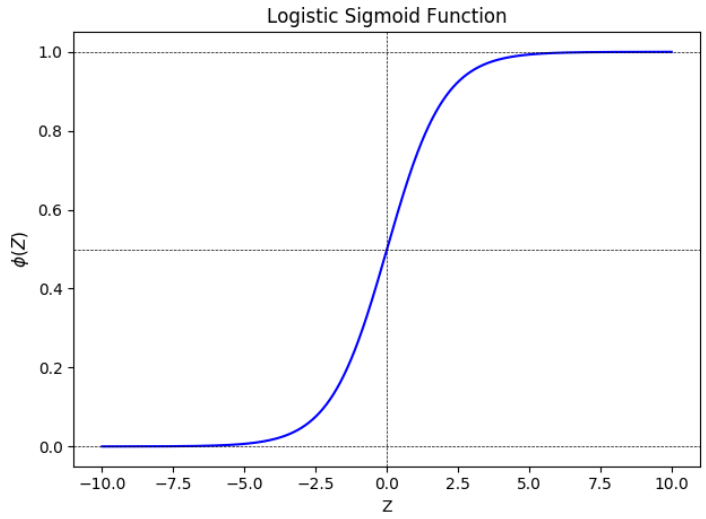
which can be explicited as follows:

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

which is formalized into the logistic sigmoid function:

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

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.



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:

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

The likelihood function of the samples is then given by:

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

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

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

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

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

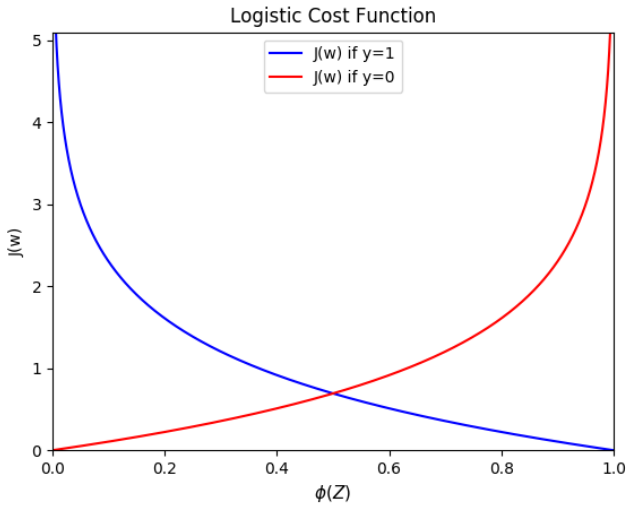
To clarify the meaning of this cost function, let's evaluate the cost of a single training sample:

$$J(\boldsymbol{w}) = -y_i \; log\Big(\phi(z_i)\Big) - (1 - y_i) \; log\Big(1 - \phi(z_i)\Big)$$

from which it is clear that:

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

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.



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:

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

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

$$\frac{\partial J}{\partial w_j} = \frac{\partial}{\partial w_j} \sum_i \Big[- \; y_i \; log\Big(\phi(z_i)\Big) - (1 - y_i) \; log\Big(1 - \phi(z_i)\Big)\Big]$$

