University of Thessaly

Department of Electrical and Computer Engineering

Deep Learning Frameworks' Effectiveness and Efficiency for Time Series Prediction

by

Tryfon Tsakiris

A thesis submitted for the partial fulfillment of the requirements for the bachelor degree in Electrical and Computer Engineering

Supervised by

Dimitrios Katsaros, Assistant Professor
Lefteris Tsoukalas, Professor

Περίληψη

Η αποτελεσματικότητα και ταχύτητα πρόβλεψης των τιμών μιας χρονοσειράς είναι σημαντική σε μια πληθώρα περιοχών όπως στα χρηματιστήρια, σεισμικές μελέτες και δίκτυα αισθητήρων. Ο στόχος της παρούσας πτυχιακής εργασίας, είναι να αντιμετωπίσει αυτό το συγκεκριμένο πρόβλημα χρησιμοποιώντας τεχνικές βαθιάς μάθησης (deep learning) και να διευρευνήσει την αποτελεσματικότητα (ακρίβειας) πρόβλεψης και την ταχύτητα εκπαίδευσης και πρόβλεψης των κυριοτέρων βιβλιοθηκών βαθιάς μάθησης (deep learning). Εστιάζουμε στα *"Long Short-Term Memory" (LSTM)* και *"Gated Recurrent Unit" (GRU)* νευρωνικά δίκτυα που είναι παράγωγα των *"Recurrent Neural Networks" (RNN)* και είναι ειδικά σχεδιασμένα για να χειρίζονται ακολουθίες δεδομένων, όπως μια χρονοσειρά. Η υλοποίηση των μοντέλων πρόβλεψης έγινε με χρήση της υψηλού επιπέδου βιβλιοθήκης βαθιάς μάθησης *Keras*, η οποία επιτρέπει την χρήση διαφορετικών πιο χαμηλού επιπέδου βιβλιοθηκών βαθιάς μάθησης (*Tensorflow, Theano, CNTK*) ως την υποστηρικτική μηχανή της, η οποία καθιστά εφικτή την δίκαιη σύγκριση μεταξύ τους.

Abstract

The speed and efficiency of predicting the values of a time series is important in a variety of areas such as stock exchanges, seismic studies and sensor networks. The aim of this dissertation is to tackle this specific task with deep learning and explore the prediction efficiency and the speed of training and prediction of the popular deep learning frameworks and libraries. Focus is put on *"Long Short-Term Memory" (LSTM)* and *"Gated Recurrent Unit" (GRU)* networks, variants of *"Recurrent Neural Networks" (RNN)* which are specifically designed to handle sequences, like time series. The forecasting models are built with *Keras*, a high-level deep learning library, that allows the use of different low-level deep learning frameworks (*Tensorflow, Theano, CNTK*) as its backend engine, which makes it feasible to make a fair comparison between them.

Acknowledgments

With the completion of my diploma thesis, i would like to express my sincere gratitude to my friends and everyone else, who contributed in their own way and helped me reach at this point.

First and foremost, i would like to thank my thesis supervisor, Assistant Professor Dimitrios Katsaros, who gave me the chance to work on this project and for all his valuable help and guidance during the development of this thesis. I would also like to thank my second supervisor, Professor Lefteris Tsoukalas, for his participation and support.

Last but by no means least, i would like to thank my beloved family for their unceasing and unparalleled love, as well as for the continued moral and financial support throughout my studies. None of this could have happened without them.

Contents

List of Figures

List of Tables

1 Introduction

One of the most challenging and important field in data science is time series forecasting. Almost every business needs to predict the future and make better and more effective decisions. Some examples of time series forecasting use cases are: financial forecasting, product sales forecasting, energy demand forecasting for servers and buildings, internet traffic flow forecasting and many more. The common point to their problem is to find a way to use historical information in order to predict the future.

Traditionally, there exist several classical models for time series forecasting such as Autoregressive (AR), Moving Average (MA), Autoregressive Moving Average (ARMA) and Autoregressive Integrated Moving Average (ARIMA). The basic assumption made by these models is that the time series follow a specific distribution and that it can be represented by a linear equation. However, time series in general behave non-linearly and these classical models are very hard to be applied effectively.

In recent years, artificial neural networks (ANNs) and deep learning in particular, have become widely popular and are used almost everywhere to solve problems in many different areas such as speech and image recognition, natural language processing, medical research and time series forecasting. These deep learning algorithms have provided new approaches for prediction problems where the underlying relationships between variables are modeled deep and layered hierarchically. The most prominent characteristic of ANNs is their inherent capability of modeling non-linearities without any assumptions about the statistical distribution of the observations.

Of all the different deep learning architectures the most suitable for time series forecasting would be one that can process sequential data and maintain information about the past. An architecture that fulfills these requirements, is the Recurrent Neural Network (RNN) and in particular its variants: Long-Short Term Memory(LSTM) and Gated Recurrent Unit (GRU). These have shown state-of-the-art results in many applications such as machine translation, hand writing and speech recognition [1, 2, 3].

Finally, with the advent of many successful use cases the number of deep learning frameworks has been growing rapidly. Each framework is built in a different manner and for different purposes. Due to this variety its often necessary to decide which is the most beneficial deep learning framework for a specific task. In this thesis, we will explore and compare the performance of Tensorflow, Theano and CNTK as the backend engine of Keras for time series prediction.

2 Time Series

2.1 Definition

A time series is a sequence of data points (observations) measured over time and arranged in chronological order. It can be either *continuous* or *discrete*, but most commonly, samples are taken at equally spaced points in time, thus having a sequence of discrete-time data. Mathematically, it can be expressed as a set of vectors X_t, each one being observed at a specified time t:

$$X = \{X_1, X_2, ..., X_T\} \text{ or } \{X_t : t \in T\}$$

Additionally, time series can be categorized as *"univariate"* or *"multivariate"* depending on the number of variables into consideration. A *univariate* time series is a sequence of observations of a single variable, such as data collected from a sensor measuring the temperature of a room. If that sensor measured the humidity as well and we were interested in the joint behavior of these two variables then the time series would be characterized as *multivariate*.

2.2 Components

Time series in general is composed of three components namely the Trend (T), Seasonality (S) and Irregular (I).

1. **Trend** (T): Trend is the long-term pattern of the time series. It can be either positive or negative depending on whether the time series exhibits an increasing or decreasing pattern.

2. **Seasonality** (S): Seasonality occurs when there is a variation in the time series which is seasonal and fixed (e.g. the same month every year or day in the week).

3. **Irregular** (I): The irregular component is the residual time series after trend and seasonality are removed from the original time series. This component is usually unpredictable and random.

There are two ways of how these components can be combined together mathematically in order to compose the time series.

- Additive Model: $X_t = T_t + S_t + I_t$
- Multiplicative Model: $X_t = T_t \times S_t \times I_t$

2.3 Stationarity

A time series is stationary if its statistical properties like mean or variance are constant over time. Most of the traditional forecasting methods, assume that the distribution of the time series values is stationary.

Non-stationary data, are unpredictable and cannot be modeled or predicted correctly. Thus, in order to deal with the forecasting task these traditional methods needed to convert the non-stationary time series into stationary ones by removing the *trend* and *seasonality* components with the appropriate pre-processing techniques.

2.4 Forecasting

Time series prediction or forecasting, refers to the process of using the past observations of a time series in order to calculate one or several values ahead in the future. This effectively means that we need to find the functional dependency that holds the relationship between past and future values. Thus, we need to find a model which approximates a function f in order to predict h future values denoted as $\hat{X}$, by using w past values of the time series. We refer to h as the *forecasting horizon* and it indicates how far into the future we should predict. Alternatively, h and w can be referred as *lookahead* and *lookback*.

2.4.1 One-Step Ahead Forecasting

In one-step ahead forecasting we are interested in predicting only a single future value by using past observations. The forecasting horizon in this case is equal to one ($h = 1$) and the equation for one-step ahead forecasts is given below:

$$\hat{X}_{t+1} = f(X_t, X_{t-1}, ..., X_{t-w+1}) \tag{2.1}$$

2.4.2 Multi-Step Ahead Forecasting

A multi-step ahead time series forecasting task consists of predicting more than a single value in the future by using past observations, hence having a forecasting horizon $h > 1$. There are three strategies that are commonly used for multi-step ahead forecasting, namely the *Iterative*, *Direct* and *MIMO* approach [4].

Iterative Approach

This is the simplest and most intuitive forecasting strategy where a single model f is trained to perform one-step ahead forecasts (Equation 2.1). In order to forecast h steps ahead, the model f is used multiple times by having the recently forecasted values denoted as $\hat{X}$ as inputs to predict the next steps.

$$\hat{X}_{t+h} = \begin{cases} f(X_t, X_{t-1}, ..., X_{t-w+h}), & \text{if } h = 1 \\ f(\hat{X}_{t+h-1}, ..., X_t, ..., X_{t-w+h}), & \text{if } h \in \{2, ..., H\} \end{cases} \tag{2.2}$$

Direct Approach

In the direct approach in order to forecast h steps ahead a different model is trained for each horizon independently. In other words, h models need to be constructed, one for each horizon, based on the observed time series data.

$$\hat{X}_{t+h} = f_h(X_t, X_{t-1}, ..., X_{t-w+1}), \ where \ h \in \{1, ..., H\} \tag{2.3}$$

MIMO Approach

In this approach a model f is trained in order to produce multiple outputs from the historical values of the time series. The forecasts are returned in one shot manner by this multiple-output model f.

$$[\hat{X}_{t+h}, \hat{X}_{t+h-1}, ..., \hat{X}_{t+1}] = f(X_t, X_{t-1}, ..., X_{t-w-1}) \tag{2.4}$$

Finally, it should be noted that the Iterative approach will have degraded performance as it suffers from the accumulation of errors in multi-step ahead forecasts. This is due to the fact that predictions are used in place of real observations and errors are propagated for longer as the forecasting horizon increases. Also, the Direct approach involves heavier computational load, as it needs to train h different models.

