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

Nicolo Ceneda
Master of Arts in Banking and Finance
University of St Gallen

Supervisor: Prof. Dr. Christoph Aymanns
Co-supervisor: Prof. Dr. Manuel Ammann

May 8, 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 assuming the probability density function and letting 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 asymmetric 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 quantiles without making assumptions on the probability density function. Indeed, besides the probability density function, quantiles are another way to describe the shape of a distribution: modeling the conditional quantiles for a finite set of probabilities is almost equivalent to modeling the conditional distribution, i.e. estimating the conditional mean, volatility, skewness and kurtosis. 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 overcome these problems, this paper forecasts the conditional quantiles of a distribution 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 a specific probability density function, which may introduce tractability and ill-posedness issues. More precisely, this paper uses Yan et al. (2018)'s 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 increasing availability of data at the highest possible frequency (tick-by-tick) has paved the way for significant advances in the field of quantitative finance. This paper contributes to the literature of time series analysis by replicating the study conducted by Yan et al. (2018), but focusing on tick data in the U.S stock market.

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 financial asset returns; implementing a LSTM unit that can be trained in a quantile regression framework to learn the time varying behavior of the tails 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 $f(\mathbf{x}, \mathbf{w}) = x\_1w\_1 + \cdots + x\_nw\_n$ 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.

$$\text{Samples}\Big\downarrow \quad \boldsymbol{X}: \begin{array}{c|cccc|} & \multicolumn{4}{c}{\xrightarrow{\text{Features}}} \\ & \boldsymbol{x}^1 & \boldsymbol{x}^2 & \dots & \boldsymbol{x}^m \\ \boldsymbol{x}_1 & x_1^1 & x_1^2 & \dots & x_1^m \\ \boldsymbol{x}_2 & x_2^1 & x_2^2 & \dots & x_2^m \\ \dots & \dots & \dots & \dots & \dots \\ \boldsymbol{x}_n & x_n^1 & x_n^2 & \dots & x_n^m \end{array} \qquad \boldsymbol{y}: \begin{pmatrix} y_1 \\ y_2 \\ \dots \\ y_n \end{pmatrix}$$

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.

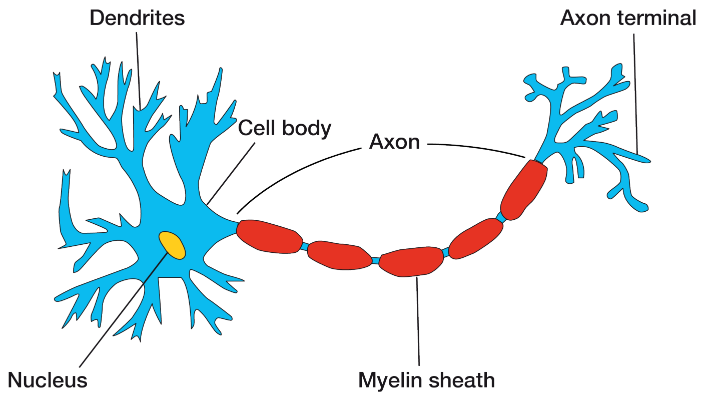


Figure 1: Neuron

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:

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

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.

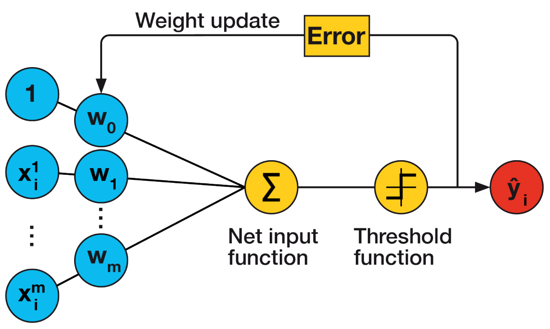


Figure 2: Perceptron learning rule

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.

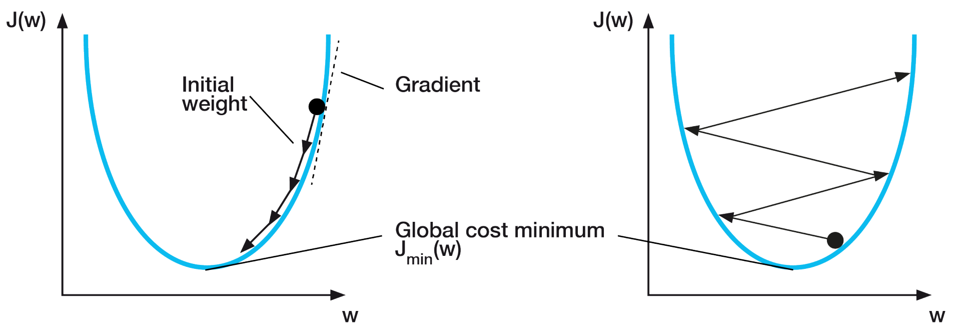


Figure 3: Gradient descent algorithm and overshooting

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.

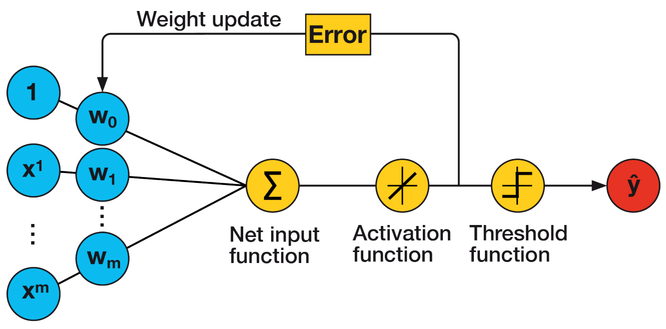


Figure 4: Adaline learning rule

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.

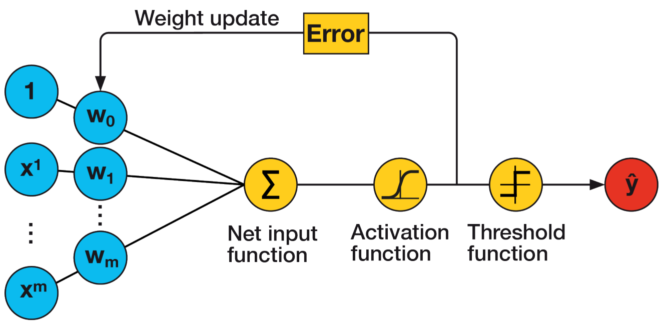


Figure 5: Logistic Regression learning rule

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 Multilayer Perceptron

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

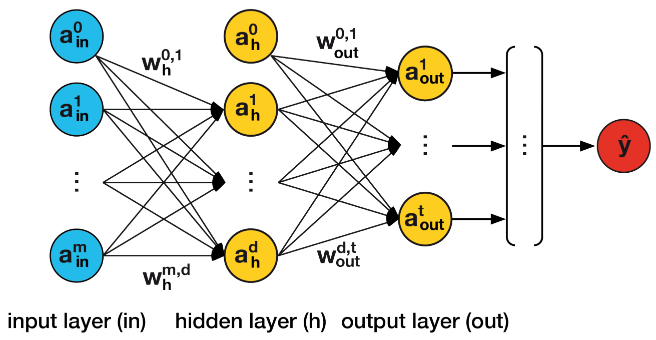


Figure 6: Multilayer Perceptron

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

$$\boldsymbol{X}^{mb} = \begin{pmatrix} x_1^1 & \dots & x_1^m \\ \dots & \dots & \dots \\ x_{bs}^1 & \dots & x_{bs}^m \end{pmatrix} \tag{23}$$

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

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

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

a weight coefficient $w^{j,k}_{(l+1)}$, 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 $\boldsymbol{W}_{(out)} \in \mathbb{R}^{(d+1\ x\ t)}$:

$$\boldsymbol{W}_{(h)} = \begin{pmatrix} w^{0,1}_{(h)} & \dots & w^{0,d}_{(h)} \\ \dots & \dots & \dots \\ w^{m,1}_{(h)} & \dots & w^{m,d}_{(h)} \end{pmatrix} \qquad \boldsymbol{W}_{(out)} = \begin{pmatrix} w^{0,1}_{(out)} & \dots & w^{0,t}_{(out)} \\ \dots & \dots & \dots \\ w^{d,1}_{(out)} & \dots & w^{d,t}_{(out)} \end{pmatrix} \tag{24}$$

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

$$0 = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \qquad 1 = \begin{pmatrix} 0 \\ 1 \\ 0 \end{pmatrix} \qquad 2 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{25}$$

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

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

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

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

$$\boldsymbol{A}_{(in)} = \begin{pmatrix} \boldsymbol{a}^0_{(in)} & \boldsymbol{a}^1_{(in)} & \dots & \boldsymbol{a}^m_{(in)} \end{pmatrix} = \begin{pmatrix} \boldsymbol{1} & \boldsymbol{x}^1 & \dots & \boldsymbol{x}^m \end{pmatrix} \tag{26}$$

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

$$\boldsymbol{Z}_{(h)} = \boldsymbol{A}_{(in)}\boldsymbol{W}_{(h)} = \begin{pmatrix} z^1_{(h),1} & \dots & z^d_{(h),1} \\ \dots & \dots & \dots \\ z^1_{(h),bs} & \dots & z^d_{(h),bs} \end{pmatrix} \tag{27}$$

$$\boldsymbol{A}_{(h)} = \begin{pmatrix} \boldsymbol{a}^1_{(h)} & \dots & \boldsymbol{a}^d_{(h)} \end{pmatrix} = \phi(\boldsymbol{Z}_{(h)}) = \begin{pmatrix} \phi(\boldsymbol{z}^1_{(h)}) & \dots & \phi(\boldsymbol{z}^d_{(h)}) \end{pmatrix} \tag{28}$$

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

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

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

$$\boldsymbol{A}_{(h)} := \begin{pmatrix} \boldsymbol{a}^0_{(h)} & \boldsymbol{a}^1_{(h)} & \dots & \boldsymbol{a}^d_{(h)} \end{pmatrix} = \begin{pmatrix} \boldsymbol{1} & \phi(\boldsymbol{z}^1_{(h)}) & \dots & \phi(\boldsymbol{z}^d_{(h)}) \end{pmatrix} \tag{29}$$

and conduct similar calculations for the output layer:

$$\boldsymbol{Z}_{(out)} = \boldsymbol{A}_{(h)}\boldsymbol{W}_{(out)} = \begin{pmatrix} z^1_{(out),1} & \dots & z^t_{(out),1} \\ \dots & \dots & \dots \\ z^1_{(out),bs} & \dots & z^t_{(out),bs} \end{pmatrix} \tag{30}$$

