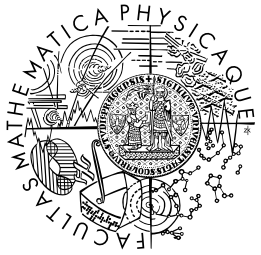


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Natália Tyrpáková

Deep Neural Networks for Sales Forecasting

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the master thesis:	Mgr. Martin Pilát, Ph.D.
Study programme:	Informatics
Study branch:	Theoretical Computer Science

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

Title: Deep Neural Networks for Sales Forecasting

Author: Bc. Natália Tyrpáková

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: Mgr. Martin Pilát, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Sales forecasting is an essential part of supply chain management. In retail business, accurate sales forecasts lead to significant cost reductions. Statistical methods that are commonly used for sales forecasting often overlook important aspects unique for the sales time series, which lowers the forecast accuracy. In this thesis we explore whether it is possible to improve short-term sales forecasting by employing deep neural networks. This thesis analyzes performance of various traditional deep neural network designs and proposes a novel architecture. It also explores several data preprocessing methods, both traditional and non-traditional, which turns out to be a crucial part of sales forecasting using deep neural networks. The best methods of deep neural network approach that we found are then compared to other forecasting methods such as traditional neural networks or exponential smoothing.

Keywords: sales forecasting, deep neural networks, machine learning

Název práce: Hluboké neuronové sítě pro předpovídání prodejů

Autor: Bc. Natália Tyrpáková

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí práce: Mgr. Martin Pilát, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Předpovídání prodejů je nezbytnou součástí řízení dodavatelských řetězců. U maloobchodů mohou přesné předpovědi značně snižovat náklady. Přesnost předpovědí je však u statistických přístupů často zhoršena tím, že techniky zanedbávají důležité vlastnosti, specifické pro časové řady prodejů. Cílem práce je zjistit, zda je vhodné pro řešení problému předpovídaní prodejů použít hluboké neuronové sítě. Tato práce testuje různé běžně používané návrhy hlubokých neuronových sítí a navíc navrhuje novou architekturu hluboké neuronové sítě kombinující tradiční přístupy. Pozornost je věnována taky předzpracování dat, což se ukáže být stěžejní prvek předpovídání prodejů pomocí hlubokých neuronových sítí. Nejlepší nalezené metody jsou porovnány například s tradičními neuronovými sítěmi nebo s exponenciálním vyhlazováním.

Klíčová slova: předpověd prodejů, hluboké neuronové sítě, strojové učení

First of all, I would like to thank my supervisor Mgr. Martin Pilát, Ph.D. for his help, advices, ideas and all the time he spend guiding this thesis. My thanks also go to Logio and Rossmann companies as they allowed me to use their valuable data. Finally, I want to thank Tom for his helpful suggestions and for being there for me, always offering support.

Contents

Introduction

Supply chain management is the practice of coordinating flow of goods, and services. The flow starts from raw materials and moves through parts suppliers, manufacturers, wholesalers to retailers and finally to consumers. The goal of the whole system is to create customer satisfaction while lowering the costs by eliminating unnecessary expenses and handling.

A crucial aspect of supply chain management that precedes most of the activities is forecasting. Proper forecasting helps to ensure that there is just enough supply to satisfy demand on time. Without good forecasts it is easy to overestimate or underestimate the demand. In the first case, a business ends up with over-stock which increases labor and storage costs and it might force the business to sell the products at discount with lower profit margin. In case of products with short durability, over-stock often leads to raised spoilage and consequently to financial loss. On the other hand, with underestimated demand, business is exposed to stock-outs. In such a case, retailer might either make a last-minute rush order to their supplier with higher purchase price or lose potential sales and disappoint the customers. Disappointed customers whose preferred products are out of stock are very likely to switch stores and never come back [1].

The only way to balance between stock-outs and over-stocks is accurate forecast. In practice, forecasting can never be perfect but its accuracy highly depends on the forecasting method. For a business with large revenues, even a small improvement in forecast accuracy can have a significant impact on company's profit.

Despite the availability of computer-based forecasting systems, businesses (especially the small ones) often used to rely on subjective forecasting methods based on personal judgment and opinion, past experience or best guess. In the last two decades, computer-based forecasting systems gained popularity. Unfortunately, commonly only the simplest forecasting methods, such as moving average, with quite a low accuracy are used. In order to minimize the costs and cover the demand, it is necessary to go beyond simple forecasting and exploit more sophisticated methods which consider various factors that influence the sales.

In this thesis we present a deep learning approach to sales forecasting. In the last few years, deep neural networks [2] outperformed previous state-of-the-art methods in various tasks. While traditional neural networks have been previously successfully applied to forecasting sales, there is not much work done related to sales forecasting using deep neural networks. One of the possible reasons is the absence of public data, since deep neural networks require a large amount of training inputs.

The goal of this thesis is to examine deep neural networks as an approach to short-term sales forecasting. To discover whether deep neural networks are a suitable model for this task, we collect a large amount of sales data and use this data to train various models. In order to determine whether a model is good enough, we should explore many different architectures and learning settings. Since data preprocessing is an important part of training neural networks, we need to experiment with it as well. To be able to state whether deep neural networks are appropriate method for sales forecasting, we compare their performance to other forecasting methods, including traditional neural networks. To summarize, our primary goals are to:

- Explore some of the deep neural network models.
  - Try various architectures of different models.
  - Experiment with learning parameters.
  - Examine several methods of data preprocessing.
- Compare deep neural networks to some other approaches.
- Recommend suitable methods for sales forecasting problem.

In the first chapter we describe artificial neural networks and their training process. Here we also present some of the deep architectures that are currently often used. The second chapter introduces the problem of time series forecasting and describes a number of forecasting methods together with ways to evaluate their performance. In the third chapter we review related work made in the field of sales forecasting and in the field of deep neural networks. The fourth chapter is about the problem that we are solving. It introduces sales forecasting in general, the data that we collected and includes an exploratory analysis of this data. The fifth chapter contains the actual experiments. At first we discuss how we conducted the experiments and prepared the data. Here we depict the methods of data preprocessing, including non-traditional methods, that we used in our experiments. Afterwards, we introduce several forecasting methods that will be used as our benchmarks. Finally, we describe the experiments with neural network models. We try convolutional neural networks and a few types of recurrent networks and propose a new architecture that combines both of these approaches. We evaluate the accuracy of these models, compare them to our benchmarks and to traditional neural network. In addition we analyze the ability of the models to generalize to different data. In the last chapter we discuss some of the extensions to our models that we tried based on the results of our experiments, and propose some ideas for future research.

1. Artificial Neural Networks

In this chapter we introduce artificial neural networks and the process of their learning. We further describe some of the popular deep neural network architectures. In the end of the chapter we mention some problems related to the neural networks and ways to solve or reduce them.

1.1 Introduction to Artificial Neural Networks

Artificial neural network (ANN) is a machine learning model inspired by sophisticated functionality of biological nervous systems. Currently it is one of the most powerful computational models since it is robust, universal and has a strong theoretical background. It is widely used for solving classification or prediction problems in various fields such as medicine, industry, science or business [3, 4, 5, 6].

ANN is basically a composition of highly interconnected processing elements called neurons. Individual neurons are able to receive input signal, process it and pass the processed result as an output signal to other neurons using weighted synaptic connections. As the pieces of information are passed from neurons to neurons, they are transformed and combined many times. In order for transformations to result in correct solution, the synaptic weights between neurons must be adjusted to appropriate values specific for a given problem. These weights are configured through a learning process, however in general, finding the correct weights is an NP-complete problem [7] and there is no guarantee of global solution.

1.1.1 Feedforward Neural Networks

The simplest form of ANN is a (single layer) perceptron [8]. It is a model for supervised learning of binary classifiers - a function that maps input data to a binary value representing corresponding class. The mapping is done by separating the classes using a linear separator which is an $(n-1)$-dimensional hyperplane in n-dimensional input space. For given input vector $x = (x_1, x_2, ..., x_n)$ perceptron constitutes function

$$\phi\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where $w = (w_1, w_2, ..., w_n)$ is a vector of perceptron weights (representing the linear separator) b is bias (shift of the linear separator) and ϕ is a non-linear *activation function*. For a single perceptron, the activation function is usually

defined as

$$\phi(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise.} \end{cases}$$

A graphical representation of a single layer perceptron is showed in Figure 1.1

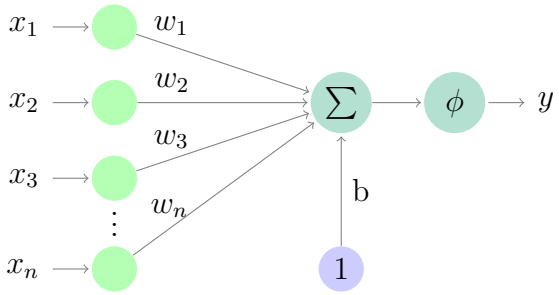


Figure 1.1: Perceptron

Single layer perceptron is quite limited since it is not able to learn patterns that are not linearly separable. For example, it is impossible for a perceptron to learn a simple XOR function [9].

An extension of a single layer perceptron is a multilayer perceptron (MLP) (see i.e. [10]). In the first (hidden) layer MLP transforms the input data using a non-linear transformation into a space where it becomes linearly separable. A single hidden layer makes MLP a *universal approximator* which means it is able, with a finite number of neurons, to approximate any continuous function on compact subsets of R^n to any defined degree of accuracy [11].

MLP is an example of a feedforward neural network (FNN) model. It means that it can be represented as multiple layers of neurons where each neuron is connected by synaptic connection with each neuron from the previous layer (see Figure 1.2). The input layer is connected to the network input and all the connections proceed towards the last, output layer. The layers between input and output layer are called hidden layers.

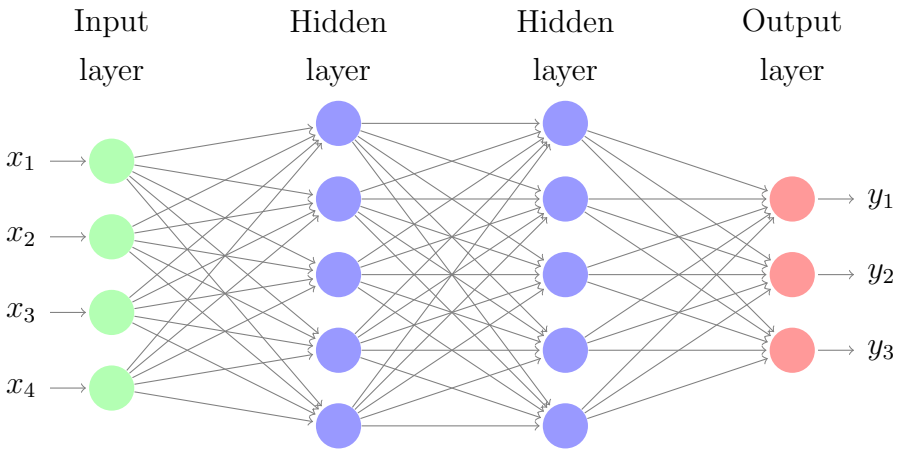


Figure 1.2: Feedforward neural network

Instead of weight vector, FNN uses multiple weight matrices that represent the weights between consecutive layers. We will denote these matrices by $W^{(1)}, W^{(2)}, \ldots$ in order from the input layer so that $W_{ij}^{(1)}$ represents weights from the j-th input neuron to the i-th hidden neuron etc. Similarly we will denote the biases $b^{(1)}, b^{(2)}, \ldots$ Output layer and each hidden layer are associated with a non-linear activation function denoted by $\phi^{(1)}, \phi^{(2)}, \ldots$ A commonly used activation function for FNN is *logistic sigmoid* defined as

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

For a feedforward network with a single hidden layer and input vector $x = (x_1, \ldots x_n)$ the network output is computed as

$$f(x) = \phi^{(2)}(W^{(2)}(\phi^{(1)}(W^{(1)}x + b^{(1)})) + b^{(2)})$$

where $\phi^{(1)}(W^{(1)}x + b^{(1)})$ represents the hidden representation.

From now on, we will include the bias term in the weight matrix. Figure 1.1 shows how the bias term can be represented as a weight from additional neuron with output of 1 in each layer except for the output layer.

1.1.2 Learning Process

As already mentioned, the correctness and accuracy of FNN depends on weights of the synaptic connections between the neurons. In this section we will describe some of the most commonly used methods to obtain these weights. There are specific types of neural networks such as self-organizing maps [12] that are trained using unsupervised learning methods. In this thesis we will only use supervised training so we will only cover methods of supervised learning. These methods

infer the model parameters from labeled training data - examples consisting of input and corresponding output.

There exists a specific learning algorithm for a perceptron [8], however it cannot be extended to learning FNNs; thus we will concentrate on other algorithms. The most popular learning method is *backpropagation* which stands for *backward propagation of errors*. This algorithm was first introduced in 1970s [13] but it was not of much usage until 1986 when it was showed that this method can be used to generate internal representations of data in neural networks [14].

Suppose we have a training set consisting of m training examples $\{(x^{(1)}, d^{(1)}), (x^{(2)}, d^{(2)}), ..., (x^{(m)}, d^{(m)})\}$ where $x^{(i)}$ denotes i-th input vector and $d^{(i)}$ denotes desired output. The goal of backpropagation is to find a combination of network weights that minimize given error function, which is a function of actual network outputs and desired outputs. The way this algorithm works is that it starts with random weights, computes network output for every training example and updates each weight in the network so that the update causes the actual network outputs $y^{(i)} = f(x^{(i)})$ to be closer to the desired outputs $d^{(i)}$. After minimizing the error function for sufficiently large training set, the network is expected to interpolate and give appropriate output even for new, previously unseen inputs.