2.4.3 Forecasting Accuracy

In order to evaluate the performance of the forecasting model we need to quantify and measure the deviation of the predictions from the real measurements. This deviation is called the forecast error and is defined as the difference between the actual value and the forecast value. If the error is denoted as E_t, the actual value as X_t and the forecasted value as $\hat{X}_t$, then the forecast error can be written as:

$$E_t = X_t - \hat{X}_t \tag{2.5}$$

Some of the most popular forecasting errors measures are:

- Mean Absolute Error (MAE):

$$MAE = \frac{1}{N}\sum_{t=1}^{N}|E_t| \tag{2.6}$$

- Mean Squared Error (MSE):

$$MSE = \frac{1}{N}\sum_{t=1}^{N}E_t^2 \tag{2.7}$$

- Root Mean Squared Error (RMSE):

$$RMSE = \sqrt{\frac{1}{N}\sum_{t=1}^{N}E_t^2} \tag{2.8}$$

- Mean Absolute Percentage Error (MAPE):

$$MAPE = \frac{100\%}{N}\sum_{t=1}^{N}\left|\frac{E_t}{X_t}\right| \tag{2.9}$$

- Symmetric Mean Absolute Percentage Error (SMAPE):

$$SMAPE = \frac{100\%}{N}\sum_{t=1}^{N}\frac{|E_t|}{(|X_t| + |\hat{X}_t|)/2} \tag{2.10}$$

From the aforementioned metrics, only *MAE* and *RMSE* are scale dependent metrics, which means that they are in the same scale as the data. Additionally, *MSE* and *RMSE* have the property to penalize large errors more than others due to the involved squaring operation.

Finally, *MAPE* and *SMAPE* are percentage errors which makes them easy understandable and can be good metrics for comparisons between different datasets. The former, though has many limitations, one of which is that it cannot be used if there are zero values, as this results in a division-by-zero which yields undefined results. The latter, is an improved version and has been used in many forecasting competitions.

3 Artificial Neural Networks

Artificial neural networks (ANNs) are machine learning models inspired by the structure and function of the brain. They try to replicate how human brain learns and deals with complex problems.

Generally, a neural network consists of several interconnected computational units called *artificial neurons* or just *neurons* and a set of directed edges between them with assigned weights. These computational units are organized in layers, specifically the *Input layer*, one or more *Hidden layers* and finally the *Output layer*. The outputs of *neurons* are passed from one layer to the next and information flows through the network. A typical neural network is depicted in Figure (3.7).

1. **Input layer**: Its main task is to bring the raw input data into the network for further processing.

2. **Hidden layers**: Is responsible for computations on the given data from the input layer and transferring this information to the next layer and eventually to the output layer.

3. **Output layer**: This is the last layer of a neural network and is responsible for performing computations and transformations on the data that came from the previous layer, in order to produce the end result.

Ultimately, just as biological neural networks learn their proper responses to given inputs artificial neural networks need to do the same. This procedure is called *learning* or *training* and its purpose is to teach the artificial neural network how to respond to given inputs.

3.1 Artificial Neuron

An artificial neuron is the basic building block of every artificial neural network. It accepts a set of inputs x, multiplies them with the appropriate weights w, adds the bias b and then passes this weighted sum to a linear or non-linear *activation function* to produce an output.

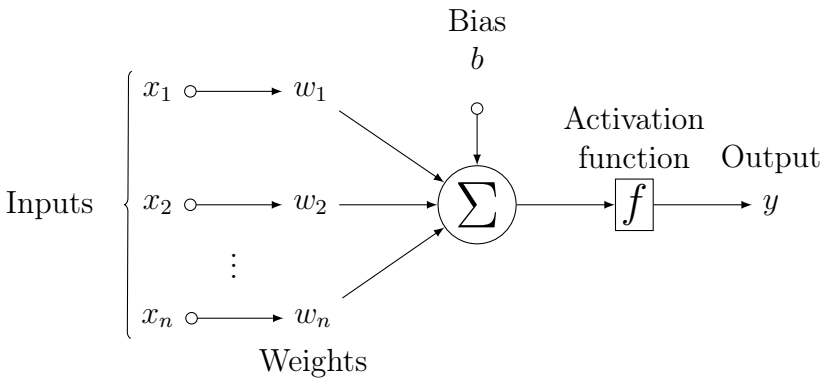


Figure 3.1: Artificial Neuron Model.

The output of the neuron, depicted in Figure (3.1) is given by the following equation:

$$y = f(\sum_{i=1}^{n} w_i \cdot x_i + b) \tag{3.1}$$

where f is the activation function with typical choices like sigmoid, tanh or ReLU.

3.2 Activation Functions

Activation functions are essential for a neural network to learn and make sense of something really complicated. They introduce non-linearity in the network by converting the input signal to an output signal via a transformation function. Some of the most popular activation functions are mentioned below.

3.2.1 Sigmoid

Sigmoid is one of the most widely used activations functions for the output layer. It has a characteristic "S-shaped" curve shown in Figure (3.2). It takes as input a real number and outputs a value in the range of [0, 1]. Hence, outputs can easily be interpreted as probabilities.

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

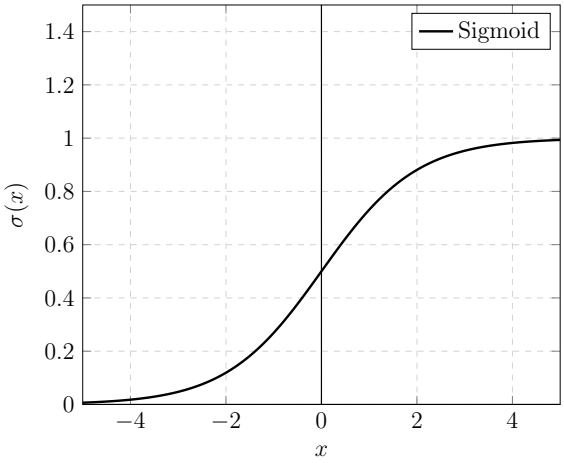


Figure 3.2: Sigmoid activation function.

3.2.2 Tanh

Tanh is similar to the sigmoid but has an output range of [-1, 1] which in turn means, the output values are zero-centered. Thus, strongly negative inputs to the tanh will map to negative outputs and zero-valued inputs will map to near-zero outputs. Hence, it can overcome the vanishing gradient problem.

$$f(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.3}$$

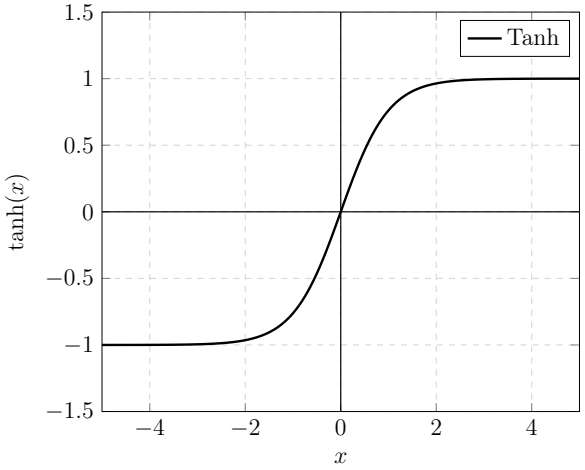


Figure 3.3: Hyperbolic tangent activation function.

3.2.3 ReLU

One of the most popular and widely used activation function is the rectified linear unit (ReLU). This function whilst simple, it is proven to be very effective with fast convergence and it is mainly used as an activation function for the hidden layers.

$$f(x) = \max(0, x) = \begin{cases} 0, & \text{if x} < 0 \\ x, & \text{if x} \geq 0 \end{cases} \tag{3.4}$$

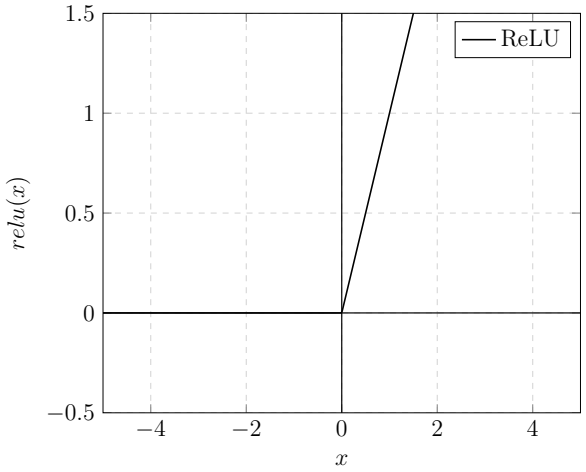


Figure 3.4: Rectifier Linear Unit activation function - ReLU.