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

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

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

$$J(\boldsymbol{W}) = \sum_{i=1}^{n}\sum_{j=1}^{t}\left[-y^j_{(oh),i}\,log(a^j_{(out),i}) - (1-y^j_{(oh),i})\,log(1-a^j_{(out),i})\right] + \frac{\lambda}{2}\sum_{l+1\in\{h,out\}}\sum_{j=1}^{u_l}\sum_{k=1}^{u_l+1}\left(w^{j,k}_{(l+1)}\right)^2 \tag{32}$$

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

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

$$\boldsymbol{\Delta}_{(out)} = \boldsymbol{A}_{(out)} - \boldsymbol{Y}_{(oh)} \tag{33}$$

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

$$\boldsymbol{\Delta}_{(h)} = \boldsymbol{\Delta}_{(out)} \big(\boldsymbol{W}_{(out)}\big)^{\top} \odot \big(\boldsymbol{A}_{(h)} \odot (1 - \boldsymbol{A}_{(h)})\big) \tag{34}$$

where $\boldsymbol{\Delta}_{(out)} \in \mathbb{R}^{(bs\ x\ t)}$, $\boldsymbol{W}_{(out)} \in \mathbb{R}^{(d\ x\ t)}$, $\boldsymbol{A}_{(h)} \in \mathbb{R}^{(bs\ x\ d)}$ and $\odot$ denotes the element wise multiplication. After calculating the error matrices, the weights can be updated. Appendix G explains how to derive Equation 34.

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

$$\frac{\partial}{\partial \boldsymbol{W}_{(out)}} J(\boldsymbol{W}) = \big(\boldsymbol{A}_{(h)}\big)^{\top} \boldsymbol{\Delta}_{(out)} \tag{35}$$

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

The weight updates are now as follows:

$$\boldsymbol{W}_{(out)} := \boldsymbol{W}_{(out)} - \eta \Delta \boldsymbol{W}_{(out)} \quad \text{where} \quad \Delta \boldsymbol{W}_{(out)} = \big(\boldsymbol{A}_{(h)}\big)^{\top} \boldsymbol{\Delta}_{(out)} + \lambda \boldsymbol{W}_{(out)} \tag{37}$$

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

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

3.8 Recurrent Neural Network

3.8.1 Sequential Data

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

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

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

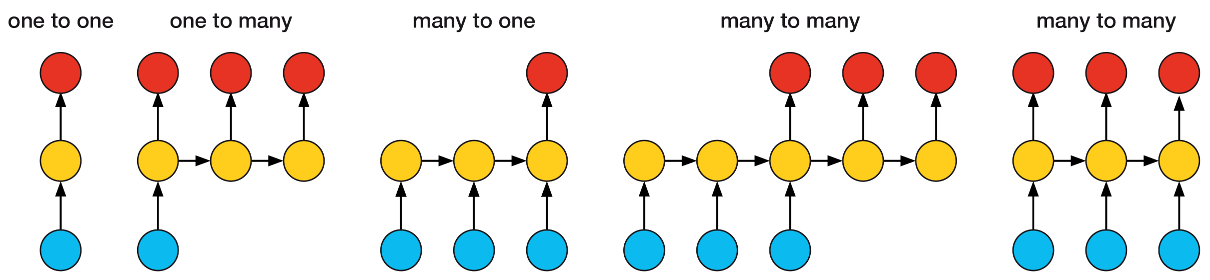


Figure 7: Types of relationships between input and output data

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

3.8.2 Memory of Sequential Data

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

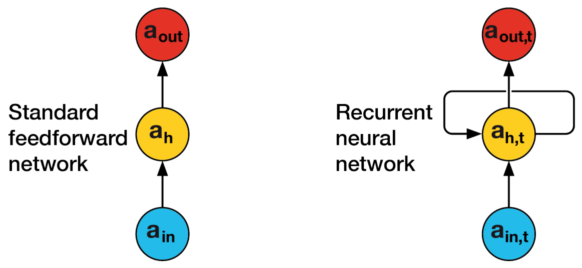


Figure 8: RNNs contain loops

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

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

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

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

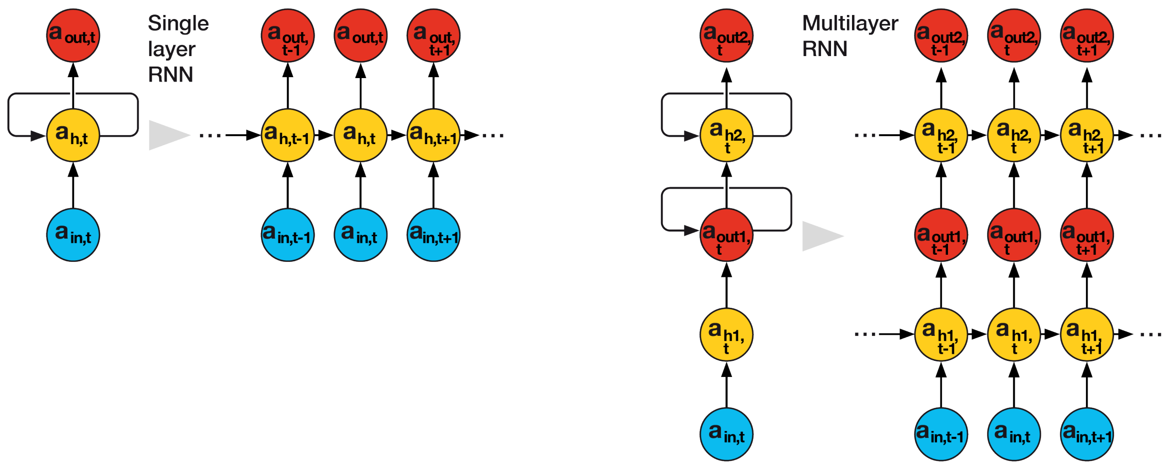


Figure 9: Compact and unrolled representations of a single- and a two-layer RNN

3.8.3 Architecture of a RNN

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

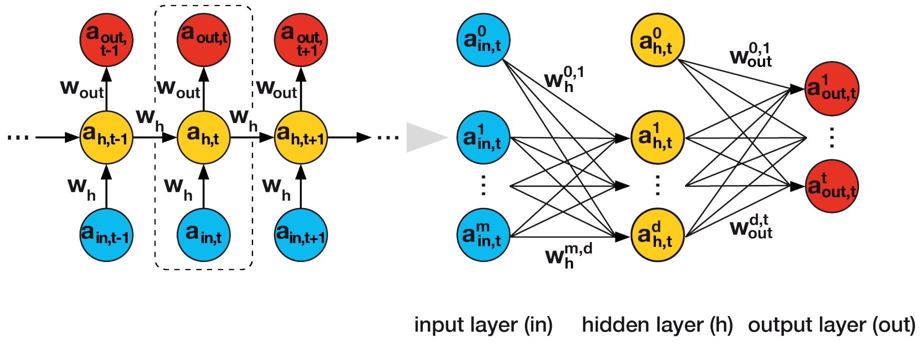


Figure 10: Unrolled and detailed representation of a single-layer RNN

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

$$\boldsymbol{W}_{(h)} = \begin{pmatrix} w^{0,1}_{(h)} & \dots & w^{0,d}_{(h)} \\ \dots & \dots & \dots \\ w^{m,1}_{(h)} & \dots & w^{m,d}_{(h)} \end{pmatrix} \qquad \boldsymbol{W}_{(out)} = \begin{pmatrix} w^{0,1}_{(out)} & \dots & w^{0,t}_{(out)} \\ \dots & \dots & \dots \\ w^{d,1}_{(out)} & \dots & w^{d,t}_{(out)} \end{pmatrix} \tag{39}$$

$$\boldsymbol{W}_{(hh)} = \begin{pmatrix} w^{1,1}_{(hh)} & \dots & w^{1,d}_{(hh)} \\ \dots & \dots & \dots \\ w^{d,1}_{(hh)} & \dots & w^{d,d}_{(hh)} \end{pmatrix} \tag{40}$$

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

$$\boldsymbol{a}_{(in),t} = \begin{pmatrix} a^0_{(in),t} & a^1_{(in),t} & \dots & a^m_{(in),t} \end{pmatrix} = \begin{pmatrix} 1 & x^1_t & \dots & x^m_t \end{pmatrix} \tag{41}$$

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

$$\boldsymbol{z}_{(h),t} = \boldsymbol{a}_{(in),t}\boldsymbol{W}_{(h)} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{(hh)} = \begin{pmatrix} z^1_{(h),t} & \dots & z^d_{(h),t} \end{pmatrix} \tag{42}$$

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

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

$$\boldsymbol{a}_{(h),t} := \begin{pmatrix} a^0_{(h),t} & a^1_{(h),t} & \dots & a^d_{(h),t} \end{pmatrix} = \begin{pmatrix} 1 & \phi_{(h)}(z^1_{(h),t}) & \dots & \phi_{(h)}(z^d_{(h),t}) \end{pmatrix} \tag{44}$$

and conduct similar calculations for the output layer:

$$\boldsymbol{z}_{(out),t} = \boldsymbol{a}_{(h),t}\boldsymbol{W}_{(out)} = \begin{pmatrix} z^1_{(out),t} & \dots & z^t_{(h),t} \end{pmatrix} \tag{45}$$

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

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

3.8.4 BPTT and Long-Range Dependencies

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

$$L = \sum_{t=1}^{s} L_t \tag{47}$$

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

$$\frac{\partial L_t}{\partial \boldsymbol{W}_{(hh)}} = \frac{\partial L_t}{\partial \boldsymbol{a}_{(out),t}} \frac{\partial \boldsymbol{a}_{(out),t}}{\partial \boldsymbol{a}_{(h),t}} \left(\sum_{k=1}^{t} \frac{\partial \boldsymbol{a}_{(h),t}}{\partial \boldsymbol{a}_{(h),k}} \frac{\partial \boldsymbol{a}_{(h),k}}{\partial \boldsymbol{W}_{(hh)}} \right) \quad \text{where} \quad \frac{\partial \boldsymbol{a}_{(h),t}}{\partial \boldsymbol{a}_{(h),k}} = \prod_{i=k+1}^{t} \frac{\partial \boldsymbol{a}_{(h),i}}{\partial \boldsymbol{a}_{(h),i-1}} \tag{48}$$