A commonly used error function is

$$E = \frac{1}{2} \sum_{i=1}^{m} \|y^{(i)} - d^{(i)}\|^2.$$

In each step, network weights are updated using gradient descent algorithm (with gradients propagated backwards) to minimize this error. Commonly a faster alternative to gradient descent is used, called stochastic gradient descent, which approximates the true gradient over the whole data set by gradients for a single example or a smaller subset of examples called a mini-batch. While iterating through the training set, the weights are continuously updated. The update for each weight is computed as negative gradient of the cost function with respect to the specific weight

$$\Delta w = -\frac{\partial E}{\partial w}.$$

To be able to compute the gradients of the error function, we need to guarantee that it is continuous and differentiable. This necessarily means that the activation function of each neuron must be continuous and differentiable. The logistic sigmoid activation function satisfies this constraint with derivation

$$\frac{\partial}{\partial x}\phi(x) = \phi(x)(1 - \phi(x)).$$

Because of the hierarchical structure of FNN, the form of the cost function can be very complicated. However it is possible to obtain the gradients for weights

in every layer more easily, by using chain rule. It means that we can express the error derivatives in one layer in terms of the error derivatives from the next layer. We will describe the procedure using notation from Section 1.1.1 for a network with a single hidden layer, but it can be easily extended for more layers. We now assume a single training sample with desired output d has been presented to the network with actual network output y. For the weight matrix $W^{(2)}$ we compute the update for w_{ji} which is the weight from i-th hidden neuren to j-th output neuron using following equation:

$$\begin{aligned}
\Delta_E w_{ji}^{(2)} &= -\frac{\partial E}{\partial w_{ji}} \\
&= -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j}\frac{\partial \xi_j}{\partial w_{ji}} \\
&= -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j} y_i \\
&= (d_j - y_j)\phi'^{(2)}(\xi_j) y_i = \delta_j y_i.
\end{aligned}$$

Here $\xi_j = \sum_{i' \in \text{hidden}} w_{ji'}^{(1)} y_{i'}$ is input of the j-th output vector. For the hidden layer, the equation is slightly different. Now w_{ji} represents weight from i-th input neuron to j-th hidden neuron:

$$\begin{aligned}
\Delta_E w_{ji}^{(2)} &= -\frac{\partial E}{\partial w_{ji}} \\
&= -\left(\sum_{k \in \text{output}} \frac{\partial E}{\partial \xi_k}\frac{\partial \xi_k}{\partial y_k} \right) \frac{\partial y_j}{\partial \xi_j}\frac{\partial \xi_j}{\partial w_{ji}} \\
&= -\left(\sum_{k \in \text{output}} \frac{\partial E}{\partial \xi_k} w_{kj} \right) \frac{\partial y_j}{\partial \xi_j} y_i \\
&= \left(\sum_{k \in \text{output}} \delta_k w_{kj} \right) \phi'^{(1)}(\xi_j) y_i
\end{aligned}$$

where δ_k is pre-computed from updates of the previous layer.

The backpropagation algorithm is summarized in Algorithm 1. The α term represents a learning rate. The higher this value is, the faster the neurons train but the accuracy of training is decreasing. It is often beneficial to use greater learning rate in the beginning of training and decrease it to fine-tune when network weights are close to the optimum.

Algorithm 1 Backpropagation algorithm

```
1: initialize network weights with random values
2: for epoch ← 1, ..., #epochs do
3:     for each batch of samples do
4:         for all (x^(k), d^(k)) ∈ batch do
5:             y^(k) ← network output for input x^(k)
6:         end for
7:         compute the error function E on current batch
8:         for each synaptic weight w_ji do
9:             compute Δ_E w_ji
10:            w_ji^(new) ← w_ji^(old) + α Δ_E w_ji
11:        end for
12:    end for
13: end for
```

There are several potential adjustments to this algorithm. An example of such adjustment is *momentum* which eliminates oscillations in the gradient descent. The idea behind momentum is to not only follow the negative gradient direction but also include correction direction of the previous update. In our algorithm this would mean to replace the code in Line 10 by

$$w_{ij}(t+1) \leftarrow w_{ij}(t) + \alpha \Delta_E w_{ij} + \alpha_m (w_{ij}(t) - w_{ij}(t-1)).$$

Here α_m is a momentum rate which defines how much the oscillations will be neutralized.

Another adjustment are RMSProp optimizer [15] that divides the learning rate by exponentially decaying average of recent root mean squared gradients or Adagrad optimizer [16] that divides the learning rate by sum of the previous gradients. These optimizers adapt the learning rate individually for each network weight which allows larger updates in case of small gradients and smaller updates in case of large gradients. This increases robustness of the network and leads to faster and more reliable convergence.

1.1.3 Data Preprocessing

In general, neural networks are known to learn faster and give better performance if the inputs are preprocessed before being fed to the network [17]. Here we will describe several objectives of data preprocessing.

First of all, it is often desired to reduce the input space. In case of many input variables, it usually helps the network to manually choose the most important

variables or combine several variables to a single one. By limiting the number of input variables it is possible to reduce the amount of training vectors needed. This results in shorter training time.

Another form of preprocessing that is usually used when training and using neural networks is data normalization (scaling). Above all, this solves the problem when different units are used to measure each input variable and the values of some variables are significantly larger than others and therefore dominate the input. In addition, normalizing inputs help the network to learn faster.

Besides input space reduction and normalization, preprocessing may be used to remove or repair noisy and corrupted data or to manually pre-extract features to simplify the learning process. Both of these can lead to significant improvements in learning, however they are strongly domain-specific and usually require deeper analysis of the data.

1.2 Deep Neural Networks

Artificial neural network architectures were always inspired by their biological counterparts. When the cat's visual cortex was studied in 1962, it was discovered that more complex cells exhibit more sophisticated behavior [18]. This gave rise to an idea of artificial neural network with multiple successive nonlinear layers of neurons - deep neural networks. These should use the intermediate layers to build up multiple levels of abstraction.

Even though it was theoretically possible to train these deep neural networks using standard backpropagation algorithm, researchers soon found out that in reality it did not work very well. In 1991 vanishing and exploding gradients [19] were described as origins of this problem. Both of these problems are further described in Section 1.3.

As a result of mentioned problems, the depth of practical neural networks was usually limited to a single hidden layer for a long time. The expression deep learning arose in 2000s with huge advances in the field of deep neural networks such as deep belief networks [20] with outstanding error rate on the MNIST dataset [21], improved convolutional neural networks with its first GPU implementations [22] or hierarchically stacked long-short term memory (LSTM) networks [23].

Since 2009 deep neural networks started to break records in many machine learning competitions and turned out to be state-of-the-art method in various fields such as sequence labeling, handwriting and image recognition, language modeling and others. In 2011 a max-pooling convolutional neural network produced twice better error rate than human test subjects in traffic sign recognition

contest [24]. This was a huge breakthrough since it was the first system to achieve superhuman visual pattern recognition.

By 2012, deep learning brought excellent results in image recognition and classification. In 2012 a huge convolutional network [25] was introduced, that significantly outperformed previous state-of-the-art methods for image classification. The network featured 60 million parameters and was trained using very efficient GPU implementation. In the same year even better results were achieved. A convolutional neural network with 10^9 parameters showed that it was possible to build high-level, class-specific feature detectors from only unlabeled data [26]. They were pre-training the network on 10 million unlabeled images downloaded from the internet, on 1000 machines each with 16 cores for three days. After the pre-training, the network was trained to recognize 22,000 object categories from ImageNet [27] dataset. It achieved a leap of 70% relative improvement over the previous state-of-the-art results.

Outperforming all other approaches on several benchmarks, in the last few years LSTM networks also showed promising results [28, 29, 30, 31, 32, 33]. These benchmarks were related to various fields such as keyword spotting, speech recognition, language identification or social signal classification. Since that time, both LSTM networks and GPU based convolutional networks have been the most competition-winning or benchmark record-setting deep learning methods.

1.2.1 Recurrent Neural Networks

Neurons and their connections in FNNs form an acyclic graph. By allowing cyclical connections we create so called recurrent neural networks (RNNs) (see i.e. Elman's simple recurrent network (SRNN) [34]). The most important feature of RNN is that recurrent connections allow retaining information from the past inputs and therefore keep some kind of an internal state of the network. This feature enables detection of long-term temporal correlations in input data and makes the RNN suitable for processing sequential inputs.

An RNN can be converted into a FNN by *unfolding over time*. An example of unfolding a simple recurrent neural network with a single input and output per time step is shown in Figure 1.3. The input to the hidden layer at each time step consist of the actual input at the time step and activation of the hidden layer from previous time step. Therefore, besides stacking multiple recurrent layers, RNNs also deliver a different concept of depth in the form of multiple nonlinear layers when unfolded in time.

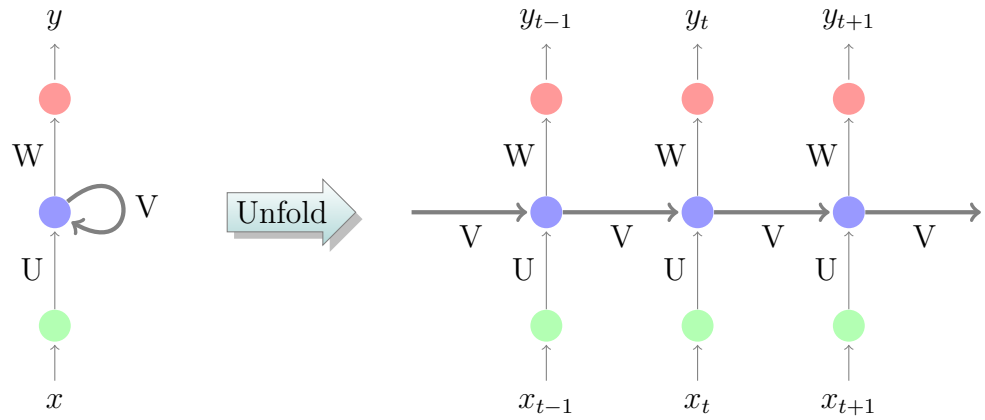


Figure 1.3: Unfolding Elman's simple recurrent neural network

A commonly used algorithm for training RNNs is *backpropagation through time* [35]. This algorithm uses unfolded recurrent network and trains it with standard backpropagation algorithm. The main difference is that the recurrent weights are shared across the unfolded layers so the gradients at each time step need to be summed up.

There is a problem with RNNs regarding vanishing gradient which causes elimination of contributions from earlier layers. This results in inability to learn long-range dependencies and limits RNNs to looking back only a few time steps. One of the possible solutions is a different neural network architecture called LSTM network that is covered in the next section.

LSTM Network

LSTM network [36] was introduced for the first time in 1997. It was designed to hold information for a long period of time. Since 1997 it has been improved and extended by many researchers. Through the time, basic LSTM networks and their modifications have become state-of-the-art method for many difficult machine-learning problems such as handwriting recognition [37], language modeling [38], prediction of protein secondary structure [39] or audio data analysis [40]. This section describes the network exactly as it is used later in the experiments. That means the original LSTM architecture introduced by Hochreiter [36] extended by a forget gate [41] that was added three years later.

As shown in Figure 1.3 in a standard RNN the repeating module is a simple self-recurrent connection on hidden neurons. In a LSTM network, the repeating module consists of a more complex structure called *memory cell*. Its architecture is briefly depicted in Figure 1.4. A memory cell is composed of a neuron with self-recurrent connection that enables maintaining its state over time and three non-linear gating units - input gate, output gate and forget gate. The gating

units regulate the information flow through the cell and manage its memory.

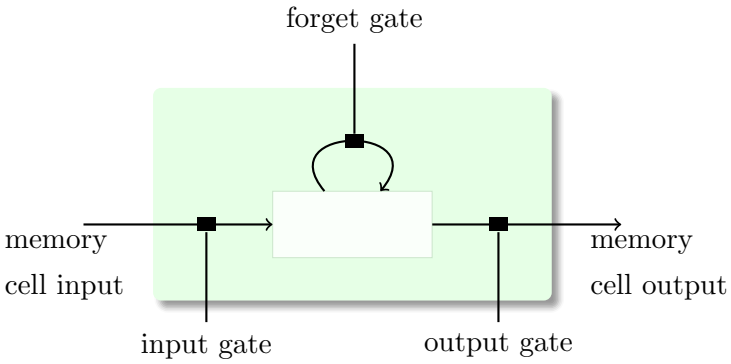


Figure 1.4: LSTM memory cell

The input signal of the memory cell consists of the input from current time step and memory cell output from the previous time step. The input signal presents new values for the cell state and defines behavior of the gates. The input gate filters which cell state values will be updated by the new input and decides how strong the updates will be. It might completely block the input signal or allow it to fully or partially update the state of memory cell. The forget gate determines how much of the previous cell state will be remembered. It also allows to completely reset the state. The output gate serves as a filter of the cell state and decides which parts of it will affect other neurons.