3.3 Network Training

Training a neural network simply means to adjust the weights of the connections between *neurons*. At the beginning, the weights of the connections are initialized randomly. This obviously means that the neural network is going to produce bad results. Thus, in order for the neural network to be used adequately for the required task, it needs to be trained to find the proper weights.

3.3.1 Backpropagation

The most popular algorithm used for the training of a neural network is *Backpropagation*. This algorithm requires a measure of the network's output error which is given by the *loss function* denoted as $\mathcal{L}$. Its ultimate goal is to minimize this loss function $\mathcal{L}$ in weight space, by using a *gradient* based optimizer such as *Stochastic Gradient Descend (SGD)*, *Adam*, *RMSProp* or

others [5]. The general idea of gradient descent based minimizers, is to iteratively take steps proportional to the negative of the gradient of the target function $\mathcal{L}$.

Backpropagation algorithm can be summarized in this three steps:

1. **Forward Pass:** The input is given to the network via the input layer. These inputs are propagated through the network and their activations are computed and stored at each layer.

2. **Error Calculation:** After the forward pass, the network uses the loss function $\mathcal{L}$ to compute the error by comparing the prediction with the target value (supervised learning), as a means to estimate its performance. If *MSE* is used as the loss function then $\mathcal{L}$ is given by the following equation:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N}(output_i - target_i)^2 \tag{3.5}$$

   where N denotes the number of neurons in the output layer.

3. **Backward Pass:** At this step the partial derivative of the loss function $\mathcal{L}$ with respect to the network weights is computed and then an update procedure is initiated which adjusts the weights of the neurons in each layer in a backward manner. Each weight is updated using the following equation:

$$W_{new} = W_{old} - \gamma\frac{\partial\mathcal{L}}{\partial W_{old}} \tag{3.6}$$

   where γ represents the learning rate, i.e., a hyper-parameter that controls the magnitude of the weight updates. Finally, it should be noted that in order to effectively compute the partial derivatives for the backward pass, the chain rule is used:

$$\frac{\partial x}{\partial z} = \frac{\partial x}{\partial y}\cdot\frac{\partial y}{\partial z} \tag{3.7}$$

3.3.2 Supervised Learning

In this type of learning, the network receives both the inputs and the output targets usually called the (*ground truth*) of each training set. Thus for each input, the output of the network, can be compared with the expected result and the network's weights can be adjusted accordingly.

3.3.3 Unsupervised Learning

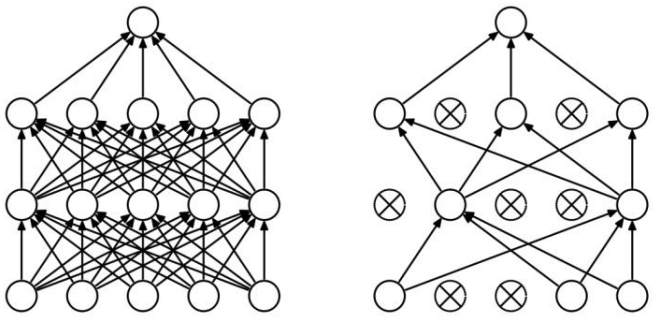
The goal of unsupervised learning is to infer useful information or patterns from a dataset without any guidance. It is called unsupervised because unlike supervised learning, the network is not given any labeled data, which means that there is no correct answer to compare the prediction.

3.4 Regularization

Regularization is a technique used in deep learning in order to prevent the model from *over-fitting*. *Over-fitting* occurs when the neural network tries to model the *training-data* too closely in an extent, that impacts negatively its ability and performance for predictions on new data. Out of all the available methods in the literature we list only the ones that are used on this thesis.

3.4.1 Dropout

Dropout [6] is a technique where a random fraction of neurons are *"dropped"* or *"cut-off"* during the training phase. In essence, this means that at each training step, randomly selected *neurons* are kept with probability p or removed from the network along with their incoming and outgoing connections, with probability $q = 1 - p$.



(a) Standard Neural Net (b) After applying dropout.

Figure 3.5: Neural network model with Dropout, in the right image crossed neurons are dropped. (Source: [7]).

3.4.2 Early Stopping

Early stopping is another technique used in deep learning to prevent the model from overfitting. Essentially, in this technique we need to keep a portion of our training dataset as a validation set. Then we need to constantly monitor the validation set error and whenever this error improves, we store a copy of the models parameters. Hence, when the training algorithm stops we return to the saved parameters rather than the latest ones. The stopping criteria for the training algorithm is when the model stops improving on the validation set i.e. validation error increases, for a specified number of epochs or iterations, usually called the *"patience"*.

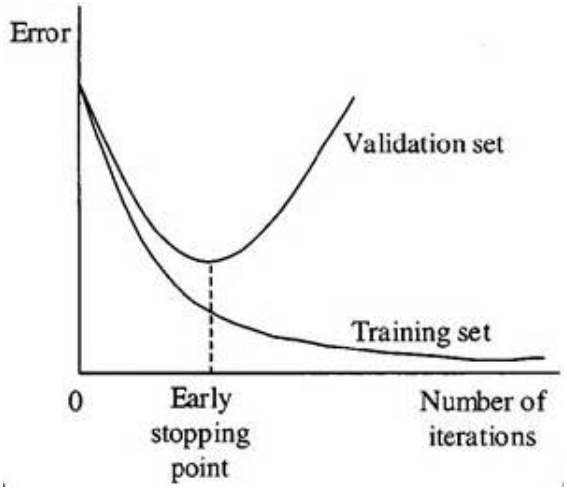


Figure 3.6: Early stopping technique to stop training exactly when validation set error starts to increase.

3.5 Neural Network Architectures

As mentioned earlier artificial neural networks are organized in layers composed of *neurons*. Depending on the problem at hand, connections can be made from neurons of one layer to another or/and between *neurons* at the same layer. This provides a flexibility to the neural network, allowing it to better adapt at a specific task.

3.5.1 Feedforward Neural Networks (FNN)

The Feedforward neural network (FNN) depicted in Figure (3.7) below, is the simplest type of artificial neural networks. In this type of networks information strictly flows in one direction, from the input-layer to the output-

layer. There aren't any feedback loops in the network, which in turn means that the output of every layer doesn't affect the same layer.

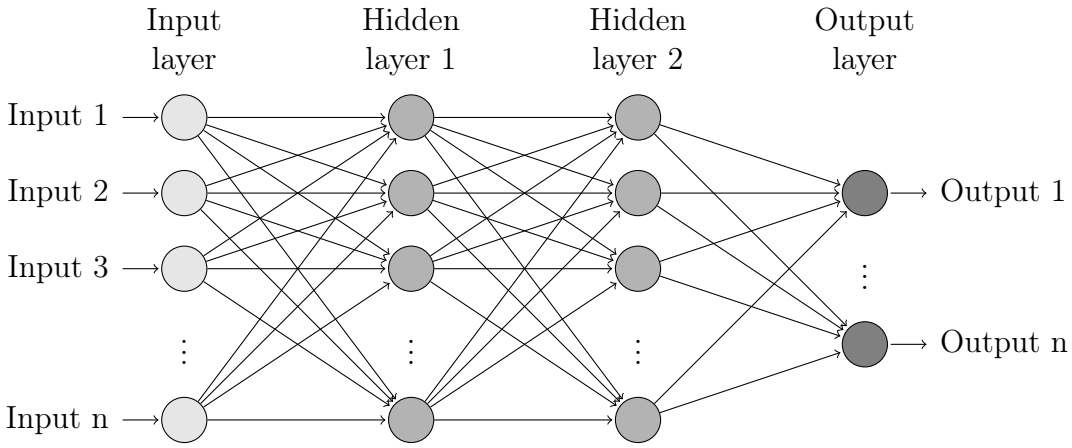


Figure 3.7: Feedforward neural network.

3.5.2 Recurrent Neural Networks (RNN)

Conventional neural networks make the assumption that every input element is independent from each other. This means that after processing one element there isn't any kind of information maintained, thus making them unsuitable for sequential data like time series.

A *recurrent neural network* (RNN) is a network that can overcome this limitation. More specifically, this network has *feedback loops* in it, making it capable of preserving historical information from previous timesteps.

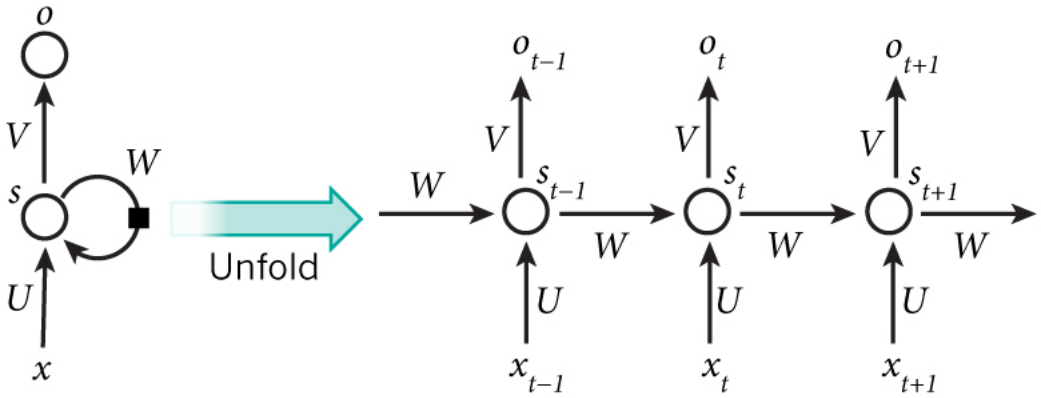


Figure 3.8: A recurrent neural network and its unfolded version to show how the network's *memory* is built (Source: [8]).