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

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

3.8.5 Long-Short Term Memory Networks

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

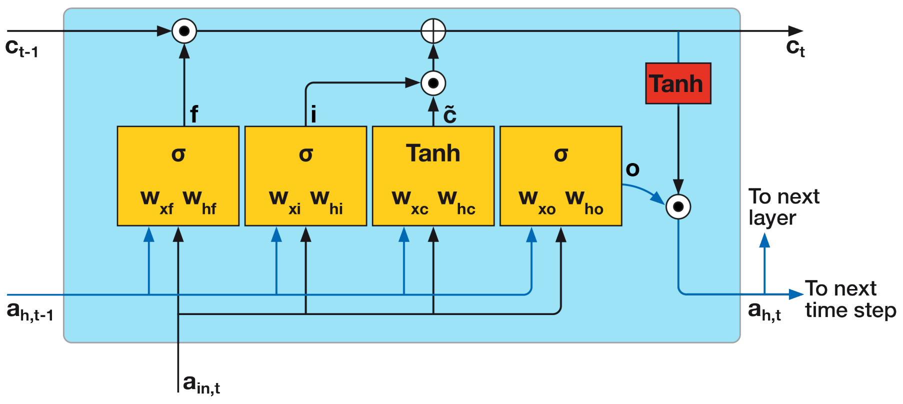


Figure 11: Internal structure of the memory cell of a LSTM

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

In the figure, $\odot$ indicates the element-wise multiplication, and $\oplus$ indicates the element-wise addition. Moreover, $\boldsymbol{a}_{(h),t-1}$ indicates the activation units in the hidden layer at time step $t-1$ and $\boldsymbol{a}_{(in),t}$ indicates the activation units in the input layer at time step t, i.e. the vector of input data. Four boxes, which are indicated by an activation function (either the sigmoid function σ or the hyperbolic tangent function $tanh$) and some weights, apply a linear combination by performing vector-matrix multiplications on their inputs $\boldsymbol{a}_{(h),t-1}$ and $\boldsymbol{a}_{(in),t}$. The gates controlling the flow of information are the three computation units with the sigmoid activation function and whose output units are passed through $\odot$, whereby the sigmoid layer outputs numbers between 0 and 1 to define how much of each component should be let through (0 being nothing and 1 being all). More precisely:

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

$$\boldsymbol{f}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xf} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hf}\big) \tag{49}$$

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

$$\boldsymbol{i}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xi} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hi}\big) \tag{50}$$

$$\tilde{\boldsymbol{c}}_t = \tanh\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xc} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hc}\big) \tag{51}$$

$$\boldsymbol{c}_t = \big(\boldsymbol{f}_t \odot \boldsymbol{c}_{t-1}\big) \oplus \big(\boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t\big) \tag{52}$$

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

$$\boldsymbol{o}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xo} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{ho}\big) \tag{53}$$

$$\boldsymbol{a}_{(h),t} = \boldsymbol{o}_t \odot \tanh\big(\boldsymbol{c}_t\big) \tag{54}$$

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

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

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. By comparing the training and validation accuracy during the training phase, it is possible to assess the network performance given the chosen architecture and hyperparameters. Since training an algorithm can be computationally expensive, it is optimal to revise its architecture and/or hyperparameters if it displays a tendency to over-fitting (training accuracy increasingly greater than validation accuracy) or under-fitting during the training phase (validation accuracy increasingly greater than training accuracy). 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 generalized autoregressive conditional heteroscedasticity (GARCH) model is a dynamic model that accounts for the conditional heteroscedasticity in an innovation process and is appropriate when positive and negative shocks of equal magnitude contribute equally to the volatility. This volatility clustering occurs when an innovation process does not have significant autocorrelation, but its variance changes over time. A GARCH model assumes the conditional distribution of the residual to be $\varepsilon_t \sim N(0, \sigma_t^2)$ and posits that the conditional variance σ_t^2 is the sum of three linear processes, with coefficients for each term: an ARCH polynomial, i.e. the past squared innovations $\{\varepsilon_{t-1}^2, \dots, \varepsilon_{t-q}^2\}$; a GARCH polynomial, i.e. the past conditional variances $\{\sigma_{t-1}^2, \dots, \sigma_{t-p}^2\}$; and the constants μ and ω for the innovation mean and the conditional variance models, respectively. More precisely, the general form of a GARCH(q, p) model is:

$$r_t = \mu + \varepsilon_t \quad \text{where} \quad \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim N(0,1) \equiv \varepsilon_t|\psi_{t-1} \sim N(0, \sigma_t^2) \tag{55}$$

$$\begin{aligned} \sigma_t^2 &= \omega + \sum_{i=1}^{q} \alpha_i \varepsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2 \qquad (56) \\ &= \omega + \alpha_1 \varepsilon_{t-1}^2 + \cdots + \alpha_q \varepsilon_{t-q}^2 + \beta_1 \sigma_{t-1}^2 + \cdots + \beta_p \sigma_{t-p}^2 \end{aligned}$$

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

The distributional assumption $\varepsilon_t \sim N(0, \sigma_t^2)$ allows to set up the likelihood function and estimate the model consisting of Equations 55 and 56 using maximum likelihood estimation. The log-likelihood is easily found as the model is conditionally Gaussian:

$$log(L) = \sum_{t=1}^{T} L_t \quad \text{where} \quad L_t = -\frac{1}{2} log(2\pi) - \frac{1}{2} log(\sigma_t^2) - \frac{1}{2} \frac{\varepsilon_t^2}{\sigma_t^2} \tag{57}$$

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

It is sometimes found that the standardized values ε_t / σ_t still have too fat tails compared to the assumed distribution $N(0,1)$. Therefore, a popular variation of the GARCH model is the

GARCH-t model, which makes a different assumptions about the distribution of the error term, namely that they follow a a t-distribution. This variation allows to account for the leptokurtosis. More precisely, the general form of a GARCH-t model is:

$$r_t = \mu + \varepsilon_t \quad \text{where} \quad \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim t(v) \tag{58}$$

where v is the degrees of freedom and where Equation 56 is unchanged.

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

$$\begin{aligned} r_t = \mu_t + \varepsilon_t \quad \text{where} \quad & \mu_t = \delta_0 + \delta_1 r_{t-1} + \cdots + \delta_r r_{t-r} \\ & \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim t(v) \end{aligned} \tag{59}$$

where δ_k is the coefficient of the lag-r regressor and v is the degrees of freedom, and where Equation 56 is unchanged.

5.2 GJR GARCH

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

$$r_t = \mu + \varepsilon_t \quad \text{where} \quad \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim N(0,1) \equiv \varepsilon_t|\psi_{t-1} \sim N(0,\sigma_t^2) \tag{60}$$

$$\begin{aligned} \sigma_t^2 &= \omega + \sum_{i=1}^{q} \alpha_i \varepsilon_{t-i}^2 + \sum_{j=1}^{p} \beta_j \sigma_{t-j}^2 + \sum_{i=1}^{q} \gamma_i \mathbb{1}_{[\varepsilon_{t-i}<0]} \varepsilon_{t-i}^2 \\ &= \omega + \alpha_1 \varepsilon_{t-1}^2 + \cdots + \alpha_q \varepsilon_{t-q}^2 + \beta_1 \sigma_{t-1}^2 + \cdots + \beta_p \sigma_{t-p}^2 + \\ &\quad + \gamma_1 \mathbb{1}_{[\varepsilon_{t-1}<0]} \varepsilon_{t-1}^2 + \cdots + \gamma_q \mathbb{1}_{[\varepsilon_{t-q}<0]} \varepsilon_{t-q}^2 \end{aligned} \tag{61}$$

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

Equation 61 shows how the effect of the shock ε_{t-i}^2 is α_i if the shock is negative and $\alpha_i + \gamma_i$ if the shock is positive. Therefore, when $\gamma_i < 0$ the negative shock has a bigger impact on volatility than a positive shock of equal magnitude. Moreover, if all leverage coefficients are zero, then the GJR model is equivalent to the GARCH model. Finally, the GJR can be estimated with maximum likelihood estimation using Equation 57.

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

5.3 EGARCH

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

$$r_t = \mu + \varepsilon_t \quad \text{where} \quad \varepsilon_t = \sigma_t z_t,\ z_t|\psi_{t-1} \sim N(0,1) \equiv \varepsilon_t|\psi_{t-1} \sim N(0,\sigma_t^2) \tag{62}$$

$$log(\sigma_t^2) = \omega + \sum_{i=1}^{q} \alpha_i \left\{ \frac{|\varepsilon_{t-i}|}{\sigma_{t-i}} - E\left[\frac{|\varepsilon_{t-i}|}{\sigma_{t-i}}\right] \right\} + \sum_{j=1}^{p} \beta_j log(\sigma_{t-j}^2) + \sum_{i=1}^{q} \gamma_i \frac{\varepsilon_{t-i}}{\sigma_{t-i}} \tag{63}$$

where, since z_t is Gaussian, then:

$$E\left[\frac{|\varepsilon_{t-i}|}{\sigma_{t-i}}\right] = E[\,|z_{t-i}|\,] = \sqrt{\frac{2}{\pi}} \tag{64}$$

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

Equation 63 shows how the effect of the shock ε_{t-i}^2 is symmetric via $|\varepsilon_{t-i}^2|$, i.e. both positive and negative shocks have the same impact on volatility. However, the leverage polynomial modifies this to make the effect asymmetric. In particular, when $\gamma_i < 0$ the negative shock has a bigger impact on volatility than a positive shock of equal magnitude.

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

$$E\left[\frac{|\varepsilon_{t-i}|}{\sigma_{t-i}}\right] = E[\ |z_{t-i}|\ ] = \sqrt{\frac{v-2}{\pi}}\ \frac{\Gamma\left(\frac{v-1}{2}\right)}{\Gamma\left(\frac{v}{2}\right)} \tag{65}$$

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

5.4 Traditional Quantile Regression

For a probability level $\tau \in (0,1)$, the τ-quantile of a probability density function $p(y)$ is defined as $q = F^{-1}(\tau)$ where $F(y)$ is the cumulative distribution function of $p(y)$. Quantile regression aims to estimate the τ-quantile q of the conditional distribution $p(y|x)$ 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:

$$L_\tau(y,q) = \begin{cases} \tau|y-q| & \text{if } y > q \\ (1-\tau)|y-q| & \text{if } y \leq q \end{cases} \tag{66}$$

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

$$\min_\theta \mathbb{E}[L_\tau(y, f_\theta(x))] \tag{67}$$

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

$$\min_\theta \frac{1}{N}\sum_{i=1}^{N} L_\tau(y_i, f_\theta(x_i)) \tag{68}$$

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:

$$\min_{\theta_1,\ldots,\theta_K} \frac{1}{K}\frac{1}{N}\sum_{k=1}^{K}\sum_{i=1}^{N} L_{\tau_k}(y_i, f_{\theta_k}(x_i)) \tag{69}$$

Besides the increasing number of parameters K, the problem with Equation 69 is that it may lead to quantile crossing, i.e. that for some x and $\tau_j < \tau_k$, it may happen that $f_{\theta_j}(x) > f_{\theta_k}(x)$. This is caused by the fact that θ_j and θ_k 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.