To fully understand the functionality of memory cell, we provide formulas that describe how the forward pass is actually realized. The formulas are given below, while the detailed schema of memory cell can be found in Figure 1.5.

$$z(t) = g(W_z x_t + R_z y_{t-1} + b_z) \qquad \text{cell input at time t}$$
$$i(t) = \sigma(W_i x_t + R_i y_{t-1} + b_i) \qquad \text{input gate at time t}$$
$$f(t) = \sigma(W_f x_t + R_f y_{t-1} + b_f) \qquad \text{forget gate at time t}$$
$$c(t) = z(t) * i(t) + c(t-1) * f(t) \qquad \text{cell state at time t}$$
$$o(t) = \sigma(W_o x_t + R_o y_{t-1} + b_o) \qquad \text{output gate at time t}$$
$$y_t = h(c(t)) * o(t) \qquad \text{cell output at time t}$$

Where:

$*$ represents point-wise multiplication of two vectors

x_t = input vector at time t

y_t = output vector at time t

W_z, W_i, W_f, W_o = input weight matrices for input vector, input gate, forget gate and output gate respectively

R_z, R_i, R_f, R_o = recurrent weight matrices for input vector, input gate, forget gate and output gate respectively

b_z, b_i, b_f, b_o = bias vectors for input vector, input gate, forget gate and output gate respectively

σ = activation function of the gates, always logistic sigmoid

g = cell input activation function

h = cell output activation function

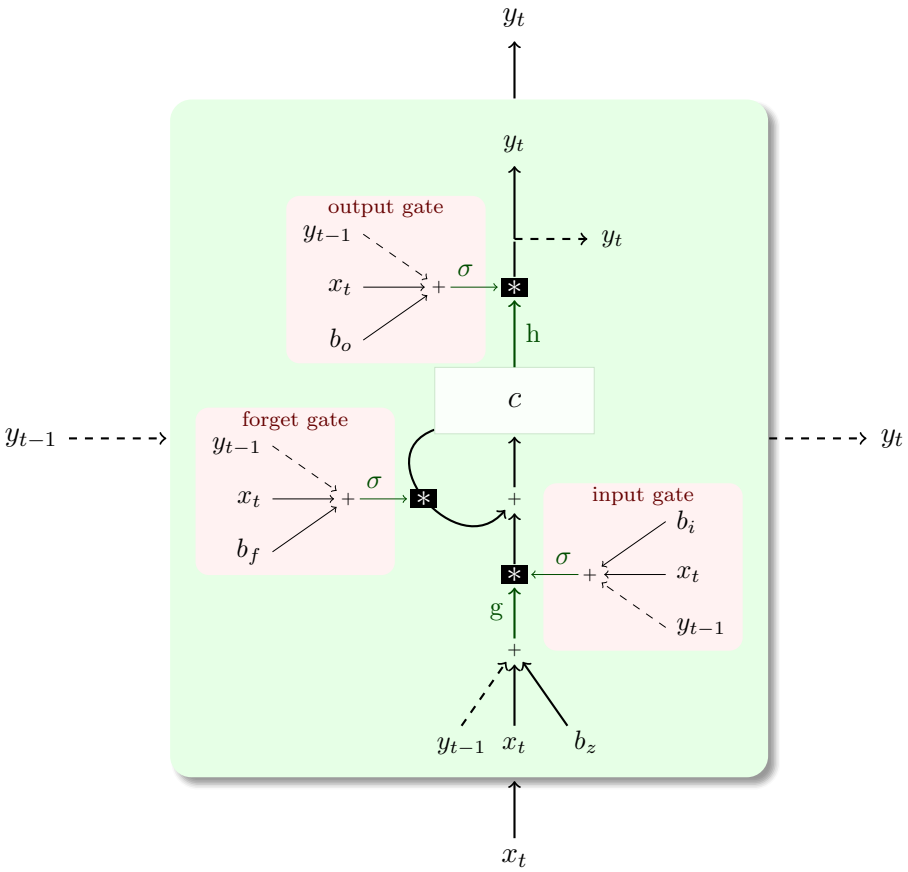


Figure 1.5: LSTM memory cell in detail

The concept described above is just a basic variant of LSTM network. There are several other variations for example LSTM cell with *peepholes* [42] (connec-

tions from the cell to the gates) or a simplification of LSTM called gated recurrent unit (GRU) [43]. The GRU combines forget gate and input gate into a single update gate and it does not contain any output gate to control how much of its state will be exposed. It always exposes the whole internal state. For further details and other variations see, for example, a large-scale systematic overview and comparison of eight LSTM variations presented by Greff et al. [44].

1.2.2 Convolutional Neural Networks

The first convolutional neural network (CNN) was introduced in 1979 [45] and was given a name Neocognitron. The architecture of this network was very similar to the modern contest-winning CNNs [22, 24, 25, 26]. It is often described as the first actually deep network, however the depth did not mean a lot then, since the network was not trained using backpropagation. It was trained using local unsupervised learning rules based on winner-takes-all methods.

Since Neocognitron, the backpropagation algorithm was designed, computational power increased significantly, efficient GPU implementation and many other improvements to neural networks and their training were discovered. As a result, while the architecture remains similar, we are able to exploit CNNs in a much greater ways. This section describes basic principles of CNNs and presents some of the advances that were made during past decades and led to current excellent results.

CNNs are often associated with image recognition and classification tasks. They make use of a commonly used technique for image processing - convolution filtering. It is a simple way to emphasize or remove certain features from the image. Filtering is a neighborhood operation where the value of an output pixel is computed as a linear combination of the original pixel's neighborhood. The weight matrix of this linear combination is called *convolution filter* or *convolution kernel*. Using this simple operation it is possible to do for example image smoothing, sharpening or edge enhancement. Figure 1.6 shows the result of an image convolution with a simple kernel

$$\begin{pmatrix} 1 & 4 & 1 \\ 0 & 0 & 0 \\ -1 & -4 & -1 \end{pmatrix}.$$

Figure 1.6: An example of edge enhancement using image convolution.

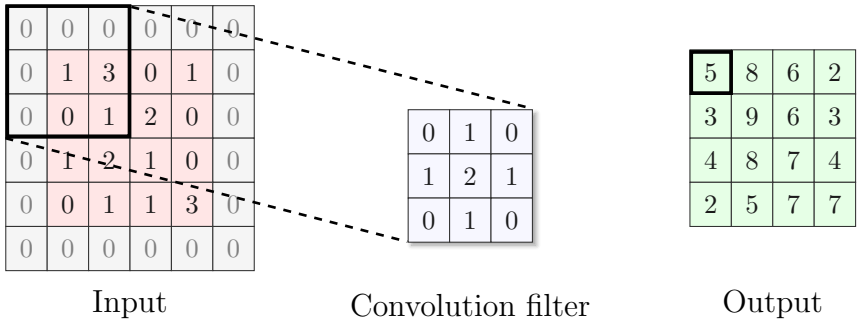


Figure 1.7: Convolution filter applied to a matrix representing the values of pixels. The convolution is a linear combination of pixel's neighborhood. For instance the value of the first pixel is computed as $0\times0+1\times0+0\times0+1\times0+2\times1+1\times3+0\times0+1\times0+0\times1$. Here we handled the edges by adding zero values, another option is to copy neighboring values.

CNN is a type of FNN, however, instead of fully-connected layers it features three kinds of layers - convolutional layer, pooling layer and fully-connected layer. A complete CNN is created by stacking these layers onto each other. The fully-connected layer is usually only the last one or last few layers and it is supposed to provide a high-level reasoning.

The convolutional layer is the core building block of a CNN. Neurons in this layer represent a filter (or multiple filters that are applied simultaneously) and perform a convolution across the input. An important fact is that the same convolutional filter is applied across the whole image. The idea behind this architecture is that the same features that are learnt at one part of the input can also be found

in other parts of the input. The convolutional filter represented by neurons in convolutional layer is therefore also called a feature detector. The network is supposed to learn filters that are activated by visual features such as edges, color blocks or more complex and higher level features in case of hierarchically stacked layers. A huge advantage of this architecture is that since the same convolutional filter is applied across the whole image, the number of network parameters can be significantly smaller than in case of FNN. This makes the training notably easier even in case of large networks and helps to prevent overfitting (see Section 1.3).

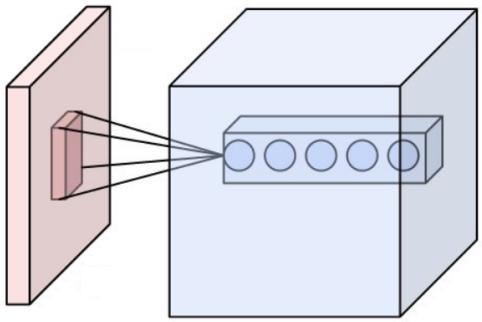


Figure 1.8: Each neuron in convolutional layer (blue) is connected to the full depth of a local region in the input space. There are multiple neurons all processing the same region - these represent different filters. Each of these filters process the input region with different weights. These filters are replicated to process all the input regions in exactly the same way.

Pooling layers are usually inserted between successive convolutional layers. Their function is to reduce the spatial size of the representation and decrease the number of network's parameters. The pooling layer partitions its input into non-overlapping regions and applies a down-sampling operation to each region. Its output is for example maximum of each region, average, sum or L2-norm ($|x| = \sqrt{\sum |x_k|^2}$). The intuition behind this is that the exact location of a feature is not as important as its relative location to other features.

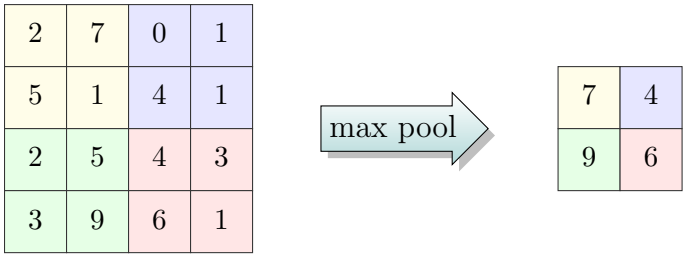


Figure 1.9: An example of max-pooling. The layer input is partitioned into 4 regions and each of them is down-sampled using max-pooling operation.

In case of image processing, the network input is usually a 3D matrix where the depth represents for example color channels. Convolution can be however also applied to inputs with lower dimensions, for example time series (see i.e. [46]). Similarly to images, convolutional networks might be able to reveal and exploit various features of a time series without need to identify them manually.

1.3 Problems

Training a neural network is not a simple task and there are several common possible problems. This section introduces these problems and describes some approaches how to overcome them.

Non-convexity

Due to the nonlinear activation functions, the problem of finding optimal parameters of neural network is generally non-convex. Therefore there does not necessarily exist a global solution and network might get stuck in a local optimum - weight configuration for which no small change would reduce the error. This problem has been studied and propagated by researches for a long time. Recent results however show, that local minima are, in general, not a serious issue [47].

Overfitting

Overfitting is one of the major issues with neural networks training. With a large number of parameters, the network does not only learn general patterns but it also learns random noise that occurred in training data. Such a network then performs very well on data that it has been trained on, but it does not generalize to previously unseen inputs which is the primary goal.

It is important to notice, that if the size of training set is significantly larger that the number of network parameters there is basically no chance of overfitting. The issue can be therefore solved by simply collecting more training data. This is however often not possible. The amount of labeled training data is limited and it is often not sufficient especially in case of deep neural networks with many parameters. A commonly used solution to missing labeled training data is pre-training the network using unlabeled data [26] or generating more training data by transforming the available data so that the label is preserved.

A common technique to avoid overfitting is early stopping [48]. This method introduces a validation data, which is a part of data, separated from training data. The network is not trained on validation data, they only serve for monitoring the

error on unseen data during the training. When the network begins to overfit, it is easily noticeable, for the error is decreasing on the training data but increasing on the validation data.

Another method for improving generalization is weight regularization. Weight regularization is based on a fact, that smaller network parameters lead to smooth outputs which change slowly and do not learn the noise. To achieve smaller network parameters, an additional term is added to the error function of the network to penalize large weights. One of the most commonly used regularization techniques is L2 regularization also known as weight decay [49]. Weight decay regularization is realized by adding term

$$\frac{1}{2}\lambda \sum_i w_i^2$$

to the network cost function. Here w_i are network parameters and λ is a parameter that controls how strongly large weights are penalized.

A quite recent approach to avoid overfitting is using dropouts [50]. The idea of dropouts is to randomly temporarily drop units with their connections from the neural network during training. It has been shown that dropouts significantly reduce overfitting and present major improvements over other regularization methods. The simplest example of dropouts is removing each unit with a fixed probability, independent of other units. An effective probability of dropout is often surprisingly large. The paper that introduces dropouts claims that 0.5 has shown to be close to optimal for various networks and tasks.

Vanishing or Exploding Gradient

Vanishing gradient results from the fact that with each subsequent layer in the network, magnitude of the gradients gets exponentially smaller. Exploding gradient is an opposite problem usually connected with recurrent neural network described in Section 1.2.1. Since the problems have been introduced for the first time [19], several ways of partially overcoming these problems were explored such as specific network architectures (i.e. LSTM network described in Section 1.2.1), unsupervised pre-training or regularization of network's weights which also helps to overcome overfitting. Another option is using more practical activation function. While logistic sigmoid or hyperbolic tangent enhance vanishing of the gradient, several other activation functions such as rectified linear unit (ReLU) defined as $\phi(x) = max(0, x)$ do not have this property.

Curse of Dimensionality

The term *curse of dimensionality* was first brought up in 1960 by Richard Bellman [51]. It refers to a problem when large number of input features cause network to memorize the data and perform worse than when trained using inputs with less features. Since each feature adds another dimension to the input space, significantly more training data is needed to sufficiently populate the input space [52]. The curse of dimensionality can be overcome by properly selecting and reducing input features. One of the most common feature extraction methods is for example principal component analysis [53].

2. Forecasting

In this chapter we introduce the problem of time series forecasting and describe some of the commonly used forecasting methods such as exponential smoothing, ARIMA model and a few machine learning approaches.

2.1 Time Series Forecasting

A time series [54] is a sequence of numerical data points made out of successive measurements across a time interval. An example of time series is daily household energy consumption, interest rates, fuel price, company profits or monthly rainfall.

Time series usually contain a lot of fluctuations - either random or systematical. These fluctuations are often analyzed to observe the nature of the phenomenon represented by the series and extract their characteristics. The systematic pattern can be decomposed into two main aspects - trend and seasonality. The trend component is linear or nonlinear pattern that does not repeat within the range of given time series. Contrastingly, seasonal component repeats in systematic intervals.

The most common motivation that leads to time series analysis is obtaining its future values. Because of the random noise which is practically always present, future values of real-life time series usually cannot be computed precisely. However most of the times they can be predicted with admissible inaccuracies. Since forecasting models can only operate with limited amount of information, they are unable to exploit all the causalities that affect the predicted variable. Instead of that, they make inferences about future values based on its past behavior, in some cases combined with some other external variables.

2.2 Forecasting Methods

For the purpose of this thesis we will use the following notation. The observation at time t will be denoted by a subscript t, i.e. y_t. The observed data will be represented as y_0, y_1, ..., y_{t-1} while predicted values will be represented as $\hat{y}_t$, $\hat{y}_{t+1}$, ..., $\hat{y}_{t+n}$.

There is a wide range of forecasting methods, each of them having specific properties, accuracies, advantages and disadvantages. There is no generally superior forecasting method, the best method usually strongly depends on nature of the data. When choosing the right method, it is necessary to evaluate its

performance by predicting past values that we already know the true value of.

2.2.1 Basic Methods

The most simple forecasting method is *naive method* which predicts the next value to be the same as the latest observed value. Naive approach can even consider seasonality and predict value from the last season. For a single-step prediction this method is quite effective and since it is very easy to implement, it is commonly used as a benchmark. Another simple methods are *average*, *moving average* or *weighted moving average* which uses higher weights for newer observations.