In general the RNN, takes as input a sequence of vectors $x_t = \{x_1, x_2, ..., x_T\}$ computes the hidden states $s_t = \{s_1, s_2, ..., s_T\}$ and finally outputs $o_t = \{o_1, o_2, ..., o_T\}$ for every time step t, $t \in \{1, 2, ..., T\}$. The hidden states represent the *memory* of the network and are computed by passing the weighted input at the current timestep x_t with the weighted hidden state of the previous timestep s_{t-1} through an activation function f.

This procedure can be described mathematically as:

$$s_t = f(Ux_t + Ws_{t-1} + b_s) \tag{3.8}$$

$$o_t = f(Vs_t + b_o) \tag{3.9}$$

where U, W, V are the weight matrices which are the same for each timestep, b_* are the biases and f is a non-linear activation function.

A recurrent neural network can be unfolded as depicted in Figure (3.8) and can be treated as a standard feedforward network that can be trained in a way similar to backpropagation. This approach is called Backpropagation Through Time (BPTT) [9]. Furthermore, even though RNNs can model sequence data decently they are not capable of maintaining long-term dependencies [10].

3.5.3 Long-Short Term Memory Networks (LSTM)

In order to overcome the aforementioned problem in the training phase of the RNN, Hochreiter and Schmidhuber introduced the LSTM architecture [11]. This architecture later got many updates and refinements from many researchers. One of the most notable additions to the vanilla architecture of the LSTM, was that of Schmidhuber, Gers and Cummins which included the *forget* gate [12].

The LSTM architecture is very similar to that of the RNN but with the exception that each neuron is replaced by an LSTM unit —Figure (3.9), which is a composite unit composed of a *memory cell* and three *gates*: input, output and forget gates.

1. **Forget Gate:** The forget gate applies a sigmoid function on the weighted sum of the input in the current timestep x_t and the output of the previous step h_{t-1} and outputs a value between [0, 1]. This effectively decides, which information is going to be removed from the cell state. A value of 0 indicates that the information should be completely removed and a value of 1 exactly the opposite.

$$f_t = \sigma(W_f x_t + R_f h_{t-1} + b_f) \tag{3.10}$$

2. **Input Gate:** The input gate applies a sigmoid function on the weighted sum of the input in the current timestep x_t and the output of the previous step h_{t-1}, to ascertain which values to be updated on the cell state C_t. This information is combined with the new candidate values for the cell state, denoted as $\widetilde{C}_t$, by passing x_t and h_{t-1} through a tanh function.

$$i_t = \sigma(W_i x_t + R_i h_{t-1} + b_i) \tag{3.11}$$

$$\widetilde{C}_t = tanh(W_c x_t + R_c h_{t-1} + b_c) \tag{3.12}$$

3. **Memory Cell:** The memory cell is updated by forgetting the information that is no longer required from the previous time step and the information that is relevant in the current timestep.

$$C_t = f_t \odot C_{t-1} + i_t \odot \widetilde{C}_t \tag{3.13}$$

4. **Output Gate:** The output gate applies a sigmoid function on the weighted sum of x_t and h_{t-1} in order to control what information should flow out of the LSTM unit. The result of this gate is then passed through a tanh function to produce the output of the LSTM unit h_t, at the current timestep t.

$$o_t = \sigma(W_o x_t + R_o h_{t-1} + b_o) \tag{3.14}$$

$$h_t = tanh(C_t) \odot o_t \tag{3.15}$$

It should be noted that in the equations above, $\odot$ is element-wise multiplication (Hadamard product) and W_*, R_* and b_* represent the input weights, the recurrent weights and the biases respectively.

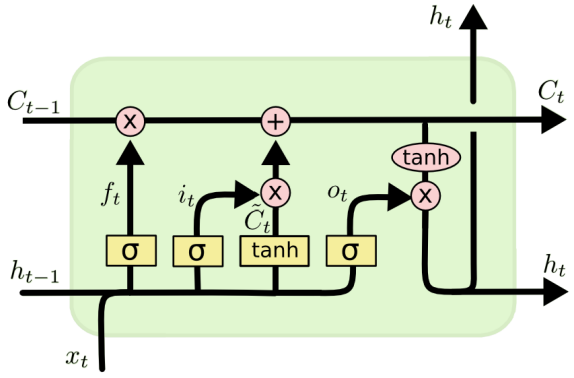


Figure 3.9: Internal structure of an LSTM unit (Source: Colah's blog [13]).

3.5.4 Gated Recurrent Unit (GRU)

The Gated Recurrent Unit was introduced by Kyunghyun Cho [14] and is another variant of RNNs that follows closely the architecture of the LSTM. Both variants, can track long-term dependencies while mitigating the problems of the training phase in RNNs [10].

Similarly to the LSTM unit, the GRU unit employs gating mechanisms to modulate the flow of information inside the unit. It combines the input and forget gates into a single *update gate* and has an additional *reset gate*. Additionally, the GRU does not maintain a *cell state* and exposes its full content without any adjustment. That being said, the GRU unit is simpler than the LSTM and has fewer parameters which makes it more computational efficient.

1. **Update Gate:** The update gate z_t decides how much the unit updates its content or information from the previous step. It applies a sigmoid function to the weighted sum of the input at the current timestep x_t and the previous output h_{t-1}.

$$z_t = \sigma(W_z x_t + R_z h_{t-1}) \tag{3.16}$$

2. **Reset Gate:** The reset gate r_t decides how to combine the new input with the previous memory. Similarly to the update gate, it applies

a sigmoid function to the weighted sum of the input at the current timestep x_t and the previous output h_{t-1}.

$$r_t = \sigma(W_r x_t + R_r h_{t-1}) \tag{3.17}$$

3. **Output:** The output of the GRU unit h_t at time t is a linear interpolation between the previous output h_{t-1} and the candidate output $\tilde{h_t}$.

$$\tilde{h_t} = \tanh(W_h x_t + R_h(r_t \odot h_{t-1})) \tag{3.18}$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h_t} \tag{3.19}$$

It should be noted that in the equations above, $\odot$ is element-wise multiplication (Hadamard product) and W_*, R_* represent the input weights and the recurrent weights respectively.

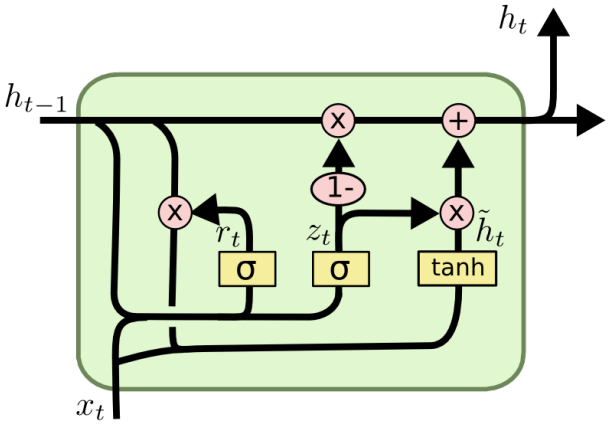


Figure 3.10: Internal structure of a GRU unit (Source: Colah's blog [13]).

4 Data Preprocessing Techniques

In deep learning, the raw input data have to undergo some transformations prior to inserting them to the network. This procedure has proven to significantly improve the convergence as well as the performance of the model [7].

4.1 Normalization

Normalization also known as "Min-Max Scaling" is the process of rescaling the data from the original scale into the desired scale, which is usually [0, 1] or [-1, 1]. Of course, a linear scaling like this requires that the minimum and maximum values can be found.

The equation to normalize the data in the range [a, b] is the following:

$$X'_t = (b-a)\frac{X_t - X_{min}}{X_{max} - X_{min}} + a \tag{4.1}$$

A variant of this method, called "Mean Normalization" subtracts the mean value μ, instead of the minimum. Thus, Equation (4.1) becomes:

$$X'_t = (b-a)\frac{X_t - \mu}{X_{max} - X_{min}} + a \tag{4.2}$$

4.2 Standardization

The goal of standardization is to rescale the input features, so that they have the properties of a standard normal distribution, i.e. zero mean and unit standard deviation, $\mu = 0$ and $\sigma = 1$. Of course, this method requires that the mean and standard deviation of the observable values can be found.

The equation to standardize the data is the following:

$$X'_t = \frac{X_t - \mu}{\sigma} \tag{4.3}$$

where μ is the mean and σ is the standard deviation of the data.

$$\mu = \frac{1}{N}\sum_{t=1}^{N} X_t \tag{4.4}$$

$$\sigma = \sqrt{\frac{1}{N}\sum_{t=1}^{N}(X_t - \mu)^2} \tag{4.5}$$

5 Software Tools

5.1 Deep Learning Frameworks

A deep learning framework is an interface or library that allows developers to easily design, build and evaluate deep learning models, by using a collection of pre-built optimized components. There is a huge collection of deep learning frameworks one can use to develop neural networks, with the most popular listed below.