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

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:

$$Q(\tau|\mu, \sigma, u, d) = \mu + \sigma Z_\tau \left(\frac{e^{uZ_\tau}}{A} + 1\right)\left(\frac{e^{-dZ_\tau}}{A} + 1\right) \tag{70}$$

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

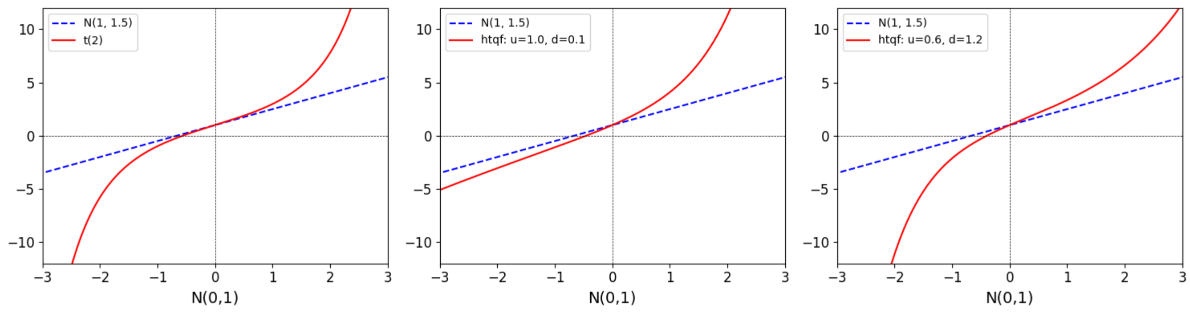


Figure 12: Perceptron learning rule

5.6 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:

$$q_k^t = Q(\tau_k|\mu_t, \sigma_t, u_t, d_t) \quad \text{for} \quad k = 1, \dots, K \tag{71}$$

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:

$$\begin{bmatrix} \boldsymbol{x}_1^t \\ \dots \\ \boldsymbol{x}_L^t \end{bmatrix} = \begin{bmatrix} r_{t-1} & (r_{t-1} - \bar{r}_t)^2 & (r_{t-1} - \bar{r}_t)^3 & (r_{t-1} - \bar{r}_t)^4 \\ \dots & \dots & \dots & \dots \\ r_{t-L} & (r_{t-L} - \bar{r}_t)^2 & (r_{t-L} - \bar{r}_t)^3 & (r_{t-L} - \bar{r}_t)^4 \end{bmatrix} \tag{72}$$

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 $[\boldsymbol{x}_1^t, \ldots, \boldsymbol{x}_L^t]^\top$:

$$\left[\mu_t, \sigma_t, u_t, d_t\right]^\top = tanh(W^o h_t + b^o) \quad \text{where} \quad h_t = LSTM_\Theta(x_1^t, \ldots, x_L^t) \tag{73}$$

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 $\{\tau_1, \ldots, \tau_K\}$, 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:

$$\min_{\Theta, W^o, b^o} \frac{1}{K}\frac{1}{T-L}\sum_{k=1}^{K}\sum_{t=L+1}^{T} L_{\tau_k}\left[r_t, Q(\tau_k|\mu_t, \sigma_t, u_t, d_t)\right] \tag{74}$$

The complete quantile regression model with HTQF and LSTM, denoted HTQF-LSTM, combines Equations 70, 72, 73, 74. 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.

6 Data

In this section, we 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.

6.1 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 on April 2015; the latter covers the period 01/01/1993 - 31/12/2014, is delivered one month at the time and has time stamps with precision of a second. The time stamps are based on the New York Eastern Time and include changes between standard and daylight savings time. Transactions reported outside of the consolidated tape hours [6] and transactions on NYSE listed securities between 8:00 AM and 9:30 AM by other markets are not included.

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

Focusing on the trade files of the Daily TAQ Product, the tables contain the following data for every trade reported in the consolidated tape by all CTA and UTP participants: date of the trade, time at which the trade was published by the SIP, exchange where the trade occurred, root and suffix of the ticker, sale condition, number of shares traded, dollar price per share,

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

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

6.2 Data Preparation

As explained in Brownlees and Gallo (2006), the structure of tick data is affected by several factors, such as the regulatory framework (e.g. the length of the trading day), the procedures of the institution that collects and produces the information (e.g. physical vs hybrid markets), and the technological improvements (e.g. changes in the precision of the tick). Furthermore, the series of tick data might contain errors, which are not identified as such by the data provider, might not be time ordered, and might be affected by unusual market conditions (e.g. opening and closing time or trading halts).

Therefore, tick data on financial markets require the application of standard filters to remove bad record, as well as methods to remove outliers, which do not correspond to plausible market activity, and to aggregate data into time series suitable to conduct an analysis. When studying ultra-high-frequency measures of volatility, these data cleaning and data management processes are particularly relevant to correct the impact of unwanted observations, which would otherwise result in wrong estimates of the coefficients and erroneous behaviors of the models. Nonetheless, these procedures involve subjective decisions, which result in different trade-offs between the loss of information contained in the discarded data and the noise reduction.

The data preparation process starts with the data cleaning process. As a first step, we keep only regular trades, which have not been corrected, changed or marked as cancel or error, i.e. all records with a correction indicator (tr_corr) field equal to 00. We remove all original trades which have later been corrected (01), outright cancelled (08), cancelled due to error (07) or cancelled due to symbol correction (09), and the corresponding correction (12), cancel (10) and error (11) records, and symbol corrections (02) (see NYSE (2012), pg. 19 and NYSE (2017), pg. 19). As a second step, we keep only trades with a regular timing, i.e. all records with a sale condition (tr_scond) equal to @, A to F, H to K, M to O, Q to S, V, W, Y or 1 to 9. We remove all trades which have been reported late (G, L, Z), executed outside the regular trading hours (T, U, X) or that refer to conditions at an earlier point in the trading session (P) (see NYSE

(2012), pg. 20-23, NYSE (2017), pg. 17-19 and Nasdaq (2018), pg. 41-43). As a third step, we remove all observations that are inconsistent with the current market activity. To identify these outliers, data series other than the tick-by-tick price are of little help. For example, while quotes cannot be used because it is very difficult to match trades and quotes with precision, as explained in Vergote (2005), volumes cannot be used either because their plausibility cannot be assessed beyond the plausibility of the corresponding price. Therefore, we adopt a variation of the heuristic procedure outlined in Brownlees and Gallo (2006). This identifies ouliers by calculating their relative distance from a neighborhood of closest valid observations. Let $\{p_t^i\}_{t=1}^s$ be the time-ordered tick-by-tick price series of stock i. The rule to identify outliers is:

$$|p_t^i - \bar{p}_t^i(k)| < 3s_t^i(k) + \gamma = \begin{cases} \text{if true, then observation } p_t^i \text{ is kept} \\ \text{if false, then observation } p_t^i \text{ is removed} \end{cases} \tag{75}$$

where $\bar{p}_t^i(k)$ and $s_t^i(k)$ are the δ-trimmed sample mean and sample standard deviation of a neighborhood of k observations around p_t^i and γ is a granularity parameter. More precisely, the neighborhood is defined so that every price observation is compared with observations from the same trading day: the neighborhood of the first/last $(k-1)/2$ observations are the first/last k observation of the day and the neighborhood of a generic observation in the middle of the day is made up of the preceding and following $(k-1)/2$ observations. The granularity parameter ensures that the right-hand side of the inequality is not zero in case the whole neighborhood contains equal prices. Brownlees and Gallo (2006) argue that the lower the level of trading activity, the smaller k should be, so that the rolling window does not contain too distant prices, and that γ should be a multiple of the minimum price variation. They then test different combinations of k and γ and choose the one that yields better results from a visual inspection of the plot. Following their methodology, we set the trimming parameter constant to $\delta = 10\%$ and we identify the outliers using the heuristic procedure outlined above, for all possible combinations of $k \in \{41, 61, 81, 101\}$ and $\gamma \in \{0.02, 0.04, 0.06\}$. For each stock, we then select the combination of k and γ that identifies the minimum number of outliers. Indeed, after testing it on a sample of ten stocks and ten time spans of different length, this rule proves to be more robust than, say, selecting the combination of k and γ that identifies the maximum number of outliers or the one that yields the highest or lowest average distance between the observations identified as outliers and the corresponding trimmed sample mean. Moreover, this rule allows to automate the parameter selection procedure and removes the need for a visual inspection of the plot.

Following these three steps of the data cleaning process, we implement the data management process. Indeed, the time series of prices still display an undesired feature, namely the presence of transactions executed at the same time but at different prices. This can be explained by the fact that some securities can trade simultaneously on different exchanges, that the execution of one trade can sometimes generate multiple transaction reports, or that the execution of non-simultaneous trades can be reported at the same time due to reporting approximations. Therefore, as a fourth step, we aggregate blocks of simultaneous observations retaining the median value of the prices, so to obtain one observation per time stamp. Finally, the time series of prices displays one last undesired feature, namely the presence of irregular time intervals between subsequent observations, as dictated by variations in the trading frequency. Therefore, as a fifth step, we resample the data at lower frequency, retaining the last observation in the interval[7], so to obtain time series with equally spaced time intervals. To choose the appropriate resampling frequency, we computed the ratio of the number of time intervals with no observation over the total number of time intervals for a sample of highly liquid stocks. We found 5 seconds to be an appropriate frequency to balance the need for interpolation and the information content provided by high frequency data. Moreover, although some exceptionally liquid stocks would allow to push the resampling frequency up to 2 seconds with no significant increase in the need for interpolation, choosing 5 second intervals allows to make the data extraction process more robust. For example, even for highly liquid stocks such as Amazon or Tesla, a resampling frequency of 2 seconds would result in 10% - 20% of the intervals not containing an observation on some days with a low trading volume, while a resampling frequency of 5 seconds drastically reduces this number to 1% on such days. Finally, we applied a linear interpolation (coupled with next and previous point interpolation for the beginning and end of the daily sequence respectively) to fill in the missing data.