2.2.2 Exponential Smoothing

Exponential smoothing describes a class of forecasting methods that weight past observations using exponentially decreasing weights. This section depicts some variations of the algorithm that have been proposed since it was introduced for the first time in 1950s. The algorithms in this chapter are mainly based on a book *Forecasting: principles and practice* [55].

Simple Exponential Smoothing

The original exponential smoothing method [56] suggested in 1956 by Brown is only suitable for time series without any seasonal or trend patterns. The value $\hat{y}_{t+1}$ is forecasted as l_t which is an *estimated level* at time t. The *estimated level* is defined as

$$l_t = \alpha y_t + (1 - \alpha) l_{t-1}$$

where $0 \leq \alpha \leq 1$ is a smoothing parameter, which can be chosen based on past experience or optimized to minimize forecasting error using a statistical technique such as least squares method. The value for initial level l_0 is usually y_0, however if the time series is sufficiently long and α is not too small the initial level does not significantly affect the resulting forecast. In case of multi step ahead prediction, the same forecast value is used for each forecasting horizon ($\hat{y}_{t+h} = l_t$).

Holt's Linear Trend Method

In 1957 Charles Holt extended Brown's exponential smoothing [57] to enable forecasting time series with trend. This method uses *estimated level* l_t similarly as it is used in simple exponential smoothing in additional combination with *estimated trend* b_t which represents an estimation of the slope of time series.

These values are computed as follows:

$$l_t = \alpha y_t + (1-\alpha)(l_{t-1} + b_{t-1})$$
$$b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$

where $0 \leq \alpha \leq 1$ is smoothing parameter for level and $0 \leq \beta \leq 1$ is smoothing parameter for trend. The actual predicted value at time $t+h$ is then defined as

$$\hat{y}_{t+h} = l_t + hb_t.$$

The forecasting function is not constant as it was with simple exponential smoothing, but linear in forecasting horizon h.

Holt-Winters Seasonal Method

Holt-Winters seasonal method [58] also called triple exponential smoothing was first suggested in 1960. This method combines three smoothing equations that compute *estimated level* l_t , *estimated trend* b_t and *estimated season* s_t. There are two variants of this method. The first one, additive method, uses the seasonal component expressed in absolute terms and is suitable for time series with constant seasonal variations. The second, multiplicative method, uses the seasonal component expressed in relative terms and is suitable for time series with seasonal components proportional to the level. The estimates and prediction of the additive method are computed as follows:

$$l_t = \alpha(y_t - s_{t-m}) + (1-\alpha)(l_{t-1} + b_{t-1})$$
$$b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$
$$s_t = \gamma(y_t - l_{t-1} - b_{t-1}) + (1-\gamma)s_{t-m}$$
$$\hat{y}_{t+h} = l_t + hb_t + s_{t-m+\lfloor (h-1) \bmod m \rfloor + 1}$$

where $0 \leq \alpha \leq 1$ is smoothing parameter for level, $0 \leq \beta \leq 1$ is smoothing parameter for trend and $0 \leq \gamma \leq 1-\alpha$ is smoothing parameter for season. The last term of prediction equation ensures that the last estimated season from the observed data is used. In comparison, the multiplicative method uses following equations:

$$l_t = \alpha\frac{y_t}{s_{t-m}} + (1-\alpha)(l_{t-1} + b_{t-1})$$
$$b_t = \beta(l_t - l_{t-1}) + (1-\beta)b_{t-1}$$
$$s_t = \gamma(\frac{y_t}{l_{t-1} - b_{t-1}}) + (1-\gamma)s_{t-m}$$
$$\hat{y}_{t+h} = (l_t + hb_t)s_{t-m+\lfloor (h-1) \bmod m \rfloor + 1}.$$

2.2.3 ARMA and ARIMA Models

Autoregressive moving average (ARMA) model [59] is often used for modeling time series and predicting their future values. It is a combination of two models, namely autoregressive model (AR) and moving average model (MA).

AR(p) model refers to the autoregressive model of order p. It models the time series as a noisy linear combination of the values at last few time step. This model is able to handle a wide range of different time series patterns. Mathematically it can be expressed as

$$\begin{aligned}\hat{y}_t &= c + \sum_{i=1}^{p} \varphi_i y_{t-i} + \varepsilon_t \\ &= c + \varphi_1 y_{t-1} + \varphi_2 y_{t-2} + ... + \varphi_p y_{t-p} + \varepsilon_t\end{aligned}$$

where φ_i are the model parameters, ε_t is random error at time t and c is constant. AR model is restricted to stationary time series - a series whose properties, such as mean, variance, autocorrelation, etc. are independent of the time at which the series is observed.

MA(q) model refers to the moving average model of order q. It is also a regression-like model but instead of using past values, it models the time series using past forecast errors. It can be expressed as

$$\begin{aligned}\hat{y}_t &= \mu + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} + \varepsilon_t \\ &= \mu + \theta_1 \varepsilon_{t-1} + \theta_2 \varepsilon_{t-2} + ... + \theta_p \varepsilon_{t-p} + \varepsilon_t\end{aligned}$$

where μ is the mean of the time series, θ_i are the model parameters and ε_t is white noise. It is not possible to fit this model using linear least square method, since the random error terms are not observable.

By combining AR(p) and MA(q) models we obtain an ARMA(p, q) model. It models the time series as

$$\hat{y}_t = c + \sum_{i=1}^{p} \varphi_i y_{t-i} + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i} + \varepsilon_t.$$

The ARMA model was first described in 1951 [59]. It was, however, really popularized much later, in 1971 when a well-known Box-Jenkins methodology [54] for ARMA model selection and parameter estimation was introduced.

The ARMA model is limited to stationary time series which is quite a restriction since in practice most of the time series contain trend or seasonal patterns which makes them non-stationary. This is why a generalization of this method called autoregressive integrated moving average model(ARIMA) [54] model has been proposed. This model first transforms a non-stationary time series to a stationary one by applying finite differencing of the data points.

2.2.4 Machine Learning Methods

In the last two decades, machine learning models have shown to be a real competition to the classical statistical models when it comes to time series forecasting. Rather than processing data using pre-programmed algorithm, machine learning methods are based on building models that learn, grow and change when exposed to new data. Data provided to the model in order to train it, is called training data.

Machine learning methods are able to solve various classification, clustering or regression tasks. In this section, we will only focus on the algorithms for regression since these are the most relevant ones when it comes to time series forecasting. It is however possible to transform the problem of time series prediction into classification problem, for example by predicting one of several possible ranges of the output value.

Regression problems are usually solved using the methods of supervised, or possibly semi-supervised learning. That means, at least some of the input data are labeled - the desired output is provided. Training data therefore consists of pairs (x, y) where $x = (x_1, ...x_n)$ is model input consisting of n features and y is desired output. For our prediction task, x might be a vector of previous values $x = (y_0, y_1, ..., y_{t-1})$ and y would be a single predicted value y_t or even a vector of multiple predicted values $(y_t, y_{t+1}, ...)$.

Artificial Neural Networks

In comparison to statistical forecasting methods, ANNs can easily model data with nonlinear relationships between variables and overcome other limitations such as problems with outliers. Another advantage of ANNs is that they are often able to discover features in the data, that other models do not exploit. Probably the most critical flaw of ANN models is that they work as a black box, meaning that they deliver the results without any explanation.

A regression using neural networks is done either by scaling data into appropriate range, coverable by a standard activation function such as logistic sigmoid or hyperbolic tangent. Another option is using activation function without limited range, for example softplus function given by

$$\phi(x) = \ln(1 + e^x).$$

The network weights and biases ale trained by feeding past data into the network. The problem with this approach is variable input size, since neural networks are trained with a fixed architecture that includes number of input neurons. This problem is usually solved in two ways. The first way is to introduce

a recurrent neural network that maintains its state and is able to process inputs of any length. Another option is to only use a fixed number of past time steps for prediction.

Random Forests

Random forest [60] is an ensemble of decision trees [61], with each tree constructed using a random subset of the training data. To introduce random forests, we will first briefly describe how do decision trees work.

Decision tree is a non-parametric supervised learning method. It is commonly known for solving classification tasks, however it can also be used for regression. Decision tree is supposed to predict the correct output by learning simple decision rules inferred from the training data features.

During the training phase, a tree with decision nodes and leaf nodes is incrementally built. For a classification task, the tree is built by breaking down the training dataset into smaller and smaller subsets containing instances of similar values. For a regression task, in each step a regression model is built for every independent variable (input feature). The models are fit to predict the output value. Then the data is split at several split points and at each split point, the sum of squared errors between predicted values and actual values are computed. Variable and split point with the lowest error is chosen as the next decision node. Each part of the split is then recursively processed in the same manner.

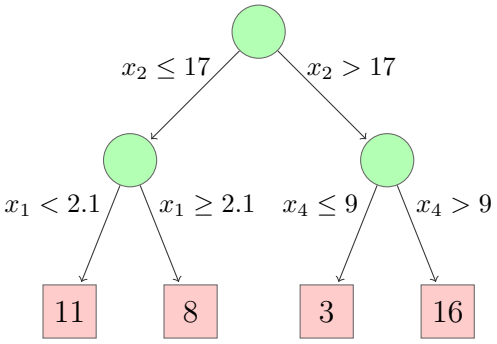


Figure 2.1: An example of decision tree for regression tasks. The green circles represent decision nodes and red squares represent the output value.

After the tree is fit, the actual classification or regression is done by walking the tree from the root node to a leaf according to the decision rules in each node. The leaf value is the predicted output.

A fully developed tree often suffers from overfitting and exploits random features that occurred in the training set but are not likely to occur generally. To overcome this problem, a validation set is usually used to prune the tree.

A random forest is composed of an arbitrary number of decision trees that all take part in determining the predicted output. In case of classification problem, the trees vote for the final class and the most popular one is chosen. In case of regression problem, the output might be computed for instance as an average of their responses. Random forests as a forecasting model do not require much parameter tuning, they are quite fast and able to work well even with unbalanced and missing data.

Support Vector Machines

Support vector machines (SVMs) [62] are a set of supervised machine learning methods used both for classification and regression tasks. In case of classification problems, the input data can be imagined as points in n-dimensional space, where n is determined by number of features. The classification into two classes is performed by finding a hyper-plane that appropriately differentiates the input data. It is however often not possible to linearly separate the two classes. SVMs solve this problem by transforming the input data into the space of higher dimension where the data becomes linearly separable (see Figure 2.2).

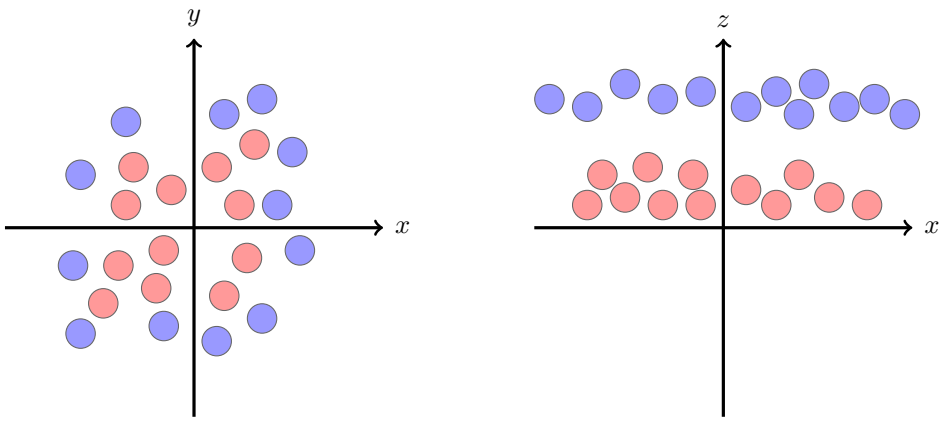


Figure 2.2: Red and blue circles in the illustration picture represent data from two different classes. In the left subfigure, these two classes are not linearly separable. By including another feature $z = x^2 + y^2$ and therefore transforming data into the space of higher dimension, data becomes linearly separable (right subfigure).

Even though the classification tasks are more common for SVMs, they are also capable of solving regression problems. This algorithm is called ε-SV regression [63] and its goal is to find a linear function $f(x)$ that has at most ε deviation from the target values from a training set. Obviously, many regression problems cannot be appropriately described using a linear model. In this case, the data is also transformed into a space of higher dimension where such a function exists.

Formally, the objective is to find a linear function

$$f(x) = w^T x + b$$

where we want to assure that it is as flat as possible. This is achieved by minimizing norm value $w^T w$:

$$J(w) = \frac{1}{2} w^T w.$$

For each training sample (x, y) the function must satisfy the condition

$$\left| y - (w^T x + b) \right| \leq \varepsilon.$$

Finding the desired function is an optimization problem that can be expressed in standard quadratic programming form and then solved using appropriate techniques.

2.3 Performance Evaluation

When choosing and tuning suitable forecasting model, prediction accuracy should be the primary concern. It is very important to acknowledge that the main goal is to obtain a model that most accurately predicts new data, not the data it was trained on. To compute the prediction accuracy or inversely, prediction error, the model is usually trained on a part of a series and it is measured how accurately it predicts the rest of the series.

There are few factors to be considered when selecting a forecasting error calculation method. The first one is whether it is scale independent. A scale dependent error is on the same scale as the data and therefore it can usually only be used to make comparisons on the time series that are on the same scale. Second feature that might limit the function usage is whether it is able to handle zero values. A common pattern is that in order to gain scale independence, the error is scaled. The scaling unfortunately often rules out time series containing zero values. The third, often ignored, factor is whether the error is biased in a way that it systematically prefers models whose predictions are too low or reversely, too high. The last important characteristic is interpretability of the error. Table 2.1 presents some of the commonly used methods for computing forecasting error. Their comparison based on described characteristics can be found in Table 2.2. It may be noticed that zero denominator in formulas for SMAPE and MIBRE implies zero error and therefore it can be properly handled.