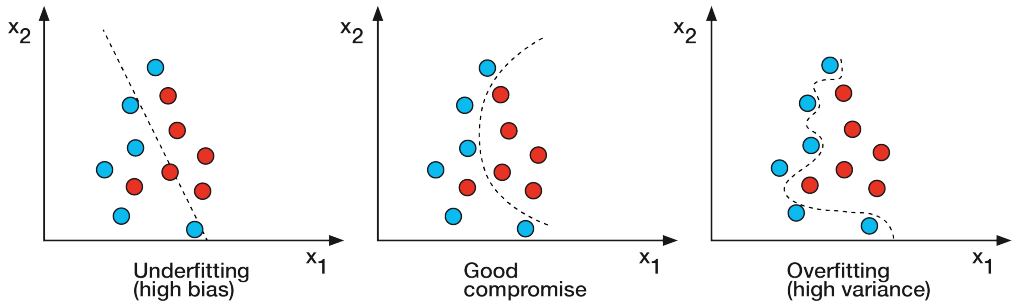
$$
\begin{aligned}
&= -\sum_i \left\{ \left[y_i \frac{\partial log\Big(\phi(z_i)\Big)}{\partial \phi(z_i)} + (1-y_i) \frac{\partial log\Big(1-\phi(z_i)\Big)}{\partial \phi(z_i)} \right] \frac{\partial \phi(z_i)}{\partial z_i} \frac{\partial z_i}{\partial w_i} \right\} \\
&= -\sum_i \left\{ \left[\frac{y_i}{\phi(z_i)} - \frac{1-y_i}{1-\phi(z_i)} \right] \frac{e^{-z_i}}{(1+e^{-z_i})^2} x_i^j \right\} \\
&= -\sum_i \left\{ \left[\frac{y_i}{\phi(z_i)} - \frac{1-y_i}{1-\phi(z_i)} \right] \frac{1}{1+e^{-z_i}} \left(1 - \frac{1}{1+e^{-z_i}} \right) x_i^j \right\} \\
&= -\sum_i \left\{ \left[\frac{y_i}{\phi(z_i)} - \frac{1-y_i}{1-\phi(z_i)} \right] \phi(z_i) \Big(1 - \phi(z_i) \Big) x_i^j \right\} \\
&= -\sum_i \left\{ \Big[y_i - y_i \phi(z_i) - \phi(z_i) + y_i \phi(z_i) \Big] x_i^j \right\} \\
&= -\sum_i \left\{ \Big[y_i - \phi(z_i) \Big] x_i^j \right\}
\end{aligned}
$$

So that the weight update becomes:

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

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.



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, which adds additional information as follows:

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

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 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:

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

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.

G Multilayer Perceptron Simplified

For pedagogical reasons, let's assume that the algorithm processes one training sample at the time. Therefore, let $\boldsymbol{x}_i$ be the vector of features for sample i:

$$\boldsymbol{x}_i = \begin{pmatrix} x_i^1 & \dots & x_i^m \end{pmatrix}$$

Let's now describe the architecture of the network. The activation units are denoted as $a_{(in)}^j$, $a_{(h)}^j$, $a_{(out)}^j$ for the input, hidden and output layers respectively, where $a_{(in)}^0$, $a_{(h)}^0$ are the bias units, set to 1. Since each activation unit $a_{(l)}^j$ in a layer (l) is connected to each unit $a_{(l+1)}^k$ in the next layer $(l+1)$ via 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 $\boldsymbol{W}_{(out)} \in \mathbb{R}^{(d+1\ x\ t)}$:

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

Finally, to perform multi-class classification via the one-versus-all technique (and the one-hot representation), let the number of units in the output layer be equal to the number of classes (this is explained in greater detail in Section 3.10).

We can now define the Multilayer Perceptron learning rule as follows. For each epoch and for each training sample $\boldsymbol{x}_i$:

1. Activate the units in the input layer.
2. Forward propagate the patterns of the training sample 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 bias unit:

$$\boldsymbol{a}_{(in)} = \begin{pmatrix} a_{(in)}^0 & a_{(in)}^1 & \dots & a_{(in)}^m \end{pmatrix} = \begin{pmatrix} 1 & x_i^1 & \dots & x_i^m \end{pmatrix}$$

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:

$$\boldsymbol{z}_{(h)} = \boldsymbol{a}_{(in)} \boldsymbol{W}_{(h)} = \begin{pmatrix} z_{(h)}^1 & \dots & z_{(h)}^d \end{pmatrix}$$

$$\boldsymbol{a}_{(h)} = \phi(\boldsymbol{z}_{(h)}) = \begin{pmatrix} \phi(z_{(h)}^1) & \dots & \phi(z_{(h)}^d) \end{pmatrix} \quad \text{where} \quad \phi(Z) = \frac{1}{1 + e^{-Z}}$$

H Backpropagation Algorithm

The backpropagation algorithm 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, than 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:

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

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.