As a final note, we would like to point out that the choices of the specific parameters made in the data cleaning and data management processes do not significantly impact the time series creation. Indeed, for very liquid stocks, a neighborhood of high frequency observations is very dense and different choices would yield very similar results. Figure 13 shows 10 minutes of trade records for Apple, before and after the five step of the data cleaning and data management processes process just outlined.

6.3 Sample

Returns are computer across days, to account for large intraday jumps. If we do not compute the return across days, we would fail to capture the right level of volatility. Assume constant

[7]For both the data aggregation and resampling processes, the volumes are added up, as if they were a single trade, although this is not relevant for this study, which is focused on prices (and returns).

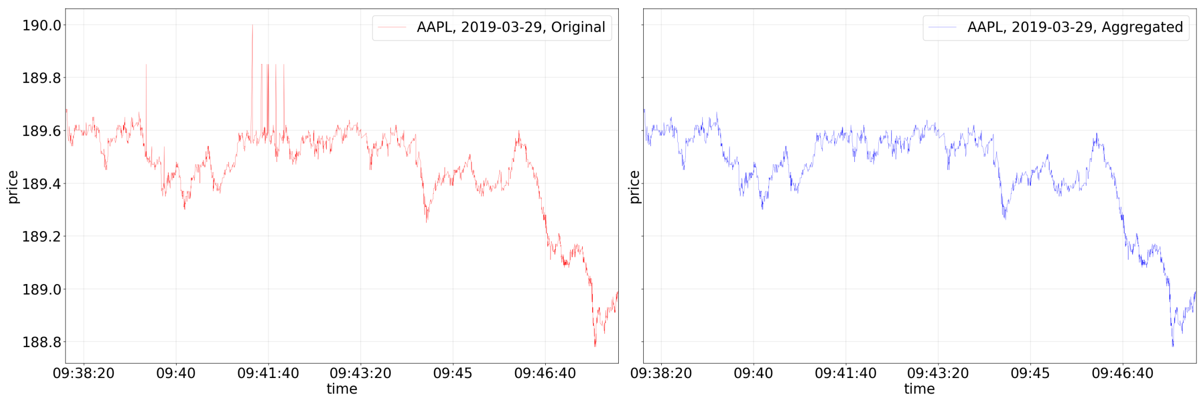


Figure 13: Sample trade records for Apple before and after the cleaning process

price the first day, a big intraday jump and a constant price the second day: this would result in zero returns if we exclude the intraday return. Doing so introduces a negligibly small number of returns that are very large compared to the rest: these observations are very large even after the standardizarion process, but their number is insignificant.

Moreover, the time range goes from 9:35 to 15:55 which allows to disregard the noise at the beginning and end of the trading day.

——————-

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 6.3, 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.

$\tau \in$ [0.01, 0.05, . . . , 0.95, 0.99]

Model \Asset	AAPL				AMD				AMZN				CSCO				FB			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	0.1377	1	1		0.1683	2	1		0.1629	1	1		0.1663	1	1		0.1446	3	2	
GARCH-t	0.1154	1	1		0.1512	1	1		0.1553	1	1		0.1541	1	2		0.1260	1	1	
AR-GARCH-t	0.1145	1	1	1	0.1484	1	1	1	0.1510	1	1	1	0.1506	2	1	1	0.1256	1	1	1
EGARCH	0.1381	1	3		0.1683	1	3		0.1626	1	1		0.1672	3	1		0.1423	1	4	
EGARCH-t	0.1149	2	4		0.1524	3	1		0.1562	3	2		0.1540	3	2		0.1276	3	2	
AR-EGARCH-t	0.1159	1	4	1	0.1498	3	4	1	0.1520	3	2	1	0.1509	3	2	1	0.1272	3	2	1
GJR	0.1377	1	1		0.1683	2	1		0.1629	1	1		0.1663	1	1		0.1446	3	2	
GJR-t	0.1154	1	1		0.1512	1	1		0.1553	1	1		0.1541	1	2		0.1260	1	1	
AR-GJR-t	0.1145	1	1	1	0.1484	1	1	1	0.1510	1	1	1	0.1506	2	1	1	0.1256	1	1	1
LSTM-HTQF	**0.1104**				**0.1467**				**0.1489**				**0.1478**				**0.1249**			

$\tau' \in$ [0.01, 0.05, 0.1]

Model \Asset	AAPL				AMD				AMZN				CSCO				FB			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	0.0728	1	1		0.0811	2	1		0.0717	1	1		0.0777	1	1		0.0712	3	2	
GARCH-t	0.0546	1	1		0.0635	1	1		0.0634	1	1		0.0674	1	2		0.0541	1	1	
AR-GARCH-t	0.0544	1	1	1	0.0628	1	1	1	0.0623	1	1	1	0.0651	2	1	1	0.0541	1	1	1
EGARCH	0.0772	1	3		0.0812	1	3		0.0715	1	1		0.0794	3	1		0.0692	1	4	
EGARCH-t	0.0574	2	4		0.0662	3	1		0.0655	3	2		0.0675	3	2		0.0565	3	2	
AR-EGARCH-t	0.0573	1	4	1	0.0654	3	4	1	0.0643	3	2	1	0.0659	3	2	1	0.0564	3	2	1
GJR	0.0728	1	1		0.0811	2	1		0.0717	1	1		0.0777	1	1		0.0712	3	2	
GJR-t	0.0546	1	1		0.0635	1	1		0.0634	1	1		0.0674	1	2		0.0541	1	1	
AR-GJR-t	0.0544	1	1	1	0.0628	1	1	1	0.0623	1	1	1	0.0651	2	1	1	0.0541	1	1	1
LSTM-HTQF	**0.0540**				**0.0618**				**0.0611**				**0.0622**				**0.0533**			

$\tau \in$ [0.01, 0.05, . . . , 0.95, 0.99]

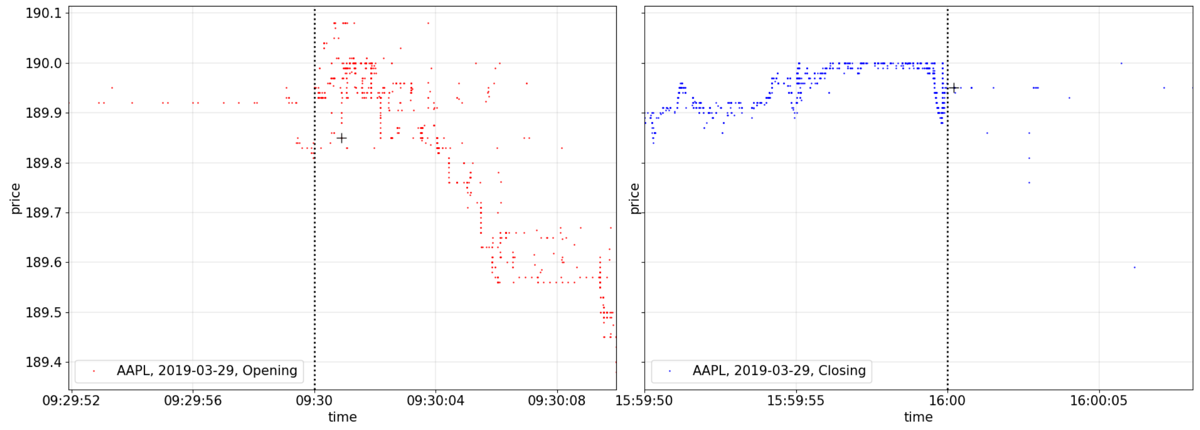
Model \Asset	INTC				JPM				MSFT				NVDA				TSLA			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	0.1611	1	1		0.1628	1	1		0.1555	1	1		0.1680	1	1		0.1559	1	1	
GARCH-t	0.1422	2	1		0.1517	1	1		0.1455	1	1		0.1500	1	1		0.1311	1	1	
AR-GARCH-t	0.1400	3	1	1	0.1509	1	1	1	0.1443	1	1	1	0.1489	1	1	1	0.1289	1	1	1
EGARCH	0.1604	3	4		0.1681	3	3		0.1661	3	1		0.1679	1	1		0.1558	1	1	
EGARCH-t	0.1414	3	2		0.1512	3	4		0.1438	3	1		0.1517	2	1		0.1336	2	3	
AR-EGARCH-t	0.1396	3	2	1	0.1506	3	4	1	0.1431	3	1	1	0.1506	2	3	1	0.1286	1	2	1
GJR	0.1611	1	1		0.1628	1	1		0.1555	1	1		0.1680	1	1		0.1559	1	1	
GJR-t	0.1422	2	1		0.1517	1	1		0.1455	1	1		0.1500	1	1		0.1311	1	1	
AR-GJR-t	0.1400	3	1	1	0.1509	1	1	1	0.1443	1	1	1	0.1489	1	1	1	0.1289	1	1	1
LSTM-HTQF	**0.1367**				**0.1478**				**0.1394**				**0.1478**				**0.1278**			

$\tau' \in$ [0.01, 0.05, 0.1]

Model \Asset	INTC				JPM				MSFT				NVDA				TSLA			
	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r	Loss	p	q	r
GARCH	0.0807	1	1		0.0778	1	1		0.0750	1	1		0.0803	1	1		0.0787	1	1	
GARCH-t	0.0637	2	1		0.0697	1	1		0.0684	1	1		0.0632	1	1		0.0541	1	1	
AR-GARCH-t	0.0625	3	1	1	0.0691	1	1	1	**0.0678**	1	1	1	0.0629	1	1	1	0.0534	1	1	1
EGARCH	0.0811	3	4		0.0841	3	3		0.0935	3	1		0.0802	1	1		0.0787	1	1	
EGARCH-t	0.0638	3	2		0.0710	3	4		0.0757	3	1		0.0660	2	1		0.0584	2	3	
AR-EGARCH-t	0.0630	3	2	1	0.0706	3	4	1	0.0748	3	1	1	0.0656	2	3	1	0.0530	1	2	1
GJR	0.0807	1	1		0.0778	1	1		0.0750	1	1		0.0803	1	1		0.0787	1	1	
GJR-t	0.0637	2	1		0.0697	1	1		0.0684	1	1		0.0632	1	1		0.0541	1	1	
AR-GJR-t	0.0625	3	1	1	0.0691	1	1	1	0.0678	1	1	1	0.0629	1	1	1	0.0534	1	1	1
LSTM-HTQF	**0.0595**				**0.0664**				0.0696				**0.0617**				**0.0527**			

Table 2: Results

Figure 14: Trade Records for [illegible]



7 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.