Method	Method name	Formula
MAE	mean absolute error	$\text{mean}(\lvert\hat{y}_i - y_i\rvert)$
MAPE	mean absolute percentage error	$\text{mean}(\frac{\lvert\hat{y}_i - y_i\rvert}{\lvert y_i\rvert})$
SMAPE	symmetric mean absolute percentage error	$\text{mean}(\frac{\lvert\hat{y}_i - y_i\rvert}{\lvert y_i\rvert + \lvert\hat{y}_i\rvert})$
MSE	mean squared error	$\text{mean}((\hat{y}_i - y_i)^2)$
RMSE	root mean squared error	$\sqrt{\text{mean}((\hat{y}_i - y_i)^2)}$
NRMSE	normalized root mean squared error	$\frac{\sqrt{\text{mean}((\hat{y}_i - y_i)^2)}}{\bar{y}}$
lnQ [64]	-	$\sum \ln(\frac{\hat{y}_i}{y_i}) = \sum \ln(\frac{y_i}{\hat{y}_i})$
MIBRE	mean inverted balanced relative error	$\text{mean}(\frac{\lvert\hat{y}_i - y_i\rvert}{\max(\hat{y}_i, y_i)})$

Table 2.1: Forecasting error calculation methods

Method	Scale independent	Non-biased	Zero values	Interpretable
MAE	✗	✓	✓	✓
MAPE	✓	✗	✗	✓
SMAPE	✓	✗	✓	✓
MSE	✗	✓	✓	✓
RMSE	✗	✓	✓	✓
NRMSE	✓	✓	✓	✓
LnQ	✓	✓	✗	✗
MIBRE	✓	✗	✓	✗

Table 2.2: Comparison of forecasting error calculation methods

3. Related Work

According to our research there are currently no available results of using deep learning to forecast future sales. Multiple studies show that this problem can be approached by more simple neural networks such as conventional feed-forward networks [65, 66] or shallow recurrent networks [67]. On the other hand, in the last decade deep neural networks have outperformed classical neural network architectures in many fields. In this chapter we present previous studies related to sales forecasting using conventional forecasting methods and traditional neural networks as well as studies presenting state-of-the-art performance of deep architectures or their usage for time series analysis.

3.1 Sales Forecasting

Sales forecasting is a common business practice. Studies from 2000 [68] however indicate that judgmental procedures are more frequent than quantitative or causal methods. They showed that out of quantitative methods, usually the most simple ones are used, for example moving averages or simple regression. This study also showed that most of the firms never even tried to use unconventional methods for sales forecasting such as neural networks or genetic algorithms. In addition, a study of the evolution of sales forecasting management [69] from 2006 shows that familiarity of companies with more sophisticated forecasting methods is actually decreasing over time.

The study of the evolution of sales forecasting management also showed that the method that companies are most satisfied with, is exponential smoothing. Exponential smoothing is a traditional method that has been used in practice for a long time. In 2010 Gelper, Fried and Croux presented a robustified version of exponential and Holt-Winters smoothing [70]. They suggested a pre-cleaning mechanism that identifies and down-weights outlying values. Their methods are significantly less sensitive to outliers than the original ones and comparisons show that the robust Holt–Winters method also yields better forecasts in case of fat-tailed error distributions.

Even though the neural network approach to sales forecasting is not frequently used, it has been around for quite a long time. In 1997 Thiesing and Vornberger [65] applied neural network to predict future values of weekly demand of 20 items in German supermarket. They trained a simple feedforward multilayer perceptron network with a single hidden layer to predict sales for the next week. Along with the past amount they also used other indicators such as price change indicator

or promo action indicator. Compared to two other simple prediction techniques already used by the supermarket (naive prediction and moving average) the neural networks gave considerably better results.

Another comparison [66] of artificial neural networks and traditional forecasting methods was made in 2001 on US monthly aggregate retail sales. They compared neural network approach to Holt-Winters exponential smoothing (see Section 2.2.2), ARIMA model (see Section 2.2.3) and multiple regression (see i.e. [71]). They employed both one-step and multi-step ahead forecast with predictions of up to 12 months. As a neural network model they used a simple FNN with eight units in hidden layer. They compared the performance of each method across two time periods. The first time period was affected by a large macroeconomic instability, while the second period contained significantly less fluctuations. Their results for the first period showed that ANNs provide superior forecast over traditional methods for both one-step and multi-step ahead prediction. In the second period, ARIMA model seemed to outperform neural networks, but the difference was not statistically significant.

There are also more recent results in this field. In 2015 a recurrent neural network approach to sales prediction [67] was presented. They tried to predict 6 monthly sales series using different types of partial recurrent networks. Their results showed that partial recurrent networks can outperform other statistical forecasting methods on certain time series but they did not confirm any general superiority of them.

3.2 Deep Neural Networks

In the last decade, deep neural networks led to many breakthroughs. Lots of them were made in the field of image classification [25, 72]. In 2015, the 1st place on the ILSVRC 2015 classification task, ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation was won by deep residual networks [73]. They solve the problem of degradation, where accuracy gets saturated with increasing network depth, in a very specific way. They introduce shortcut connections that perform identity mapping and skip one or more layers. This allowed them to successfully train models with more than 100 hidden layers that showed an excellent performance.

Another field where deep neural networks show outstanding performance is natural language processing. An example of such a task is character-level language modeling with a task of predicting the next character in a stream of text. Recurrent neural networks with a Hessian-Free optimizer [74] has shown excep-

tional results when solving this task [75]. The researchers that introduced this approach presented a new temporal network architecture called multiplicative recurrent neural network that gates the recurrent weight matrix with the input symbol. They used three datasets of size about 100MB. The datasets included characters, digits and punctuation. They trained the network on many shorter sub-sequences, with every prediction supported by at least 50 characters of context. The training took 5 days on 8 GPU units. They showed that it was very easy for the network to learn words, however a much harder task is to learn correct semantic and syntactic structure. Compared to different methods, the network learned to balance parentheses and quotes over long distances and it was also able to extract higher level knowledge from the text.

In 2015 a deep recurrent neural network model was introduced that connected computer vision and natural language processing by solving a task of generating natural sentences describing given image [76]. This task is very challenging and is significantly harder than image classification or object detection. There is a need to capture objects in the image, detect their attributes and relations to each other. Besides visual understanding, a correct language model is necessary. As an image encoder, they used a convolutional neural network pre-trained for an image classification task. The last hidden layer of this network was then fed to a recurrent neural network, specifically LSTM network, that served as a decoder and generated resulting sequences. The LSTM model was trained to predict each word of the sentence given the input from convolutional network and preceding words. Using this architecture extended by dropouts and ensembling, they achieved state-of-the-art results on multiple datasets and reached near-human performance.

Deep neural networks are applicable for various tasks in many fields of study. A good example is biology. In 2014 recurrent neural network predicting protein secondary structure from the amino acid sequence provided outstanding results [39]. This is a classical bioinformatics problem, often solved using feedforward neural networks or support vector machines. Even though it is a prediction task, its character is a bit different since the entire sequence is known beforehand, not only the past observations. They exploit this feature by using bidirectional recurrent neural network. It is a combination of two separate multilayer recurrent networks, one working forward from the beginning of the sequence and one going backward from the end. Their results are then combined using a feedforward neural network. On a given dataset, this architecture outperformed previous state-of-the-art results.

There are also other studies suggesting that recurrent neural networks outperform standard methods in the time-series forecasting task. A comparative

analysis of the wind speed forecasting accuracy of ARIMA models and recurrent neural networks [77] showed that recurrent neural network models outperformed the ARIMA models. Their data was collected on the same tower and over the same time period for a variety of heights and the task was to forecast 15 minute ahead wind speeds. They compared two variants - univariate and multivariate settings. In univariate settings, each height was treated separately. In multivariate setting the input variables from different heights were combined into single input vector. The recurrent neural networks outperformed ARIMA models on both univariate and multivariate settings.

4. Problem Analysis

In this chapter we will describe the problem of retail sales forecasting with focus on short-term sales forecasting which is the problem that we were dealing with. We will also present the sales data that we used in our experiments and provide its exploratory analysis.

4.1 Sales Forecasting

Sales forecasting might be done on multiple levels. Long-term forecasting, such as forecasts for next few years are necessary for strategic planning. These forecasts are usually applied to groups of aggregated products, since yearly forecasts of individual products would be very imprecise and usually even pointless. These forecasts help to make the right business decisions and they are affected by many economical and environmental factors.

Medium-term forecasting refers to forecasting sales for the next few weeks or months. Medium-term forecasts are necessary in the terms of planning production, creating contracts with suppliers, hiring more personnel, ordering goods with higher durability, etc. Depending on the purpose of forecast, these forecasts may be useful for individual products or groups of products.

In this thesis we will focus on short-term forecasting, specifically on forecasting daily sales. The short-term forecasts are mostly used to determine the amount of goods that should be ordered from suppliers, but they might be also used for other purposes such as managing the number of employees that must be present in the store in the next few days. The products that need to be predicted on daily bases are usually groceries with short durability such as meat, fruit and vegetables, dairy products, baked food and also some drugs or toiletries. These products are often referred to as fast-moving consumer goods (FMCG).

The short-term forecasts are usually made at stock keeping unit (SKU) level. The SKU specifies a product and all its attributes distinguishing it from other products such as flavor, color, manufacturer, packaging size etc. It also includes the specific store in which the item is sold, since the sales of the same product might vary significantly in different locations.

Daily sales are much harder to forecast than monthly or weekly sales since

aggregation removes a lot of random noise.[1] With daily sales usually come lower volumes, high volatility and they are also affected by a large number of factors such as holidays, promo actions or weather. Daily sales time series often have trend pattern and multiple seasonal patterns such as weekly seasonality or yearly seasonality. The season is not even necessarily related to a strict period of time, it can be affected by other factors such as moving holidays (Easter) or climate conditions (i.e. the first warm weekend in a year). Another feature of sales time series is that they often contain outliers - an unrepresentative observations. It is often not easy to distinguish outlier from a pattern that should be considered when predicting future sales. Additionally, there is a question of how to handle historical stock-outs. There are obviously zero sales during stock-outs, however it does not mean that there was no customer demand. All these factors need to be considered in order to compute sufficiently accurate forecast.

To forecast daily sales, it is necessary that the sales are very frequent. Products that are not sold on daily bases are not suitable for this type of predictions because of large noise and irregularity. In practice, the stores usually order these kind of products less frequently for a longer period of time and it is sufficient to forecast weekly or monthly sales, or even yearly sales in some specific cases.

4.2 Data

There is not a lot of scientific research done in the field of sales forecasting. The problem is that companies are not willing to share their data, not even with their own suppliers. For the purpose of this thesis we mainly used data provided by Logio company. We collected sales data from 27 different stores, mostly supermarkets, some of them with multiple physical stores. Altogether we gathered sales data of approximately 18 millions SKUs differing in many factors such as product type, product life time or sale frequencies. It is very rare to obtain dataset of such a size in this field.

The data provided by Logio company was used for the most of our experiments, however it is not public. We were also allowed to use a public dataset provided by Rossmann, a major European Pharmacy retailing company, for a kaggle.com competition. This dataset contains daily sales of 1,115 Rossmann's drug stores located across Germany. The sales in this dataset are aggregated by

[1] A coefficient of variation defined as ratio of standard deviation to mean of the series is a good measure of volatility or uncertainty. The higher this value is, the higher is volatility. Consider, for example, two normal distributions N(270, 80) and N(150, 65). Their coefficients of variations are 0.3 and 0.43 respectively. By aggregating these distributions we obtain a distribution N(420, 103.1) with coefficient of variation approximately 0.25.

stores and are therefore quite different from data provided by Logio company. We used this dataset to inspect how well do our models generalize to different data and also to present our models.

The Rossmann dataset consists of sales aggregated by stores thus all the time series are very frequent and suitable for daily sales forecasting. The only flaw in the data was missing sale records for several stores. We filled this gaps with zero sales to obtain dense time series, containing a sales record for each day between the start and the end of the series.

While the Rossmann dataset was almost ready to be used in our experiments, it was necessary to process and analyze the data provided by Logio company. We aggregated all the sales by product (SKU) and day to determine daily sales of each product and filled the days with no sales with zeros. After that, we eliminated short time series and series with low frequencies and chose 10,000 products to create a dataset that we will refer to as Multistore dataset. We additionally cut away a randomly long part of the end of those time series that will be used for validation and testing purposes (see Section 5.1). We did this because most of the series ended in the same few days (when the data was collected). These last days of the series were used for evaluations and testing and we wanted to make sure the endings are randomly distributed during the year and our results are not affected by any seasonality.

Since daily sales of 10,000 products produce a huge amount of training data, we decided to also create a smaller and simpler version of Multistore dataset. The idea was to first determine the most suitable models using the small Multistore dataset and then train them using the full Multistore dataset. For our small dataset we separated 2,000 SKUs from a single store and processed their time series so that they covered the same time window.

4.3 Exploratory Data Analysis

We have already described how we obtained three datasets of sales time series. In this section we analyze each one of them and provide an overview of the time series used in our experiments.

4.3.1 Small Multistore Dataset

The small dataset consists of 2,000 time series which are daily sales of the 2,000 most frequently sold items in a single supermarket (covering several physical stores). These time series are dense, meaning that missing data in the middle of the series were replaced by zero sales. Each time series consist of 1,122 time

steps and they come from the same time range. The dataset contains 91 negative values which are caused by returns. These values occur rarely, so they would not affect our results and we will only make sure they will not be used for testing purposes.

As Figure 4.1 shows, daily sales follow approximately an exponential distribution.

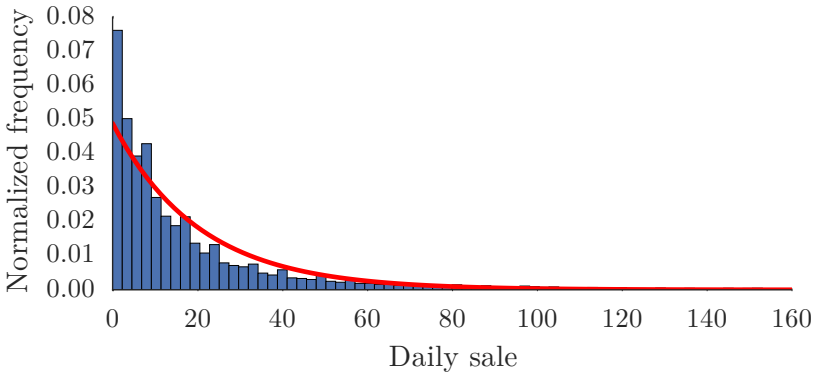


Figure 4.1: Histogram of daily sales representing 95% of the data. Negative values and the most extreme values were removed before plotting for better clarity. The red line represents exponential distribution that we fit the data to.