Tensorflow

Tensorflow is an open source library for high performance numerical computation using data flow graphs. Its flexible architecture allows easy deployment of computations across various platforms such as CPUs and GPUs. It was developed by Google Brain Team and is one of the most commonly used frameworks today.

Theano

Theano is a python library that allows you to define, optimize and evaluate mathematical expressions involving multi-dimensional arrays efficiently on either CPU or GPU architectures. It was developed by a research group at the University of Montreal and it was one of the first widely used deep learning frameworks.

Microsoft Cognitive Toolkit

Microsoft Cognitive Toolkit (CNTK) is an open-source software toolkit for commercial-grade distributed deep learning, developed by Microsoft. It describes neural networks as a series of computational steps via a directed graph. It also allows for easy integration with Azure Cloud Services.

Keras

Keras is a high level deep-learning library in Python that offers a very human-friendly and easily operated interface in order to build neural networks. It's capable of running on top of of either Tensorflow, Theano or Microsoft Cognitive Toolkit. Keras was developed and maintained by François Chollet and has a huge community and support.

5.2 Scientific Libraries

Python has a wide variety of libraries that transform it from a general purpose programming language into a very powerful tool for data analysis and visualization. Some of the core libraries used in this thesis and that make this feasible are listed below.

NumPy

NumPy is the foundational library for scientific computing in Python, and many of the most popular libraries use NumPy arrays as their basic inputs and outputs. In short, NumPy introduces objects for multidimensional arrays and matrices, as well as routines that allow developers to perform advanced mathematical and statistical functions on those arrays.

SciPy

SciPy is a collection of mathematical algorithms and high-level functions built on top of NumPy. This package provides efficient numerical routines such as numerical integration, optimization, linear algebra and many others via its submodules.

Scikit-learn

Scikit-learn is a python machine learning library that builds on NumPy and SciPy by adding algorithms for common machine learning and data mining tasks such as clustering, regression and classification.

Pandas

Pandas is a software library written for the Python programming language for data manipulation and analysis. In particular, it offers high-level data structures and operations for manipulating numerical tables and time series.

Matplotlib

Matplotlib is the standard Python library that is used to generate visualizations such as 2D plots and graphs.

6 Methodology

In this thesis, we decided to use Keras to build the deep learning models for the time series prediction task. In the subsequent sections we briefly explain all the required steps of our implementation.

6.1 Data Preparation

6.1.1 Data Split

In order to build the model and estimate its performance, we need to split the dataset into three subsets namely the *training, validation* and *test* set.

- **Training Dataset:** This set is used solely during the training procedure of the neural network in order to fit the parameters of the model (e.g. weights).
- **Validation Dataset:** This set is mainly used to provide an unbiased evaluation of the model fit on the training dataset. This set is used to tune the *hyperparameters* of the neural network (e.g. number of hidden units).
- **Test Dataset:** This set is comprised by previously unseen data and it is used to assess the predictive performance of the final model.

This method is called Holdout Validation and the general procedure one follows when doing this split on the dataset is depicted in Figure (6.1) below.

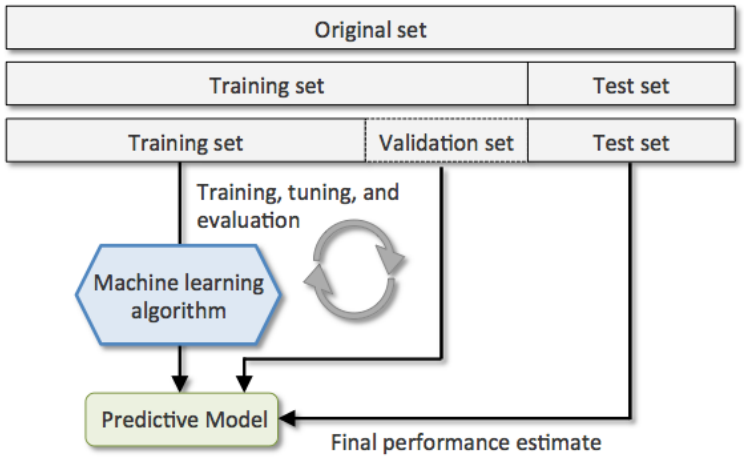


Figure 6.1: Holdout validation.

6.1.2 Supervised Learning

In order to train the neural network, the time series data needs to be reformed and framed as a supervised learning task. A common approach is to use a *sliding window* over the dataset in order to create input-output pairs. For instance, if we have the series $X = \{X_1, X_2, ..., X_{10}\}$ and we would like to use the last three observations (timesteps) in order to predict the next value in the series, then we need to produce the following sequences:

Input	Output
$[X_1, X_2, X_3]$	X_4
$[X_2, X_3, X_4]$	X_5
$[X_3, X_4, X_5]$	X_6
$[X_4, X_5, X_6]$	X_7
$[X_5, X_6, X_7]$	X_8
$[X_6, X_7, X_8]$	X_9
$[X_7, X_8, X_9]$	X_{10}

Table 6.1: Time series conversion to supervised learning.

6.1.3 Data Preprocessing - Transformation

Before we input the raw data in the network they have to undergo some transformations as mentioned in Section (4). For the LSTM network, all the input time series have been normalized in the range [-1, 1] to match the output range of the activation function tanh. This is accomplished by using Equation (4.1).

It should be noted that in order to evaluate the model predictions and gather metrics, the predictions need to be inverted back to the original scale. This is accomplished by using a reverse operation of Equation (4.1).

It is also really important to not leak any information when doing this operation from the test dataset during training. Thus, in order to normalize or standardize the time series, one should estimate the parameters of the equations only from the training dataset.

6.2 Keras Implementation Details

Working with LSTM in Keras is not that straightforward and requires some attention when building the model. In the following sections, we briefly discuss these tricky parts.

6.2.1 LSTM Input/Output

To start with, the LSTM layer requires the input data to be in the form of a three dimensional array with dimensions [batch_size, timesteps, features].

1. **Batch_size**: is the number of samples in a forward/backward pass before a weight update.

2. **Timesteps**: is the number of past observations to use (i.e. lookback or lags).

3. **Features**: is the number of features for every timestep (1 for univariate or n if multivariate).

The output of the LSTM layer by default is a 2D array with dimensions [batch_size, units] or [batch_size, timesteps, units] in case the argument *"return_sequences"* in the LSTM layer, is set to True. This option is mainly used when multiple LSTM layers are stacked together, in order for the next LSTM layer to obtain the whole sequence (timesteps) from the previous.

6.2.2 LSTM State

Another option in the LSTM layer is the parameter *"stateful"*. Even though, LSTM layers are inherently stateful, this option allows batches of samples to share the values of their hidden states. Thus, if this option is enabled the hidden state of i_{th} sample in the previous batch k will be provided as initial state to the i_{th} sample in the subsequent batch $k + 1$.

It should be noted though, that in case a network is stateful the batch_size has to be a divisor of the size of all the datasets (train, validation, test). Thus, an additional pre-processing step has to be made for the time series data that might require to truncate the last batches of the datasets.

Additionally, after the network is rendered stateful, there should be special care on when the model should reset the the states. It is common to reset the states after the whole sequence has been processed. Thus, for time series forecasting, we should reset the states after each epoch (i.e. a complete forward/backward pass of all the samples).

6.2.3 Callbacks

Keras allows the creation of callbacks in order to make changes to the network during training. In thesis, the following callbacks are created/used:

1. **EarlyStopping**: To monitor the validation loss and stop training when there is no improvement for a number of epochs.

2. **ModelCheckpoint**: This callback saves the weights as well as all the information regarding the training state of the model in a special file (HDF5 format).

3. **ReduceLROnPlateau**: This is used to reduce the learning rate of the optimizer in case there is no improvement in the validation loss for a number of epochs.

4. **ResetStateCallback**: This callback is mainly created to reset the states of the model at the end of each epoch (in case the model is stateful).

5. **CVSLogger**: Callback in order to store the evaluation metrics for each epoch, during training.

6.3 Hyperparameter Tuning

The whole complexity of the experiments with ANNs is the process of tuning the hyperparameters (e.g. number of neurons, number of layers, learning rate, epochs etc.). In our experiments we manually tuned the hyperparameters of the model via trial and error. For deeper and more complex networks, one can look into more time-consuming and exhaustive searching methods like grid-search, random-search or bayesian optimization [15, 16].

7 Experiments and Results

As mentioned earlier, our deep learning models are built with Keras. Keras allows the use of different backends (frameworks) such as Tensorflow, Theano and CNTK to handle all the low-level operations. In this section, we test the performance of the aforementioned frameworks in terms of training speed and prediction accuracy for out-of-sample forecasts (i.e. forecasts on the testing dataset). We will focus on univariate time series —Table (7.3) and a method for one-step ahead forecasts will be employed.

—*It should be noted, that all frameworks are CPU only variants and have been installed from the official sites (see Section 5.1) following the proposed guidelines and using the default configurations.*

7.1 Setup

The following tables contain the detailed specifications of the machine and the frameworks that are used in the experiments.

System Details	
Platform	Linux x86_64
Distribution	Ubuntu 16.04.5 LTS
CPU Model	Intel i7-4700MQ
Number of Cores	4
Number of Threads	8
CPU Frequency	2.40 - 3.40 GHz
Memory (RAM)	8 GB

Table 7.1: Machine specifications.