7.1 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 [8]. 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).

7.2 LSTM-TQR

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 73 and 74 as follows:

$$q_k^t : \left[q_1^t, \dots, q_K^t\right]^\top = tanh(W^o h_t + b^o) \tag{76}$$

$$\min_{\Theta, W^o, b^o} \frac{1}{K} \frac{1}{T-L} \sum_{k=1}^{K} \sum_{t=L+1}^{T} L_{\tau_k}(r_t, q_k^t) \tag{77}$$

For new time t' the predicted quantiles $q_1^{t'}, \dots, q_K^{t'}$ of $r_{t'}$ are sorted to avoid the quantile crossing issue.

Moreover, generally, the feature vector sequence $\{x_1^t, \dots, x_L^t\}$ 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 $\{r_{t-1}, r_{t-2}, \dots\}$.

7.3 Discussion

The HTQF is more suited to model the leptokurtic and asymmetric behavior of the tails of a distribution than the GARCH based approach, which must assume a particular specification of the PDF, such as the skewed t-distribution. Moreover, the HTQF has time dependent parameters, which allow to account for the time varying behavior of the tails; on the other hand, in order to do so, the GARCH based models must specify the PDF in a complicated analytical form,

[8]Overnight jumps are eliminated.

which makes the model estimation unfeasible if we assume time varying degrees of freedom. Lastly, while the non-linear activation functions used in the LSTM allow to learn the non-linear dependencies on the past data, the linear specification of the GARCH based models does not allow to do so.

The HTQF also overcomes the three limitations of the traditional quantile regression. Firstly, the HTQF always requires four time-varying parameters (μ_t, σ_t, u_t, d_t) to estimate a number K of quantiles, which removes the need to estimate an increasingly number of parameters. Secondly, the HTQF is monotonically increasing in Z_τ and τ, which eliminates the problem of quantile crossing. Thirdly, the HTQF's mapping from the parameters to the conditional quantiles is clearly interpretable.

7.4 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:

$$v_t = max\{8 - 2\pi_t, 3\} \quad \text{where} \quad \pi_t = \sqrt{0.136 + 0.257 r_{t-1}^2 + 0.717 \pi_{t-1}^2} \tag{78}$$

$$\sigma_t = \sqrt{0.293 + 0.161 r_{t-1}^2 + 0.575 \sigma_{t-1}^2} \tag{79}$$

$$r_t = \sigma_t z_t \quad \text{where } z_t \text{ is sampled from } t(v_t) \tag{80}$$

From Figure 15 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 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.

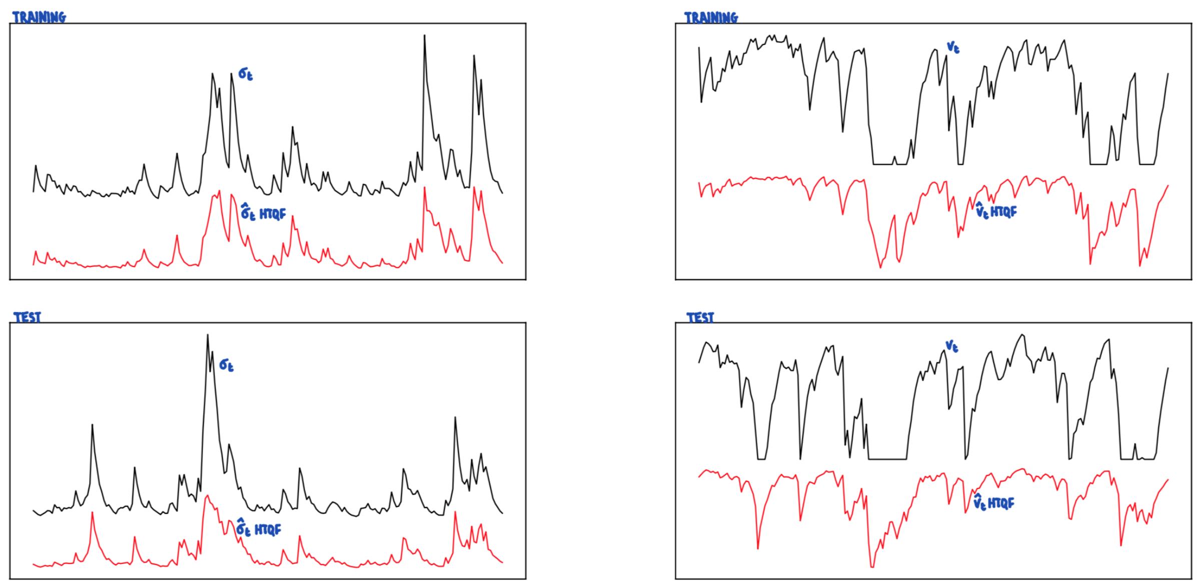


Figure 15: Comparison between true parameters and learnt ones.

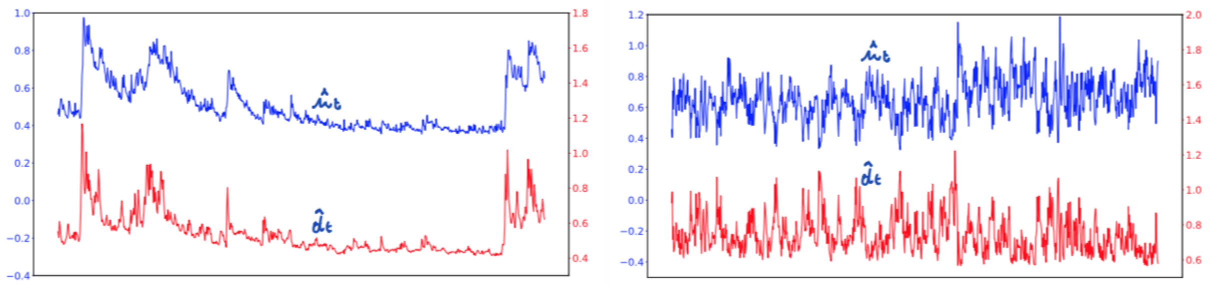


Figure 16: HTQF parameters.

7.5 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.

In order to investigate the dynamics captured by the HTQF-LSTM model, Figure 16 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 16 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.

7.6 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 $\left[0.01, 0.05, 0.1\right]$, the quantiles of which are VaR representing downside risk.

8 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.

Appendices

A Weight Updates in the Adaline Model

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

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w} \quad \text{where} \quad \Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w}) \tag{81}$$

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

$$\begin{aligned}
\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j}\frac{1}{2}\sum_i \Big(y_i - \phi(z_i)\Big)^2 \\
&= \frac{1}{2}\frac{\partial}{\partial w_j}\sum_i \Big(y_i - \phi(z_i)\Big)^2 \\
&= \frac{1}{2}\sum_i 2\Big(y_i - \phi(z_i)\Big)\frac{\partial}{\partial w_j}\Big(y_i - \phi(z_i)\Big) \\
&= \sum_i \Big(y_i - \phi(z_i)\Big)\frac{\partial}{\partial w_j}\Big(y_i - \sum_j (w_j x_i^j)\Big) \\
&= \sum_i \Big(y_i - \phi(z_i)\Big)(-x_i^j) \\
&= -\sum_i \Big(y_i - \phi(z_i)\Big)x_i^j
\end{aligned} \tag{82}$$

So that the weight update becomes:

$$\Delta w_j = -\eta\frac{\partial J}{\partial w_j} = \eta\sum_i \Big(y_i - \phi(z_i)\Big)x_i^j \tag{83}$$

B Logistic Sigmoid Function

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

$$f(P) = \frac{P}{1-P} \tag{84}$$

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

Then, let the logit function be:

$$logit(P) = log\Big(f(P)\Big) = log\left(\frac{P}{1-P}\right) \tag{85}$$

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

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

$$logit(P) = logit\Big(p(y=1|\boldsymbol{x}_i)\Big) = x_i^0 w_0 + \boldsymbol{x}_i \boldsymbol{w} = x_i^0 w_0 + x_i^1 w_1 + \cdots + x_i^m w_m = z_i \tag{86}$$

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

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

$$p(y=1|\boldsymbol{x}_i) = logit^{-1}(z_i) \tag{87}$$

which can be explicited as follows:

$$\begin{aligned} log\left(\frac{P}{1-P}\right) &= Z \\ \frac{P}{1-P} &= e^Z \\ P &= e^Z(1-P) \\ P &= e^Z - e^Z P \\ P(1+e^Z) &= e^Z \\ P &= \frac{e^Z}{1+e^Z} \\ P &= \frac{e^Z e^{-Z}}{(1+e^Z)e^{-Z}} \\ P &= \frac{1}{e^{-Z}+1} \end{aligned} \tag{88}$$

which is formalized into the logistic sigmoid function:

$$\phi(Z) = \frac{1}{1+e^{-Z}} \tag{89}$$

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

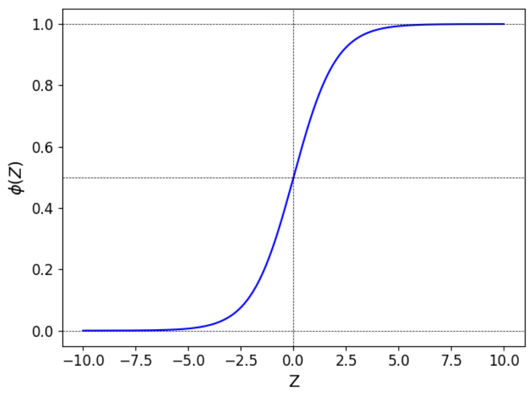


Figure 17: Logistic sigmoid function

C Cost Function in the Linear Regression Model

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

$$f(\boldsymbol{y}|\boldsymbol{X};\boldsymbol{w}) = \prod_{i=1}^{n} f(y_i|\boldsymbol{x}_i;\boldsymbol{w}) \tag{90}$$