Sales time series often contain patterns that are affected by season or big holidays such as Christmas. As a result, the average daily sales vary in different months. Even though the variations would be stronger for specific products or groups of products, even the average daily sales over all products are higher in some months than in the others (see Figure 4.2).

In addition to variations caused by season and holidays, daily sales are also strongly affected by weekdays. Significant within-week sales swings are shown in Figure 4.2. As can be seen there is a large peak on Fridays which can be explained by customers often shopping for a whole weekend on Friday. On the contrary, mean Sunday sales are significantly lower than any other day of week.

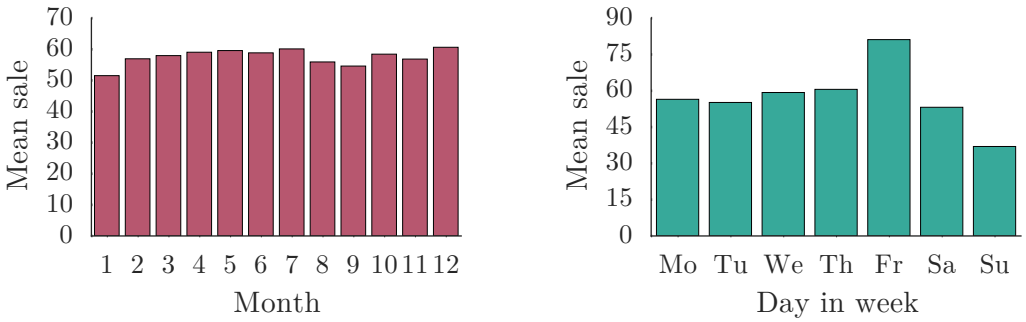


Figure 4.2: Aggregated mean sales computed over the whole dataset.

The dataset consists of 2,000 time series corresponding to the daily sales of 2,000 different SKUs. Each of the series follows different patterns, consists of its own specific trend and seasonal components. In addition, the average sales of each item are very different (see Figure 4.3). Besides the differences, there is a pattern that all of the series follow in a similar manner - holidays. Figure 4.4 show how most of the series produce a huge peak a few days before Christmas and New Year's Eve and then there are no sales on December 25th and January 1st.

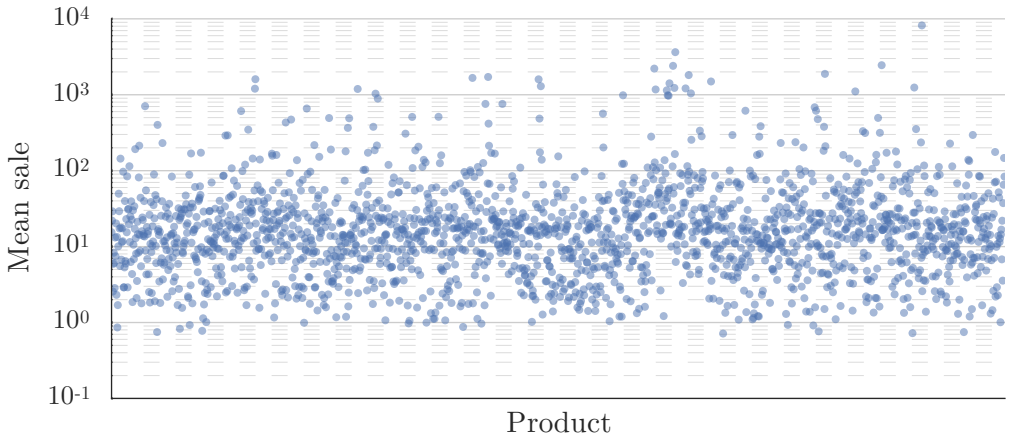


Figure 4.3: Average daily sales by product.

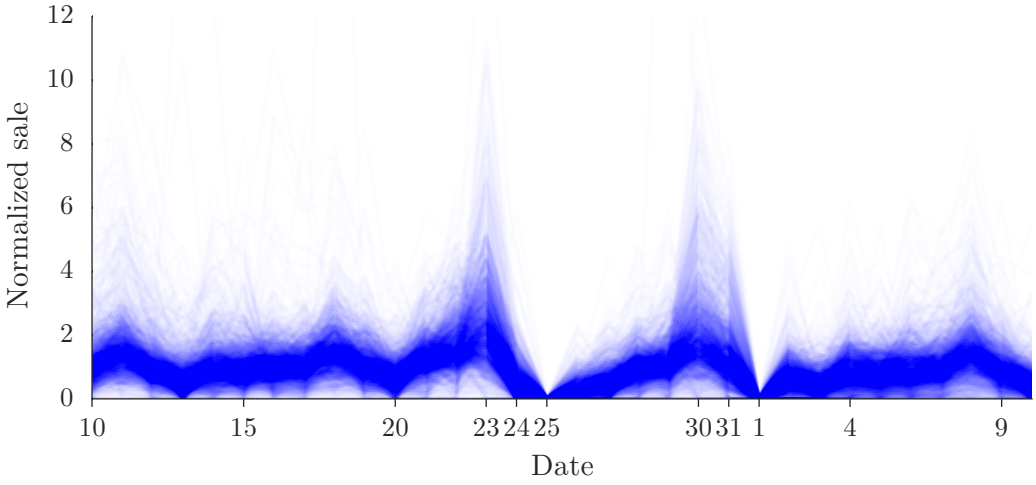


Figure 4.4: The plot captures the period around the Christmas holidays and covers all the time series from the dataset normalized by their mean. The strong peaks are one and two days before Christmas and New Year's Eve

The last aspect of the time series that we analyzed was autocorrelation - the correlation of a time series with its own shifted values. We studied the autocorrelation of a number of time series in our dataset and came up with two general patterns, see Figure 4.5, that occur with only small diversities. In the first pattern, the largest correlation is between subsequent days and then it is slowly decreasing for a while with increasing day shift. There is usually also slightly higher correlation with other days corresponding to the same weekday (shifts which are multiples of 7). This pattern explains the effectivity of naive forecasts. The second pattern shows very large correlation with days of the same weekday and a bit smaller correlation with surrounding days.

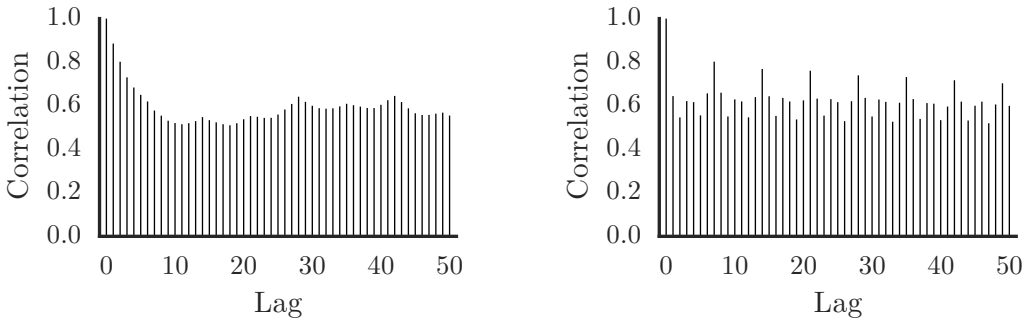


Figure 4.5: When observing autocorrelation of the series, we found two general patterns. These two example autocorrelations represent these patterns.

The mean autocorrelations can be found in Figures 4.6 and 4.7. The later figure represents a longer time period and shows that the correlation is slowly decreasing with longer time lag but the pattern remains the same. There is also a slightly visible peak near the end of the series corresponding to the correlation with same day in the next year.

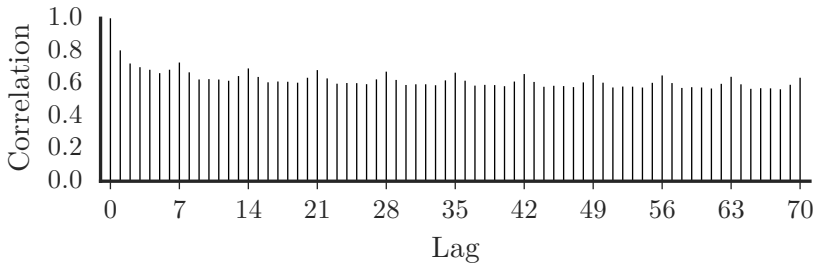


Figure 4.6: Mean autocorrelation with maximum lag of 70.

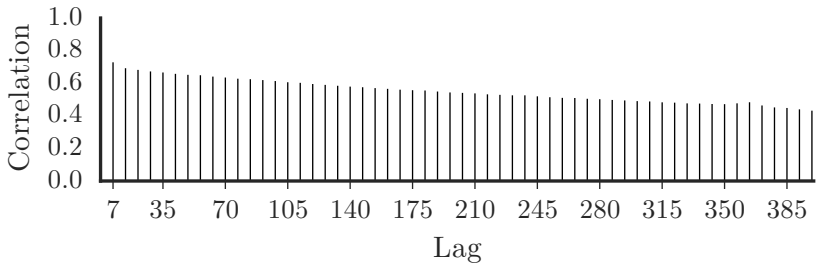


Figure 4.7: Mean autocorrelation with maximum lag of 400. Only every 7th lag is shown.

4.3.2 Large Multistore Dataset

We carried out a similar analysis on the large dataset with similar results. This dataset consists of 10,000 time series representing daily sales of various products from multiple stores. These series were also filled to become dense. Again, daily sales follow approximately exponential distribution (see Figure 4.8), but the values are somewhat smaller than values in the small dataset.

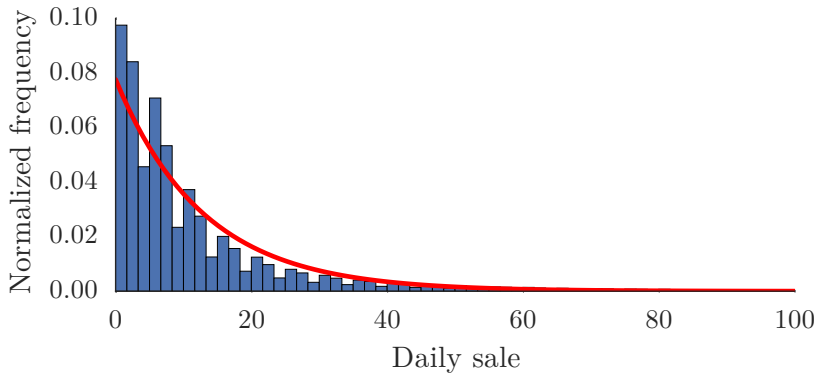


Figure 4.8: Histogram of daily sales representing 96% of the data. The most extreme values were removed before plotting for better clarity.

Series in the large dataset do not come from the same time range and they are not of the same length. Their lengths vary between approximately 1 and 4 years. A histogram of time series lengths is shown in Figure 4.9.

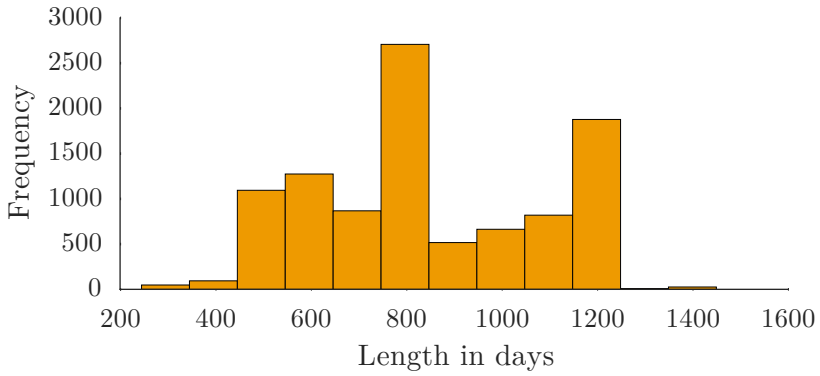


Figure 4.9: Histogram of time series lengths.

Mean monthly and weekday sales in the large dataset show the same patterns as in the small dataset (see Figure 4.10). The sales in January are again considerably smaller than sales in other months. This can be explained by people saving money after Christmas and many people are also trying to shop less in order to meet their New Year's resolutions such as losing weight or spending less money.

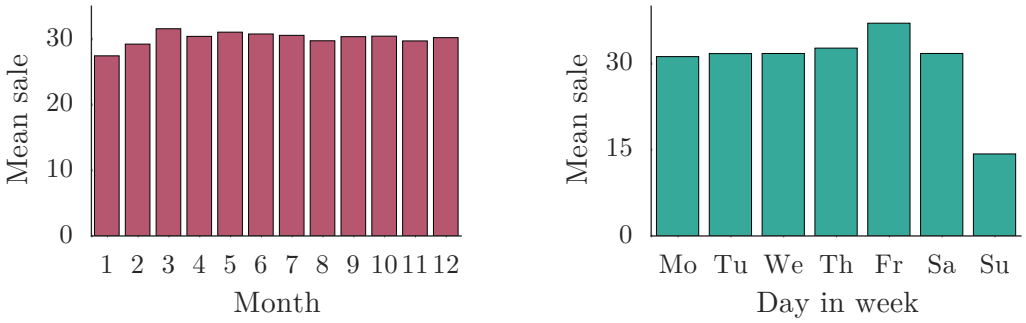


Figure 4.10: Aggregated mean sales computed over the whole dataset.

In addition to daily sales, the large dataset also contains sale price for each product and day. Unsurprisingly, we were able to detect a significant correlation between price and amount sold. Figure 4.11 shows a histogram of correlations over all time series. Negative correlation, which dominates the dataset, usually corresponds to the effect of promo actions where even small decrease in price causes large increase of amount sold. There are however also products with positive correlation of price and amount sold. These cases are usually connected with highly seasonable products that are sold in large amounts during a specific period of time for example before Christmas. Their price, which was regular or even higher during this period, is significantly lowered after the season but sales go down too.