Framework	Version
Keras	2.2.2
Tensorflow	1.9.0
Theano	1.0.2
CNTK	2.5.1

Table 7.2: Deep learning frameworks version.

7.2 Datasets

The datasets used in the experiments are publicly available and their details are listed in Table (7.3) below.

Dataset Name	Abstract	Size	Attributes	Source
Beijing PM2.5 Dataset	This hourly data set contains the PM2.5 data of US Embassy in Beijing. Meanwhile, meteorological data from Beijing Capital International Airport are also included.	43824	13	UCI Machine Learning Repository
Internet traffic data (in bits) from an ISP	This dataset contains aggregated traffic in the United Kingdom academic network backbone. It was collected between 19 November 2004, at 09:30 hours and 27 January 2005, at 11:11 hours. Data is collected at five minute intervals.	19888	1	Time Series Data Library
Zürich monthly sunspot numbers	This dataset describes a monthly count of the number of observed sunspots for over 230 years (1749-1983).	2820	1	Time Series Data Library

Table 7.3: Datasets details.

The Beijing PM2.5 dataset contains also meteorological data, but we're interested in forecasting the PM2.5 concentration, thus we are going to drop all the other attributes. Furthermore, we can observe that the dataset contains some missing values hence, an additional pre-processing step has to be done to remove these rows. The Zürich and Internet traffic datasets, are left as is because there are no missing values.

7.2.1 Dataset Visualizations

In order to get a visual grasp of the patterns that govern the time series, the datasets are plotted over their whole range —Figures (7.1 - 7.3).

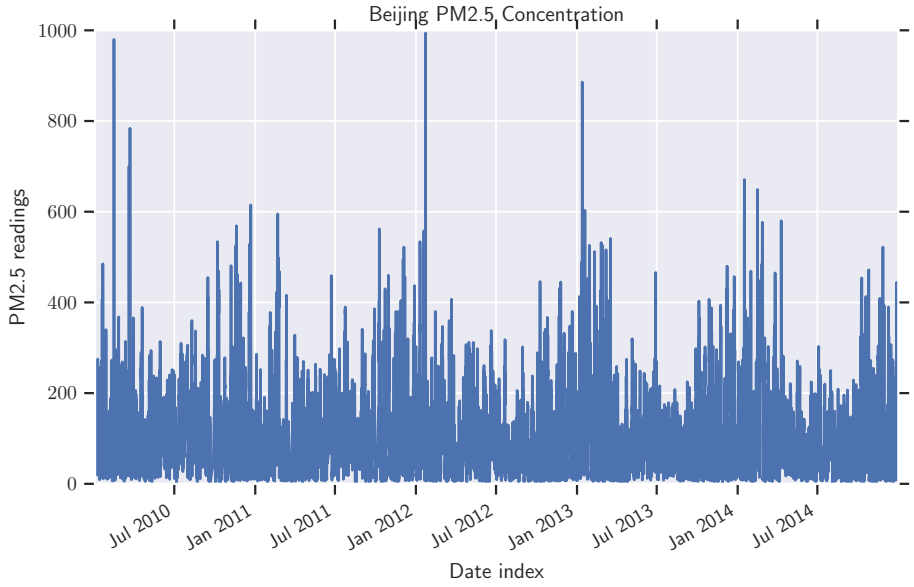


Figure 7.1: Beijing PM2.5 dataset visualization.

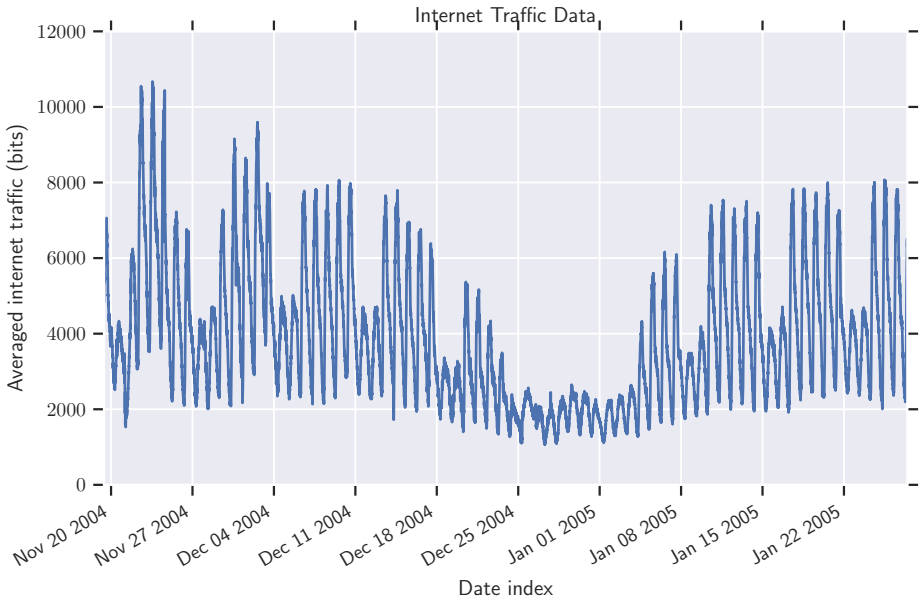


Figure 7.2: Internet traffic dataset visualization.

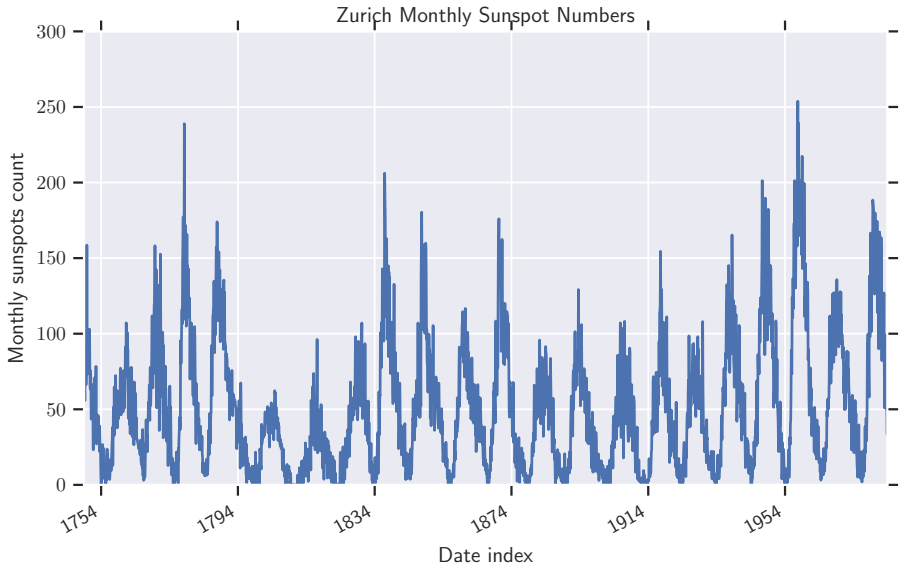


Figure 7.3: Zürich sunspots dataset visualization.

7.3 Model Details

The same model is used for all the datasets, in order to compare the scalability of the frameworks in terms of the length of the datasets. Additionally, after experimentation with the tuning procedure of the hyperparameters, we chose the ones that lead the model to produce the best prediction performance on all validation datasets.

The predictor is a stateful multi-layered model that consists of two stacked LSTM layers with 128 and 256 neurons respectively and a Dense layer (fully connected layer) with a single neuron to produce a single prediction (one-step ahead forecasting). The first LSTM layer returns as an output the whole sequence (timesteps), while the second one returns only the last step, thus dropping the temporal dimension. In-between layers we added a dropout layer of 0.2, in order to prevent the model from overfitting. The model is trained for 30 number of epochs with early-stopping and a batch size of 128. The objective loss function is MSE and RMSprop [17] is used as the network's optimizer with a learning-rate of 0.001. Furthermore, the learning rate is monitored and is reduced by a factor of 0.5 in case there is no improvement in the validation loss for 5 epochs. The time series have been converted to input-output pairs (supervised learning) with a number of 24 past timesteps and one output, with a process like the one depicted in Section (6.1.2) —Table (6.1). We use 70% of data for training, 10% for validation and the last 20% to test the prediction performance. The model's architecture as well as its input/output dimensions are depicted in Figures (7.4 - 7.5) below.

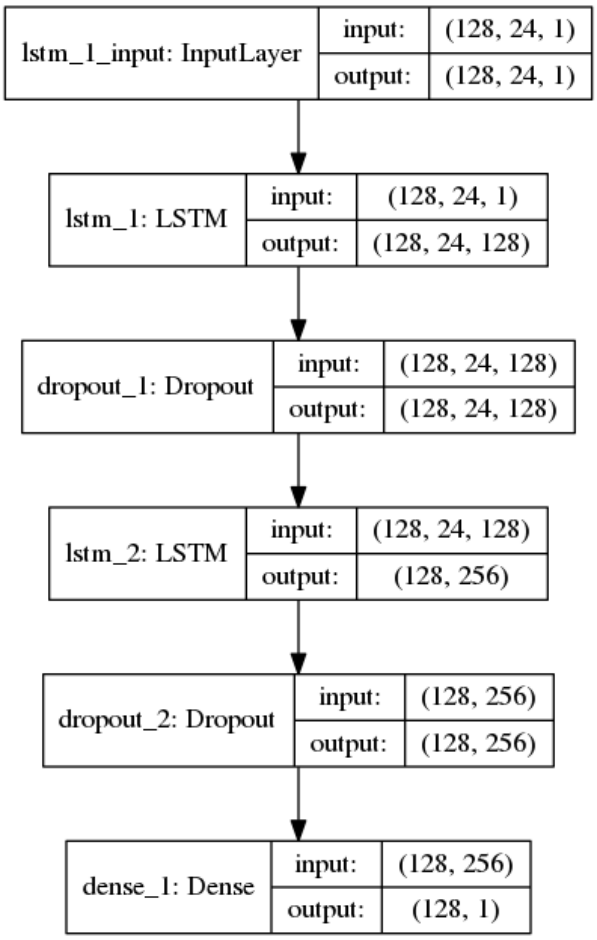


Figure 7.4: Model's architecture with each layer's I/O dimensions.