The likelihood function of the samples is then given by:

$$\begin{aligned} L(\boldsymbol{w}|\boldsymbol{y},\boldsymbol{X}) &= \prod_{i=1}^{n} L_i(\boldsymbol{w}|y_i,\boldsymbol{x}_i) \\ &= \prod_{i=1}^{n} f(y_i|\boldsymbol{x}_i;\boldsymbol{w}) \\ &= \prod_{i=1}^{n} \Big(\phi(z_i)\Big)^{y_i}\Big(1-\phi(z_i)\Big)^{1-y_i} \end{aligned} \tag{91}$$

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

$$logL(\boldsymbol{w}|\boldsymbol{y},\boldsymbol{X}) = \sum_{i=1}^{n}\Big[y_i\ log\Big(\phi(z_i)\Big) + (1-y_i)\ log\Big(1-\phi(z_i)\Big)\Big] \tag{92}$$

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

$$J(\boldsymbol{w}) = \sum_{i=1}^{n}\Big[-y_i\ log\Big(\phi(z_i)\Big) - (1-y_i)\ log\Big(1-\phi(z_i)\Big)\Big] \tag{93}$$

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

$$J(\boldsymbol{w}) = -y_i\, log\Big(\phi(z_i)\Big) - (1-y_i)\, log\Big(1-\phi(z_i)\Big) \tag{94}$$
$$= \begin{cases} -\,log\Big(\phi(z_i)\Big) & \text{if } y_i = 1 \\ -\,log\Big(1-\phi(z_i)\Big) & \text{if } y_i = 0 \end{cases}$$

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

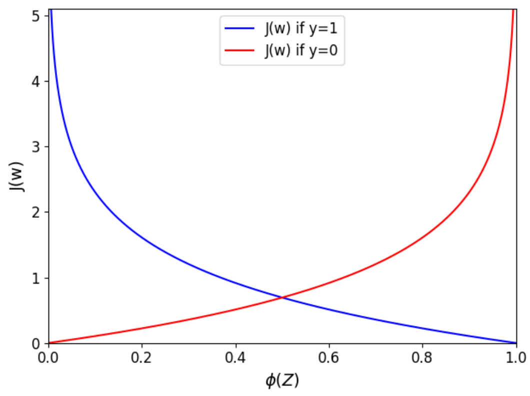


Figure 18: Logistic cost function

D Gradient of the Cost Function in the Linear Regression Model

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

$$\boldsymbol{w} := \boldsymbol{w} + \Delta\boldsymbol{w} \quad \text{where} \quad \Delta\boldsymbol{w} = -\eta\nabla J(\boldsymbol{w}) \tag{95}$$

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

$$\begin{aligned}\frac{\partial J}{\partial w_j} &= \frac{\partial}{\partial w_j}\sum_i \Big[-y_i\, log\Big(\phi(z_i)\Big) - (1-y_i)\, log\Big(1-\phi(z_i)\Big)\Big] \\ &= -\sum_i\left\{\left[y_i\,\frac{\partial log\Big(\phi(z_i)\Big)}{\partial\phi(z_i)} + (1-y_i)\,\frac{\partial log\Big(1-\phi(z_i)\Big)}{\partial\phi(z_i)}\right]\frac{\partial\phi(z_i)}{\partial z_i}\,\frac{\partial z_i}{\partial w_i}\right\} \\ &= -\sum_i\left\{\left[\frac{y_i}{\phi(z_i)} - \frac{1-y_i}{1-\phi(z_i)}\right]\frac{e^{-z_i}}{(1+e^{-z_i})^2}x_i^j\right\} \\ &= -\sum_i\left\{\left[\frac{y_i}{\phi(z_i)} - \frac{1-y_i}{1-\phi(z_i)}\right]\frac{1}{1+e^{-z_i}}\left(1-\frac{1}{1+e^{-z_i}}\right)x_i^j\right\}\end{aligned} \tag{96}$$

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

So that the weight update becomes:

$$\Delta w_j = -\eta \frac{\partial J}{\partial w_j} = \eta \sum_i \Big(y_i - \phi(z_i) \Big) x_i^j \tag{97}$$

E L2-Regularization to Prevent Overfitting

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

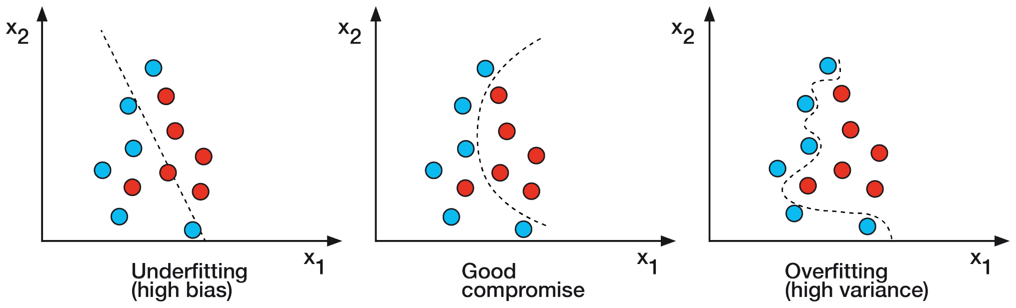


Figure 19: Underfitting and overfitting

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

$$\frac{\lambda}{2}||\boldsymbol{w}||^2 = \frac{\lambda}{2} \sum_j w_j^2 \tag{98}$$

where λ is the regularization parameter, which allows to control how well the model fits the training data while avoiding extreme weight values. The higher the value of λ, the stronger the regularization. For regularization to work successfully, it is necessary to apply a standardization to the data, so that all features are on a comparable scale.

F Backpropagation Algorithm

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

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

$$\begin{aligned}\frac{\partial}{\partial X}F(X) &= \frac{\partial}{\partial X}f(g(h(u(v(X))))) \\ &= \frac{\partial f}{\partial g}\frac{\partial g}{\partial h}\frac{\partial h}{\partial u}\frac{\partial u}{\partial v}\frac{\partial v}{\partial X}\end{aligned} \tag{99}$$

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

G Derivative of the Error Term of the Hidden Layer

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

$$\boldsymbol{\Delta}_{(h)} = \boldsymbol{\Delta}_{(out)}(\boldsymbol{W}_{(out)})^\top \odot \frac{\partial \phi(\boldsymbol{Z}_{(h)})}{\partial \boldsymbol{Z}_{(h)}} \tag{100}$$

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

$$\begin{aligned}\frac{\partial \phi(\boldsymbol{Z}_{(h)})}{\partial \boldsymbol{Z}_{(h)}} &= \frac{\partial}{\partial \boldsymbol{Z}_{(h)}}\left(\frac{1}{1+e^{-\boldsymbol{Z}_{(h)}}}\right) \\ &= \frac{e^{-\boldsymbol{Z}_{(h)}}}{(1+e^{-\boldsymbol{Z}_{(h)}})^2} \\ &= \frac{1+e^{-\boldsymbol{Z}_{(h)}}}{\big(1+e^{-\boldsymbol{Z}_{(h)}}\big)^2} - \left(\frac{1}{1+e^{-\boldsymbol{Z}_{(h)}}}\right)^2 \\ &= \frac{1}{1+e^{-\boldsymbol{Z}_{(h)}}} - \left(\frac{1}{1+e^{-\boldsymbol{Z}_{(h)}}}\right)^2 \\ &= \phi(\boldsymbol{Z}_{(h)}) - \big(\phi(\boldsymbol{Z}_{(h)})\big)^2 \\ &= \phi(\boldsymbol{Z}_{(h)})\big(1-\phi(\boldsymbol{Z}_{(h)})\big) \\ &= \boldsymbol{A}_{(h)}\big(1-\boldsymbol{A}_{(h)}\big)\end{aligned} \tag{101}$$

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

$$\boldsymbol{\Delta}_{(h)} = \boldsymbol{\Delta}_{(out)}(\boldsymbol{W}_{(out)})^\top \odot \Big(\boldsymbol{A}_{(h)} \odot \big(1-\boldsymbol{A}_{(h)}\big)\Big) \tag{102}$$

H Variants of the LSTM

Gers and Schmidhuber (2000) introduced a variant of the traditional LSTM that adds peephole connections, which allow the gates to receive the cell state as an additional input (Figure 20a).

$$\boldsymbol{f}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xf} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hf} + \boldsymbol{c}_{t-1}\boldsymbol{W}_{cf}\big) \tag{103}$$

$$\boldsymbol{i}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xi} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hi} + \boldsymbol{c}_{t-1}\boldsymbol{W}_{ci}\big) \tag{104}$$

$$\boldsymbol{o}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xo} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{ho} + \boldsymbol{c}_{t}\boldsymbol{W}_{co}\big) \tag{105}$$

Another variant uses coupled forget and input gates: instead of separately deciding what to forget and what to add new information to, the LSTM only forgets where new information is added (Figure 20b).

$$\boldsymbol{c}_t = \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + (1-\boldsymbol{f}_t) \odot \tilde{\boldsymbol{c}}_t \tag{106}$$

Cho et al. (2014) introduced the Gated Recurrent Unit, which combines the forget and input gates into an update gate, merges the cell state and the hidden state, and makes some other minor changes (Figure 20c).

$$\boldsymbol{z}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xz} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hz}\big) \tag{107}$$

$$\boldsymbol{r}_t = \sigma\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xr} + \boldsymbol{a}_{(h),t-1}\boldsymbol{W}_{hr}\big) \tag{108}$$

$$\tilde{\boldsymbol{a}}_{(h),t} = tanh\big(\boldsymbol{a}_{(in),t}\boldsymbol{W}_{xh} + (\boldsymbol{r}_t \odot \boldsymbol{a}_{(h),t-1})\boldsymbol{W}_{ph}\big) \tag{109}$$

$$\boldsymbol{a}_{(h),t} = (1 - \boldsymbol{z}_t) \odot \boldsymbol{a}_{(h),t-1} + \boldsymbol{z}_t \odot \tilde{\boldsymbol{a}}_{(h),t} \tag{110}$$