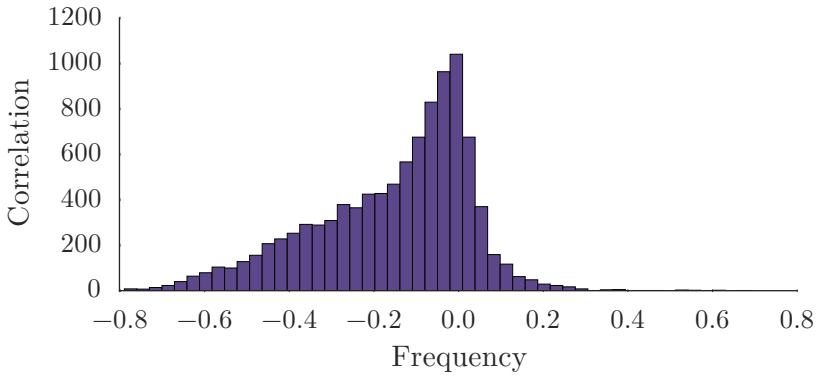


Figure 4.11: Histogram of correlations between price and sold amount over all time series.

4.3.3 Rossmann Dataset

Data in the Rossmann dataset were provided by Rossmann for a kaggle.com competition and they allowed us to use this data for the purpose of this thesis. Data

in this dataset are quite different from the Multistore dataset since it contains daily sales aggregated by stores. These data are public.

The dataset consists of 1,115 time series representing aggregated daily sales of 1,115 stores. The missing dates were filled with zeros to obtain dense time series. Each time series is 942 days long except for one, which starts one day later. All the other series come from the same time range between January 1st, 2013 and July 31st, 2015. Daily sales in Rossmann dataset follow a gamma distribution. A histogram of these values is depicted in Figure 4.12.

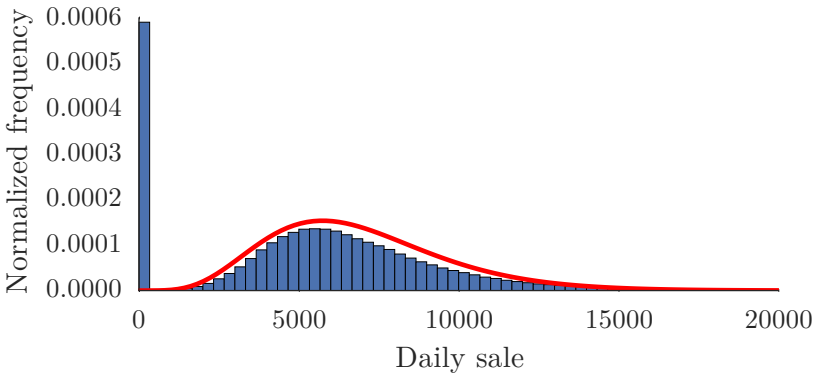


Figure 4.12: Histogram of daily sales. Zero values where excluded when fitting to gamma distribution.

The patterns in mean monthly and weekday sales in the Rossmann dataset are more significant (see Figure 4.13). The sales in December are apparently larger than sales in any other month. This corresponds to the nature of the stores - Rossmann products are often used as a Christmas gift, while not many people buy Christmas gifts in a supermarket. It can be also noticed that there are almost no sales on Sundays. The reason is that most of the Rossmann stores are closed on Sunday.

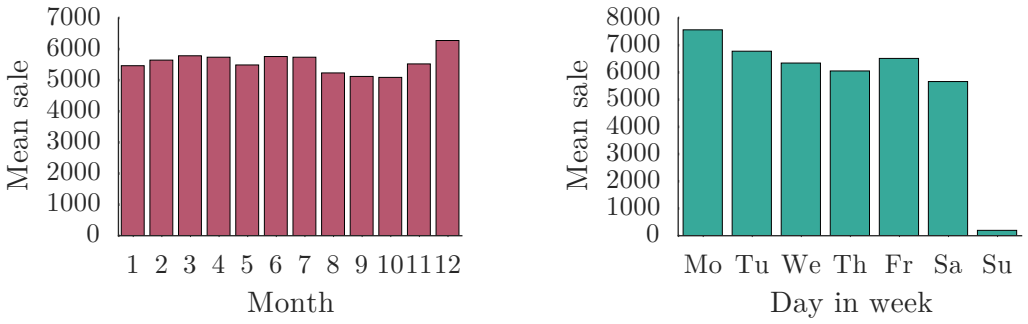


Figure 4.13: Aggregated mean sales computed over the whole dataset.

In comparison to large differences between mean daily sales of each product, mean daily sales of each store are not that different; see Figure 4.14. Also, as shown in Figure 4.15, the autocorrelation with lags that are multiples of 7 is significantly higher in this dataset.

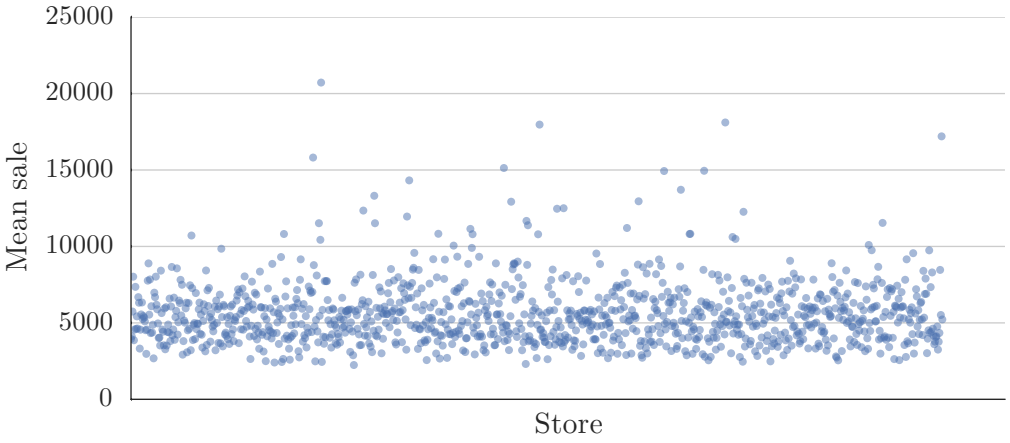


Figure 4.14: Average daily sales by store.

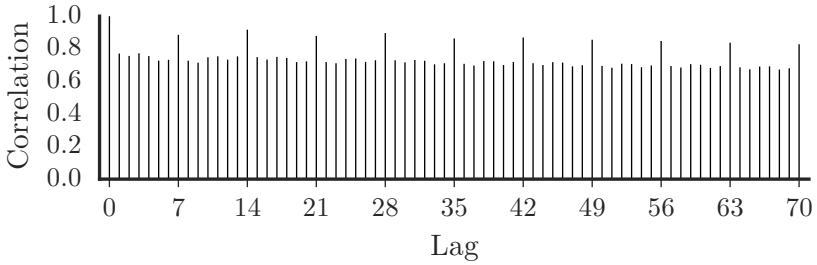


Figure 4.15: Mean autocorrelation with maximum lag of 70.

5. Experiments

Using the datasets described in Chapter 4 we conducted various experiments in order to find the most appropriate model for predicting next 7 days of sales time series based on the past sales. Our goal was to train a universal model, that would be able to predict sales of a product without being trained on its history. This would allow very fast predictions, after the model has been trained sufficiently.

Our models were prepared in Python using Keras[1] library. We used the Multistore dataset described in Section 4.2 to find and prepare the best model. First, the small variant of the dataset was used to train a large number of models. The models that have shown the most promising results were then also trained using large data. We compared their behavior and analyzed their performance in contrast to three other forecasting methods. In addition, we trained the best models on the Rossmann dataset (Section 4.2). We examined how well do our trained models generalize and analyzed whether it makes sense to fine-tune pre-trained model to forecast Rossmann time series, which are significantly different from Multistore data.

5.1 Model Evaluation

To select the best model and compare its performance to other models or different forecasting methods, it is necessary to correctly evaluate model's accuracy. When determining the accuracy, it is especially important that the measurements are unbiased and made using new data that has never been seen and used before. To achieve unbiased evaluations, we split both Multistore dataset and Rossmann dataset into three parts - training set, validation set and test set. Each experiment can be then divided into three steps - training, evaluation and test phase. In this section we depict these tree steps and describe which error functions we chose to be used for evaluations and tests.

5.1.1 Training Phase

Training the model means finding the appropriate parameters. In case of neural networks, it means to find network weights that produce the best results on the *training set*. The objective of our training was to minimize MSE on time series from the training set which is done using backpropagation algorithm described in Section 1.1.2 or some of its modifications.

[1] http://keras.io

5.1.2 Evaluation Phase

By evaluating the model we were able to determine how well it learns and performs in comparison to other models. The evaluation consists of two parts.

First of all we need to be able to determine whether the model itself is learning anything and whether it is not overfitted - behaving very well on training data, but not able to generalize to unseen data. We do this by regularly computing loss on the validation data set and portion of the training set that we refer to as an evaluation set. These losses are computed during the training phase, after each epoch. If the loss computed on the evaluation set is not decreasing, it means that the model is not learning anymore. If it is decreasing for the evaluation set, but not decreasing for the validation set, it means that the model is overfitting.

It is common to compute the loss over the whole training set during training instead of computing it in on evaluation set after each epoch. We used the later method even though it is slightly more time consuming. We believe it makes more sense to compute both losses on equally trained model.

The second part of evaluation is actual evaluation of model's predictions. After the model is trained, we compute the predictions of the last few time steps of the time series from validation set. These are the data that the model has not been trained on. If the model predictions are scaled or normalized in any way, it is necessary to postprocess them so that the predictions correspond to the actual amount that will be sold. Only that way we will be able to compare prediction errors of different models with different preprocess methods. Based on this comparison made on the validation set, we can choose the appropriate model.

5.1.3 Test Phase

It is important that the actual final model performance is evaluated on different data than the model was trained on and chosen by. This would bring an unwanted bias into our result. We therefore use the test set to do the final evaluation of the models' accuracy by predicting the last few time steps of each time series.

5.1.4 Error Functions

To evaluate accuracy of models it was necessary to choose unified criterion. We considered all the error functions mentioned in Section 2.3 and decided to use RMSE, SMAPE and MIBRE. While NRMSE seems to have the best properties (as shown in Table 2.2), its normalization based on average of the whole series is very unstable.

The chosen error functions were not used in the same manner. It is not possible to use RMSE to compare models over multiple different time series because it is scale dependent. We can however use it to compare behavior of multiple models on a single (potentially specific) time series and find out their strengths and weaknesses. To compare models over different time series we used SMAPE and MIBRE. Most of the times smaller SMAPE value corresponded to smaller MIBRE value. Thus, for the purpose of this thesis we mostly use SMAPE for model comparison since it is clearer and more easily interpretable.

5.2 Data Preparation

As described in previous section, we first needed to randomly split the datasets into three disjunctive sets of data, namely training set, validation set and test set. In case of the large dataset this led to 2,000 time series in both validation and test set and 6,000 time series in training set. In case of the small dataset, we did not use the test set since we only wanted to choose the most promising models. Their accuracy was evaluated on the test set extracted from the large dataset. Note that the test set in large Multistore dataset was carefully selected so that it does not contain any time series from the small Multistore dataset. We hence split the small Multistore dataset into a validation set with 400 time series and training set with 1,600 time series. The Rossmann dataset that contains 1115 time series was split into train set consisting of 781 time series and test set and validation set both of size 167.

The length of our time series varied from around 500 days to around 4 years. To predict sales for the next seven days it is enough to consider a much shorter history. Too much historical data might be misleading and additionally, given a too long past, neural networks would need to learn a lot of parameters and it might lead to a problem with learning. We decided to train the network to predict the next seven days based on 28 and 91 days long look back. These numbers represent 4 and 13 weeks long period and the divisibility by length of the week is made on purpose to help the network with weekly seasonalities.

To utilize the whole time series, we split it into many small training series. The splitting can be depicted as a sliding window shifting by a single day and providing a new time series with each shift. The series created by this process were used for training and for the first part of evaluation. The second part of evaluation and tests were only made by predicting the ends of the time series. The process of splitting significantly increased the amount of training data. We were able to extract around 1,640,800 training examples (the exact number depends

on the look back) out of the 1,600 training series from the small dataset. Out of the 10,000 large dataset training time series we extracted 4,918,502 training examples.

Preprocessing is a crucial part of neural networks training. Chapter 4 shows that there are significant differences between individual time series in our datasets. To be able to create a universal forecasting model, data normalization is necessary. In addition, as described in Section 1.1.3, it is helpful to preprocess the network input as it usually increases the learning speed. Besides that, it is often necessary to preprocess the network outputs as well so that the network is actually able to output the correct values.

We experimented with several ways of preprocessing on our small dataset however it took a lot of time to preprocess each input series right before feeding into the network. We therefore decided to choose a few preprocessing methods, store preprocessed series and experiment with different models using these preprocessed data. It is worth noting that for the second part of evaluation and testing it was not possible to use these preprocessed data. It was necessary to preprocess the data right before feeding into network so that we were able to apply an inverse operation to the result to obtain the actual prediction. As a side effect we were able to observe how much time do the predictions take overall.

We used two traditional and developed two novel preprocessing methods to use in our experiments with different models:

Min-max normalization (MMN) transforms data into new range $[A, B]$ using the following equation:

$$\mathrm{MMN}(y_i) = \frac{y_i - \min}{\max - \min}(B - A) + A$$

Where min and max are minimum and maximum of the series respectively. This preprocessing transforms each series into the same interval of values. Normalizing the series and especially the output values into interval $[0, 1]$ would allow us to use logistic sigmoid as an activation function for the output layer. To be able to predict values larger than maximum or smaller than minimum of the series, we normalized the series into interval $[0.2, 0.8]$.

Z-score normalization (ZN) is a commonly used normalization that transforms all the input vectors to a common scale with zero mean and standard deviation of one. This is done using following equation:

$$\mathrm{ZN}(y_i) = \frac{y_i - \mu}{\sigma}$$

where μ is the mean of the series and σ is standard deviation. The values in processed series are concentrated around zero. If we use the same preprocessing for the output series, we could use for example hyperbolic tangent as

an activation function for the output. We also experimented with modified hyperbolic tangent, for example with doubled values that allow to output values more deviated from the mean of the series.