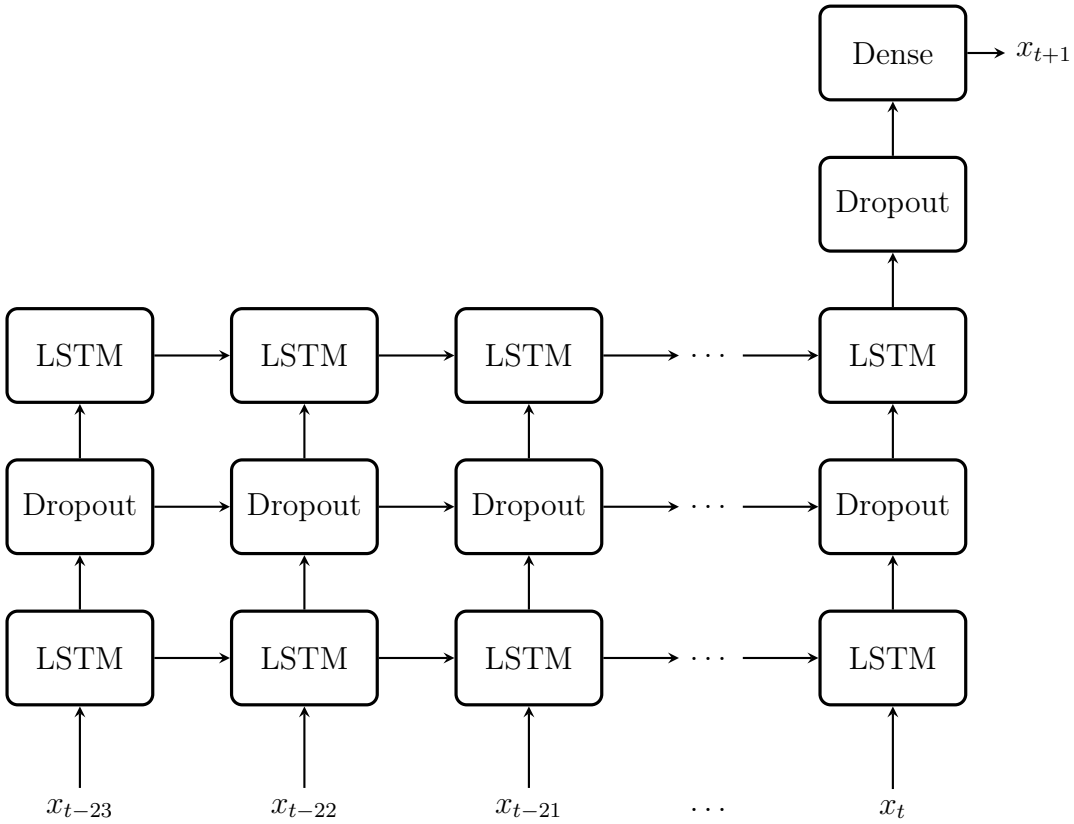


Figure 7.5: An illustration of the LSTM model. It takes as input the past 24 observations (timesteps) and produces an output sequence to the next layers. Finally, the output of the last cell of the second LSTM layer is used as input to the fully connected layer to produce a scalar output prediction. In-between layers we've introduced a Dropout layer in order to prevent the model from overfitting.

7.4 Evaluation Metrics and Results

To assess the performance of the frameworks for the LSTM model, we will use three different metrics, namely the Mean Absolute Error (MAE) Equation (2.6), the Root Mean Square Error (RMSE) Equation (2.8) and the time it took for the model to train on the training dataset for 30 epochs.

Dataset	Beijing PM2.5			Internet Traffic			Zürich Sunspots		
Metrics	MAE	RMSE	Time	MAE	RMSE	Time	MAE	RMSE	Time
Tensorflow	12.30	21.75	16m59s	62.15	84.56	7m29s	14.06	19.00	1m0s
Theano	11.88	21.59	27m14s	60.17	81.86	14m57s	14.16	19.23	2m7s
CNTK	12.19	21.85	33m48s	61.97	84.60	16m38s	13.43	18.28	2m14s

Table 7.4: Framework evaluation metrics of the LSTM for every dataset

Additionally, we trained with TensorFlow a GRU network as an exact replica of the LSTM, to compare their performance. From Table (7.5), it is clear that GRU achieves better score for the given datasets while being faster during training, indicating that it might be better candidate for modeling the time series.

Dataset	Beijing PM2.5			Internet Traffic			Zürich Sunspots		
Metrics	MAE	RMSE	Time	MAE	RMSE	Time	MAE	RMSE	Time
LSTM	12.30	21.75	16m59s	62.15	84.56	7m29s	14.06	19.00	1m0s
GRU	11.94	21.69	13m35s	57.91	78.76	6m31s	13.41	18.39	0m55s

Table 7.5: Tensorflow: LSTM vs GRU evaluation metrics for every dataset

7.5 Predictions Visualization

In the following subsections we will illustrate the first 300 out-of-sample predictions of the GRU model for every dataset. The GRU model was chosen because it achieved the best MAE and RMSE scores, as depicted in Table (7.5).

7.5.1 Beijing PM2.5 Dataset

Figure (7.6) depicts the first 300 continuous plots of one-step-ahead forecasted values and the actual values on the test dataset.

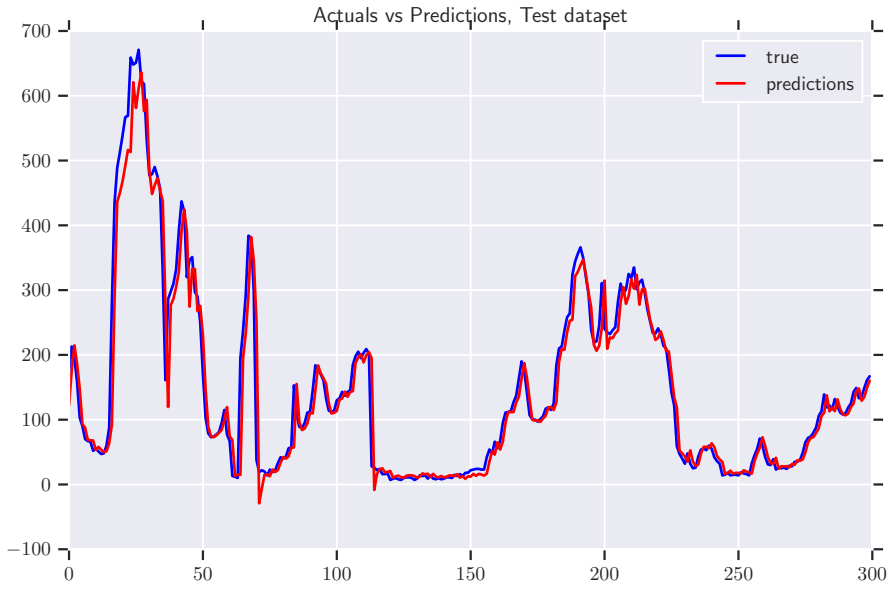


Figure 7.6: One-step-ahead predictions on the Beijing PM2.5 test dataset. Predictions are represented with red color and actual values with blue.

7.5.2 Internet Traffic Dataset

Figure (7.7) depicts the first 300 continuous plots of one-step-ahead forecasted values and the actual values on the test dataset.

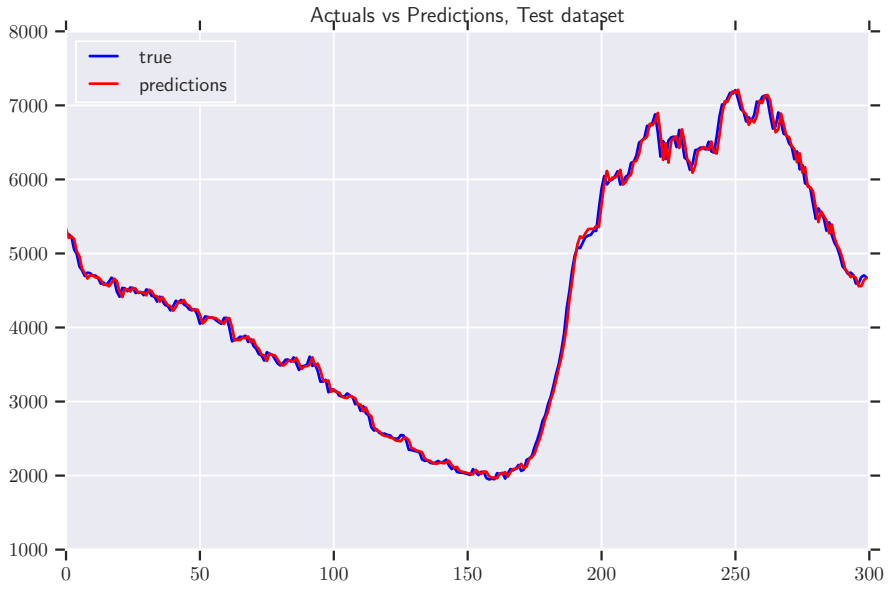


Figure 7.7: One-step-ahead predictions on the Internet traffic test dataset. Predictions are represented with red color and actual values with blue.