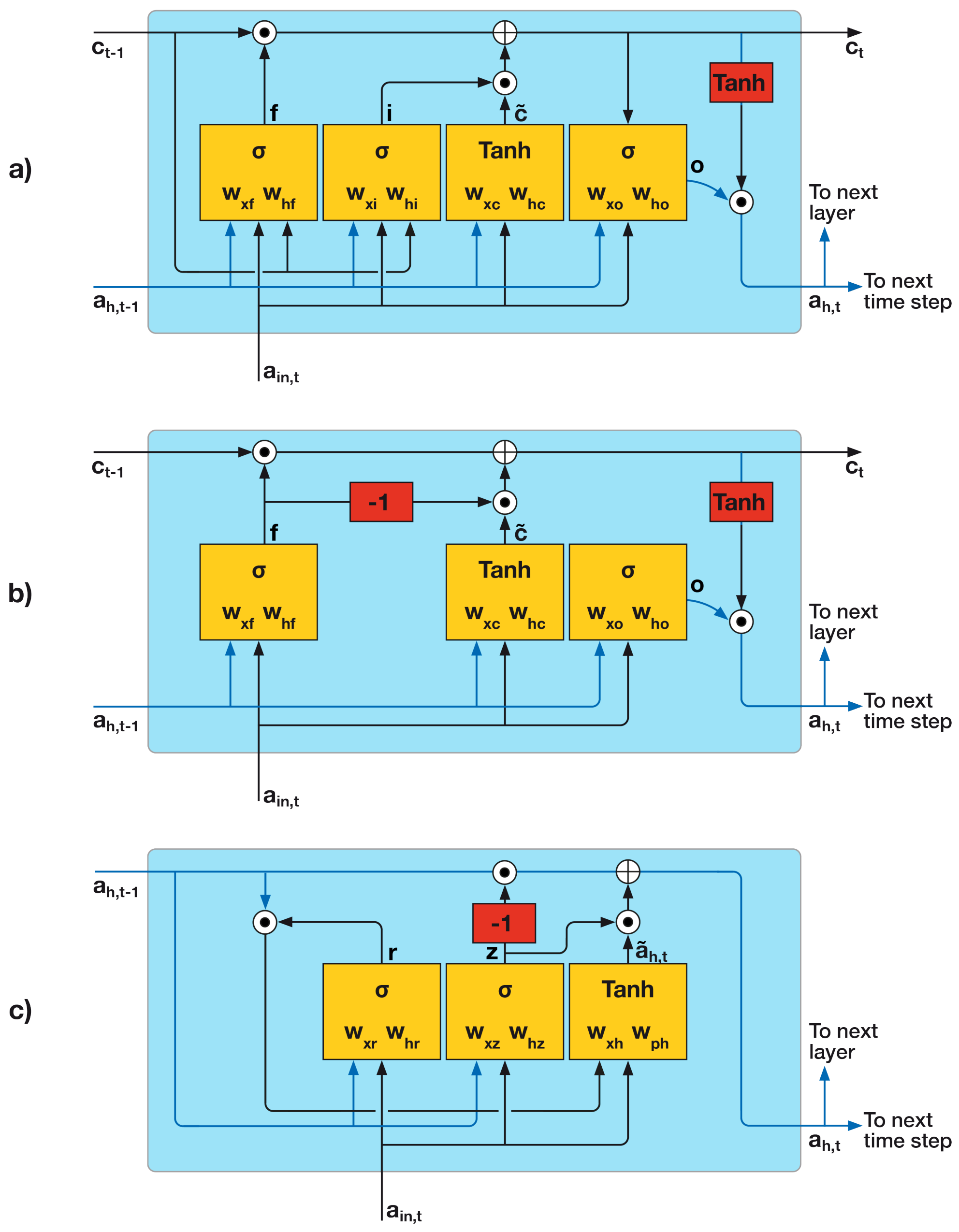


Figure 20: Maximization of the margin

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

I Support Vector Machine

Support Vector Machine is an algorithm for binary and multi-class classification, which learns the optimal weights by maximizing the margins, i.e. the distance between the decision boundary and the training samples that are closest to this hyperplane (called support vectors). The rationale behind the maximization of the margins is that such models are less subject to overfitting and tend to generalize better on unseen data. Figure 21 illustrates the concept of margin and support vectors.

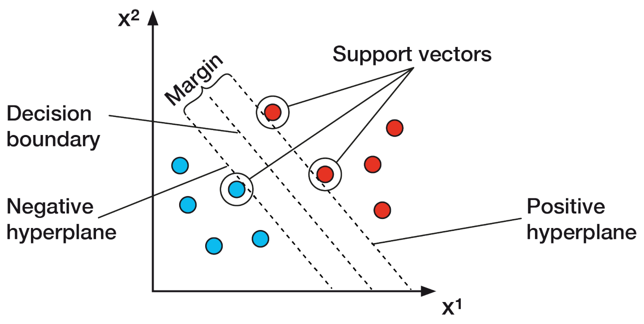


Figure 21: Maximization of the margin

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

$$\text{positive hyperplane: } w_0 + \boldsymbol{x}^{(+)}\boldsymbol{w} = 1 \tag{111}$$

$$\text{negative hyperplane: } w_0 + \boldsymbol{x}^{(-)}\boldsymbol{w} = -1 \tag{112}$$

Subtracting Equation 112 from Equation 111:

$$\left(\boldsymbol{x}^{(+)} - \boldsymbol{x}^{(-)}\right)\boldsymbol{w} = 2 \tag{113}$$

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

$$\frac{\left(\boldsymbol{x}^{(+)} - \boldsymbol{x}^{(-)}\right)\boldsymbol{w}}{||\boldsymbol{w}||} = \frac{2}{||\boldsymbol{w}||} \quad \text{where} \quad ||\boldsymbol{w}|| = \sqrt{\sum_j w_j^2} \tag{114}$$

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

$$\begin{aligned} \max_{\boldsymbol{w}} \quad & \frac{2}{||\boldsymbol{w}||} \\ \text{s.t.} \quad & w_0 + \boldsymbol{x}_i \boldsymbol{w} \geq 1 \quad \text{if } y_i = 1 \\ & w_0 + \boldsymbol{x}_i \boldsymbol{w} \leq -1 \quad \text{if } y_i = -1 \end{aligned} \tag{115}$$

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

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

$$\begin{aligned} \min_{\boldsymbol{w}} \quad & \frac{||\boldsymbol{w}||^2}{2} + C\Big(\sum_i \xi_i\Big) \\ \text{s.t.} \quad & w_0 + \boldsymbol{x}_i \boldsymbol{w} \geq 1 - \xi_i \quad \text{if } y_i = 1 \\ & w_0 + \boldsymbol{x}_i \boldsymbol{w} \leq -1 + \xi_i \quad \text{if } y_i = -1 \end{aligned} \tag{116}$$

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

J K-Nearest Neighbors

K-Nearest Neighbors is a type of lazy learner [9] 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

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

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

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

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

The choice of K is important to find a good trade-off between over- and under-fitting [10]. 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:

$$d(\boldsymbol{x}_{i1}, \boldsymbol{x}_{i2}) = \sqrt{\sum_j |x_{i1}^j - x_{i2}^j|^2} \tag{117}$$

Figure 22 illustrates the learning rule.

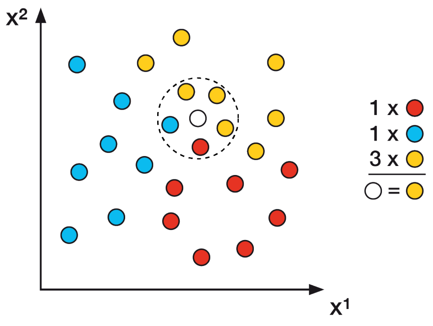


Figure 22: K-Nearest Neighbors learning rule

K Decision Tree

Decision Tree is an algorithms for binary and multi-class classification, which significantly facilitates interpretability by breaking down the data based on a series of questions, which are learnt from the features in the training set. More precisely, the learning algorithm is an iterative process, which starts at the tree root and sequentially splits the data at each node based on the feature that results in the largest information gain, until the leaves are pure, i.e. they all contain samples that belong to the same class. Since this procedure can result in a very deep tree with many sequential nodes (which could potentially result in overfitting), the tree depth is usually capped to a maximum level. Figure 23 illustrates the concept of a Decision Tree.

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

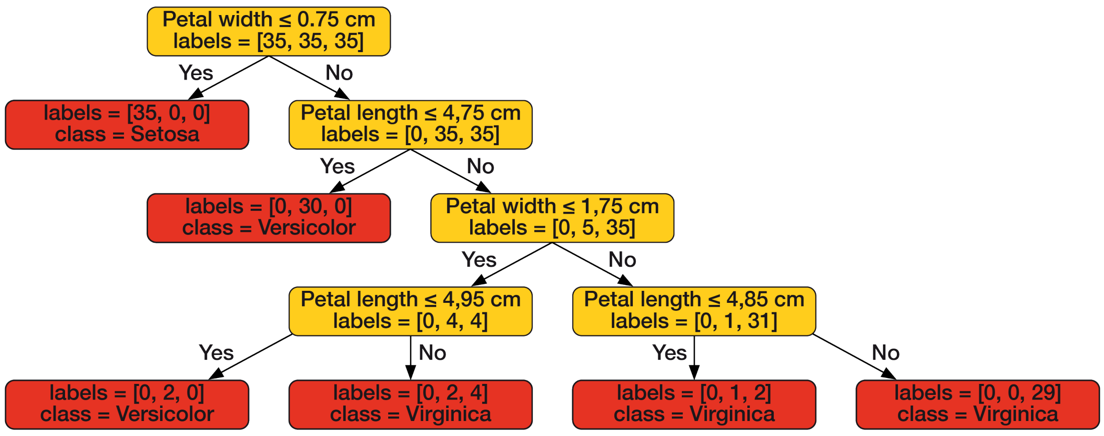


Figure 23: Decision Tree Learning

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

$$IG(D_p, f) = I(D_p) - \sum_j \frac{N_j}{N_p} I(D_j) \tag{118}$$

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:

$$IG(D_p, f) = I(D_p) - \frac{N_{left}}{N_p} I(D_{left}) - \frac{N_{right}}{N_p} I(D_{right}) \tag{119}$$

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:

$$I_H(t) = -\sum p(i|t) log_2 p(i|t) \tag{120}$$

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:

$$I_G(t)=\sum_i p(i|t)(1-p(i|t))=1-\sum_i p(i|t)^2 \tag{121}$$

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

The classification error is defined as follows:

$$I_E(t)=1-max\{p(i|t)\} \tag{122}$$

L 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 [11] is intended to provide all programming and computing setup to fully replicate the study [12]: 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.

[11] available at `github.com/nicoloceneda/Thesis`

[12] a valid WRDS account is needed

References

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

[2] 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.

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

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

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

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

[7] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. “Learning to forget: Continual prediction with LSTM”. In: (1999).

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

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

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

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

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

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

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

[15] 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.

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

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

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

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

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

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

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

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

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

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

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

[27] 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.

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

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

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

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

[32] 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.

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