I Derivative of the Activation Function in the Backpropagation Algorithm

Then the error term of the hidden layer is calculated as:

$$\boldsymbol{\delta}^{h} = \boldsymbol{\delta}^{out}(\boldsymbol{W}^{out})^{\top} \odot \frac{\partial \phi(\boldsymbol{Z}^{h})}{\partial \boldsymbol{Z}^{h}}$$

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

$$\begin{aligned}\frac{\partial \phi(Z)}{\partial z} &= \frac{\partial}{\partial z}\left(\frac{1}{1+e^{-z}}\right) \\ &= \frac{e^{-z}}{1+e^{-z}}\end{aligned}$$

$$
\begin{aligned}
&= \frac{1+e^{-z}}{(1+e^{-z})^2} - \left(\frac{1}{1+e^{-z}}\right)^2 \\
&= \frac{1}{1+e^{-z}} - \left(\frac{1}{1+e^{-z}}\right)^2 \\
&= \phi(z) - \big(\phi(z)\big)^2 \\
&= \phi(z)\big(1-\phi(z)\big)
\end{aligned}
$$

so that the error term of the hidden layer can be reformulated as follows:

$$
\begin{aligned}
\boldsymbol{\delta}^h &= \boldsymbol{\delta}^{out}(\boldsymbol{W}^{out})^\top \odot \Big(\phi(\boldsymbol{Z}^h) \odot \big(1-\phi(\boldsymbol{Z}^h)\big)\Big) \\
&= \boldsymbol{\delta}^{out}(\boldsymbol{W}^{out})^\top \odot \Big(\boldsymbol{A}^h \odot \big(1-\boldsymbol{A}^h\big)\Big)
\end{aligned}
$$

Now, the derivation of the cost function is as follows:

$$
\frac{\partial}{\partial w^{out}_{i,j''}} J(\boldsymbol{W}) = \phi(z^h_{j''})\delta^{out}_i
$$

$$
\frac{\partial}{\partial w^{h}_{i,j'}} J(\boldsymbol{W}) = \phi(z^{in}_{j'})\delta^{h}_i
$$

Now we need to accumulate the partial derivative of every node in each layer and the error of the node in the next layer, where $\Delta^l_{i,j}$ is calculated for every sample in the training set:

$$
\boldsymbol{\Delta}^h = \boldsymbol{\Delta}^h + \big(\phi(\boldsymbol{z}^{in})\big)^\top \boldsymbol{\delta}^h
$$

$$
\boldsymbol{\Delta}^{out} = \boldsymbol{\Delta}^{out} + \big(\phi(\boldsymbol{z}^{h})\big)^\top \boldsymbol{\delta}^{out}
$$

Now the regularization term can be added (except for the bias term):

$$
\Delta^l := \Delta^l + \lambda^l
$$

and the weight is updated by taking an opposite step towards the gradient for each layer l:

$$
\boldsymbol{W}^l := \boldsymbol{W}^l - \eta\Delta^l
$$

J Programming and Computing Setup

This study has been implemented using the programming language *Python 3.7* and the integrated development environment *PyCharm CE*. A public GitHub repository [9] is intended to provide all programming and computing setup to fully replicate the study [10]: it contains the full code and a detailed explanation of the setup required to run Python locally on a Mac (Apple Inc.) device and on the WRDS cloud.

[9] available at `github.com/nicoloceneda/Thesis`
[10] a valid WRDS account is needed

References

[1] 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.

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

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

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

[5] 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.

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

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

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

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

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

[11] Sebastian Raschka. *Introduction to Artificial Neural Networks and Deep Learning*. 2016.

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

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

[14] 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.

[15] NASDAQ Market Technology. *The UTP Plan Trade Data Feed-SM (UTDF-SM)*. Version 14.4. Nov. 2015.

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

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

[18] 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.