7.5.3 Zürich Sunspots Dataset

Figure (7.8) depicts the first 300 continuous plots of one-step-ahead forecasted values and the actual values on the test dataset.

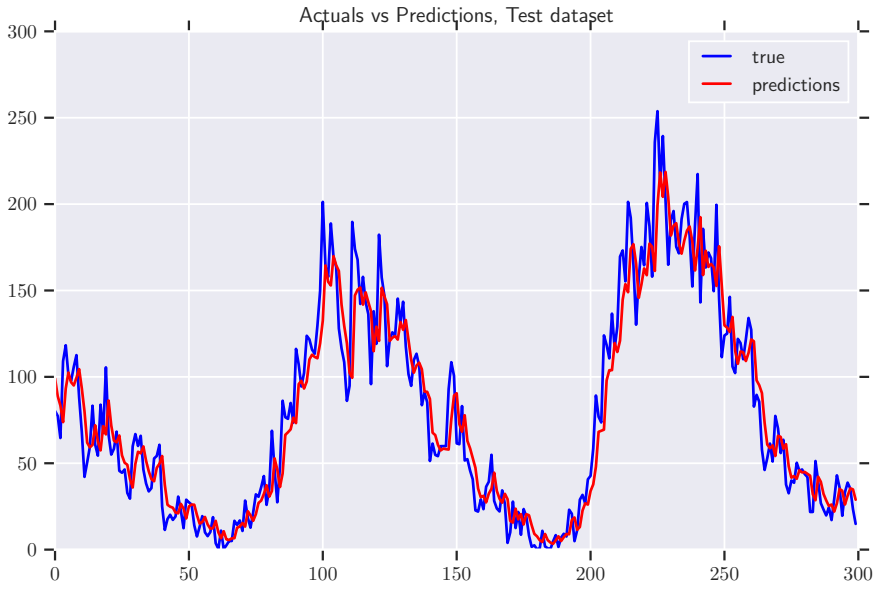


Figure 7.8: One-step-ahead predictions on the Zürich sunspots test dataset. Predictions are represented with red color and actual values with blue.

8 Conclusion

In this thesis we presented the relevant theory, of time series and deep learning, that is required in order to build a model to perform time series forecasting. Subsequently, we presented the state-of-the-art deep learning architectures for sequence based tasks (LSTM, GRU), their inner workings as well as methods that are widely used in deep learning. Additionally, we showed the steps of how to manipulate time series and frame them in a form that is suitable for the LSTM model's training. Furthermore, many of the implementation details of Keras are presented, which are required in order to build the LSTM, to feed the time series data to the LSTM as well as to monitor its training. Finally, these models were trained with the available deep learning frameworks that serve as the backend engine of Keras and comparisons were made in terms of prediction accuracy and training speed for different datasets and for different network architectures (LSTM - GRU).

For the last part, even though different frameworks use slightly different implementations of the mathematical operations, the results gathered from the experiments are approximately the same, except for the training speed. This is mainly due to the different optimizations of the mathematical libraries used in the background that handle the matrix and linear algebra operations. Additionally, from the experiments presented in Table (7.4) we can clearly see that CNTK is the slowest in terms of training speed, with Theano being a little bit faster than the former and TensorFlow being the fastest.

As of the comparison of the LSTM versus the GRU network, it seems that GRU is less computationally expensive than LSTM, mainly because of the fewer parameters which result in a faster training speed. It is also evident that the GRU can achieve better predictions than the LSTM, having better evaluation scores as indicated by the results presented in Table (7.5).

To sum up, it seems that the winner of the frameworks comparison is Google's TensorFlow, as it achieved approximately the same evaluation score, but it was approximately 2x faster than the others. Hence, to build LSTM and GRU models for time series prediction, TensorFlow seems to be the most suitable framework as the backend engine of Keras in a CPU (non-GPU) setup, by using the default configurations provided in the installation guidelines of each framework.

References

[1] Alex Graves, Marcus Liwicki, Santiago Fernández, Roman Bertolami, Horst Bunke, and Jürgen Schmidhuber. A Novel Connectionist System for Unconstrained Handwriting Recognition. *IEEE Trans. Pattern Anal. Mach. Intell.*, 31(5):855–868, May 2009. ISSN 0162-8828. doi: 10.1109/TPAMI.2008.137. http://dx.doi.org/10.1109/TPAMI.2008.137.

[2] Alex Graves, Abdel-rahman Mohamed, and Geoffrey E. Hinton. Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778, 2013. http://arxiv.org/abs/1303.5778.

[3] Biao Zhang, Deyi Xiong, and Jinsong Su. Recurrent neural machine translation. *CoRR*, abs/1607.08725, 2016. http://arxiv.org/abs/1607.08725.

[4] S. Ben Taieb, G. Bontempi, A. Atiya, and A. Sorjamaa. A review and comparison of strategies for multi-step ahead time series forecasting based on the NN5 forecasting competition. *ArXiv e-prints*, August 2011. https://arxiv.org/abs/1108.3259.

[5] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016. http://arxiv.org/abs/1609.04747.

[6] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014. http://jmlr.org/papers/v15/srivastava14a.html.

[7] Yann LeCun, Léon Bottou, Genevieve B. Orr, and Klaus-Robert Müller. Efficient BackProp. In *Neural Networks: Tricks of the Trade, This Book is an Outgrowth of a 1996 NIPS Workshop*, pages 9–50, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65311-2. http://dl.acm.org/citation.cfm?id=645754.668382.

[8] Yann LeCun, Y Bengio, and Geoffrey Hinton. Deep learning. In *Nature*, volume 521, pages 436–44, 05 2015. http://dx.doi.org/10.1038/nature14539.

[9] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, Oct 1990. ISSN 0018-9219. http://dx.doi.org/10.1109/5.58337.

[10] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, Mar 1994. ISSN 1045-9227. doi: 10.1109/72.279181. https://dl.acm.org/citation.cfm?id=2328340.

[11] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[12] Felix A. Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to Forget: Continual Prediction with LSTM. *Neural Computation*, 12: 2451–2471, 1999. https://dl.acm.org/citation.cfm?id=1121915.

[13] Christopher Olah. Understanding LSTM Networks. http://colah.github.io/posts/2015-08-Understanding-LSTMs/, 2015.

[14] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. *CoRR*, abs/1406.1078, 2014. http://arxiv.org/abs/1406.1078.

[15] Patrick Koch, Brett Wujek, Oleg Golovidov, and Steven Gardner. Automated hyperparameter tuning for effective machine learning. 2017. https://support.sas.com/resources/papers/proceedings17/SAS0514-2017.pdf.

[16] Jeremy Jordan. Hyperparameter tuning for machine learning models. https://www.jeremyjordan.me/hyperparameter-tuning/, 2017.

[17] T. Tieleman and G. Hinton. Lecture 6e—RmsProp: Divide the gradient by a running average of its recent magnitude. Coursera: Neural Networks for Machine Learning, 2012. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[18] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[19] Z. C. Lipton, J. Berkowitz, and C. Elkan. A Critical Review of Recurrent Neural Networks for Sequence Learning. *ArXiv e-prints 1506.00019 cs.LG*, may 2015.

[20] François Chollet et al. Keras. https://keras.io, 2015.

[21] Sorjamaa, Antti and Hao, Jin and Reyhani, Nima and Ji, Yongnan and Lendasse, Amaury. Methodology for Long-term Prediction of Time Series. *Neurocomput.*, 70(16-18):2861–2869, October 2007. ISSN 0925-2312. doi: 10.1016/j.neucom.2006.06.015. http://dx.doi.org/10.1016/j.neucom.2006.06.015.

[22] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014. URL http://arxiv.org/abs/1412.6980. http://arxiv.org/abs/1412.6980.

[23] Rob J Hyndman and Anne B Koehler. Another look at measures of forecast accuracy. *International Journal of Forecasting*, pages 679–688, 2006. https://doi.org/10.1016/j.ijforecast.2006.03.001.

[24] Souhaib Ben Taieb and Gianluca Bontempi. Recursive Multi-step Time Series Forecasting by Perturbing Data. In *Proceedings of the 2011 IEEE 11th International Conference on Data Mining*, pages 695–704. IEEE Computer Society, 2011. doi: 10.1109/ICDM.2011.123. https://doi.org/10.1109/ICDM.2011.123.

[25] Gianluca Bontempi, Souhaib Ben Taieb, and Yann-Aël Le Borgne. Machine Learning Strategies for Time Series Forecasting. volume 138 of *Lecture Notes in Business Information Processing*, pages 62–77. Springer, 2012. https://link.springer.com/chapter/10.1007%2F978-3-642-36318-4_3.