Normalized weekday difference (NWD) preprocessing was proposed in order to help the network and manually pre-extract a specific features of sales time series - weekday regularity. We computed mean sales for each day of week and used normalized difference between actual sale and average sale for corresponding weekday both for network inputs and desired outputs. This should relieve the network from need to learn this strongly problem-specific feature and allow it to concentrate on other aspects of provided time series. It was very important to compute the weekday mean correctly and only include the past time steps of the series, otherwise we would bring a bias to our data. The resulting series may contain values of an arbitrary size so we additionally scaled them using MMN preprocessor.

Weekday ratio (WR) is similar to the NWD preprocessing however it uses ratio of the sales to corresponding weekday. We suggested this form of preprocessing since it additionally serves as a normalization.

5.3 Benchmarks

This section describes forecasting techniques that were used as benchmarks for our experiments.

Exponential Smoothing

Our first benchmark method is exponential smoothing. This method is commonly used for sales forecasting in practice since it works well generally and can provide accurate forecasts, comparable to those obtained with more complex methods.

We used an existing implementation of exponential smoothing in R programming language called ETS [55]. It is a very sophisticated implementation covering 15 separate exponential smoothing methods including simple exponential smoothing, Holt's linear trend method and both additive and multiplicative Holt-Winter's exponential smoothing. The ETS model automatically selects the best exponential smoothing method for given time series and also determines unknown parameters of the time series by minimizing the squared prediction error.

As Figure 5.1 shows, it is quite easy for this method to correctly predict time series that are regular, however irregular time series can often produce significant errors.

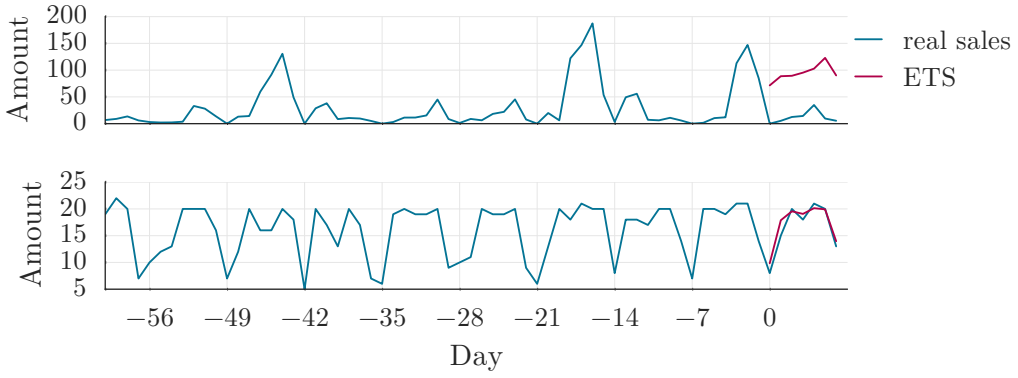


Figure 5.1: Example predictions of exponential smoothing with exceptionally high (upper series) and low (lower series) SMAPE error value.

Random Forests

As our second benchmark we chose random forests for their robustness and accuracy. We used their Scikit-learn [78] implementation with unlimited depth of the trees and trained a separate estimator for each day of forecasting horizon. The data that was fed into forest was not normalized in any way.

We found out that it was not optimal to fit this model using all the data since the training speed increased linearly to the input size, however the error produced was not decreasing any more after a sufficient amount of training data was presented to the model (see Figure 5.2).

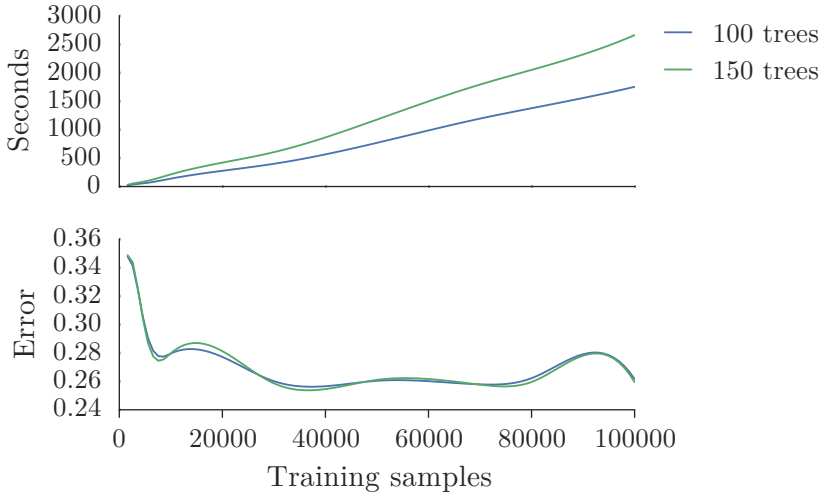


Figure 5.2: Error and training time of random forests method with increasing number of presented training data. The error is computed on validation set from small Multistore dataset.

Weekday Average

As the last benchmark we used a simple model that predicted the sales to be the average sale of the corresponding day of week. We suggested this benchmark to ensure that our models with WR and NWD preprocessors would not simply learn to predict zero difference or unit ratio. As we will see, this model produced quite bad results and it is easy to outperform it.

5.4 Models

To determine whether neural networks and especially deep neural networks are a suitable model for sales prediction, we tried various neural network architectures combined with several preprocessing methods. For each model, we tried different numbers of layers, neurons, different learning rates, learning methods or other parameters in order to find the appropriate settings. Since deep neural network models are quite computationally expensive, even when training on GPU as we did, it is only possible to try a limited amount of variations. We tried more than 180 different architectures and settings and we believe that this number is sufficient enough to back up our results.

The first class of models that we tried were simple FNNs with a single hidden layer. This architecture with a simple min-max or z-score normalization was already applied to a similar problem [65] and we will use it as a neural network based benchmark.

We then decided to use recurrent neural networks (see Section 1.2.1), for these models are theoretically the most suitable ones for time series processing. There are several options from which we selected traditional SRNN, LSTM network and GRU network. We compared these models to each other, to simple feedforward neural networks and to our benchmark model. In our comparisons we focused on the ability to accurately predict our sales time series but we also compared the speed of learning and prediction.

Another deep learning model that seemed promising were convolutional neural networks (see Section 1.2.2). CNNs might be able to detect repeating patterns in time series or even extract features that are not that easily spotted by human or statistical models. We tried various architectures varying in amount of filters, their sizes and numbers of layers. Similarly as for previous models, we also tried different kinds of parameter settings to discover the most suitable combination.

Besides standard models, we also tried several hybrid models that combined recurrent and convolutional layers in various manners. Out of these models we selected an architecture that seemed to work the best and tuned the parameters

of this network.

Along with exploring different neural network architectures, we also experimented with network inputs. First of all it is possible to change *look back* - the number of time steps that are fed into network and used for prediction. We experimented with 28 days (4 weeks $\sim$ 1 month) and 91 days (13 weeks $\sim$ 3 months). Next to the look back, we also tried various time series preprocessing described earlier in Section 5.2.

5.5 Small Dataset Experiments

In this section we depict the experiments that we conducted using the small Multistore dataset. These data were used to train many variations of selected models differing in preprocessing, architecture and learning parameters. In the end of the section, we describe the best models that we found for each type of deep neural network and compare their performance.

5.5.1 Simple Feedforward Neural Network

The first type of neural network we tried to train was a simple feedforward neural network. FNNs are commonly used and they have been previously used to forecast sales time series.

We experimented with network parameters, activation functions and learning rule variations and prepared some of the models to be very similar to networks implemented by Thiesing and Vornberger in 1997 [65]. Our models also consisted of a single hidden layer and among others we tried to use the same preprocessing method. We also used backpropagation algorithm with momentum. The main differences between our networks and their networks was in the provided data - while they used monthly time series, we forecasted daily time series and we used no other indicators such as price or advertising campaigns.

When experimenting with number of neurons in hidden layer, we found 100 to be a sufficient amount. Most of our experiments with feedforward neural networks therefore use 100 hidden neurons. We tried all of our 4 preprocessing methods with both stochastic gradient descent and RMSProp optimizer while adjusting the learning rate. The overall results can be found in Table 5.1.

	look back	RMSE	**SMAPE**	MIBRE
min-max normalization (MMN)	28	232.7	**0.2421**	0.3482
z-score normalization (ZN)	28	321.7	**0.2459**	0.3527
normalized weekday difference (NWD)	28	322.3	**0.2531**	0.3602
weekday ratio (WR)	28	315.1	**0.2417**	0.3487
min-max normalization (MMN)	91	306.2	**0.2343**	0.3391
z-score normalization (ZN)	91	264.9	**0.2303**	0.3336
normalized weekday difference (NWD)	91	313.5	**0.2488**	0.3560
weekday ratio (WR)	91	288.0	**0.2425**	0.3504

Table 5.1: The best mean error achieved by FNN on validation data for each preprocessing method.

We did not achieve very good results with NWD preprocessor however the WR preprocessor worked very well in combination with softplus activation function in the output layer and logistic sigmoid activation function in the hidden layer. With look back of 28 time steps, we were able to achieve the best results with this preprocessor.

Very good results were also achieved using the MMN preprocessor using RMSProp optimizer with quite large learning rate of 0.1. This network was not learning very well (see Figure 5.3) but surprisingly it consistently resulted in smaller error than networks trained using smaller learning rate.

We also tried the ZN preprocessor which transforms the series so that they have zero mean and standard deviation equal to one. Both inputs and outputs are therefore values around 0. To allow the network to produce this type of outputs we propose to use a modified hyperbolic tangent function that doubles the output value. We will refer to this function as doubled hyperbolic tangent (DHT). This activation function produces outputs in range $[-2, 2]$. Among other variations that we tried, DHT activation function worked out the best. With ZN preprocessing, look back of 91 days and RMSProp optimizer with learning rate 0.0005 we achieved the best results out of all the FNN models.

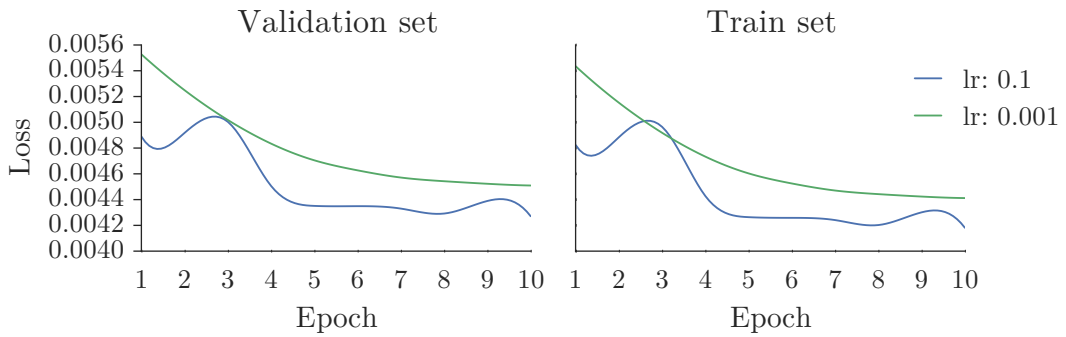


Figure 5.3: Learning curves of a FNN with MMN preprocessor trained with RMSProp using different learning rates. Learning rate of 0.1 leads to problems with learning, however it consistently led to better results. Note that the error values produced on train set cannot be compared to error values produced on test set. The reason is that each set contains different time series and MMN preprocessor normalizes each series in very different way - a large outlier in the series may cause the preprocessed amounts to be very small leading to smaller MSE which is the objective function of the network.

As can be seen in Table 5.1, when comparing the results for 28 days long look back, the best SMAPE was achieved by WR preprocessing. However, a significantly smaller RMSE was achieved by MMN preprocessing (with a large learning rate as mentioned before). We compared the actual predictions of both networks and analyzed them. In many cases the predictions of the network with WR preprocessor seemed more legitimate and valid and in our opinion, they exploited the features of the series in a greater way even though the error was larger (see Figure 5.4). This confirms that using SMAPE as a primary objective is a good choice.

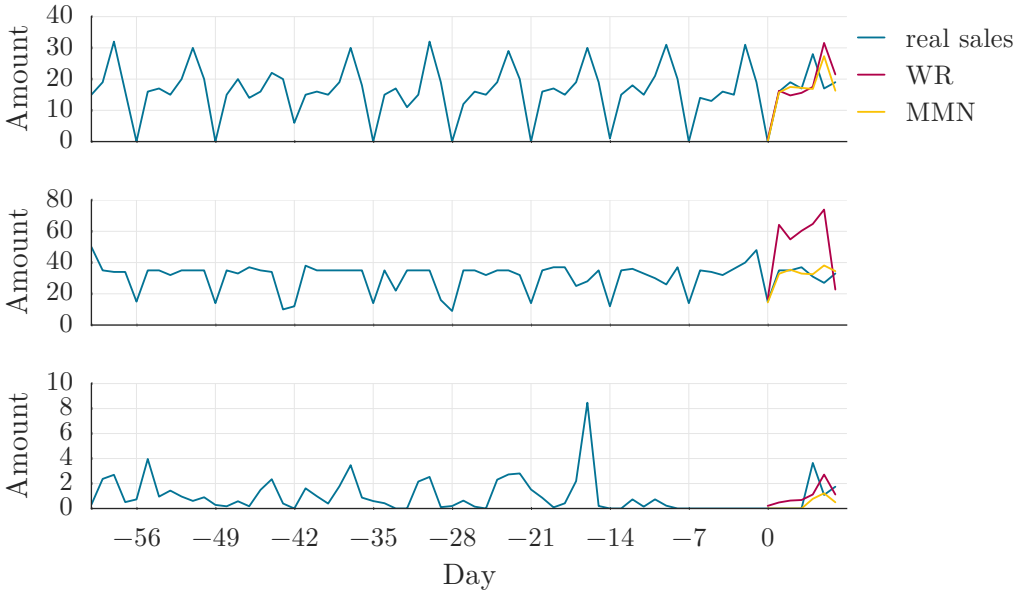


Figure 5.4: Example time series where usage of WR preprocessing resulted in larger error than usage of MMN preprocessing. Even though the error is higher, we believe that predictions with WR preprocessing seem very reasonable since they follow the pattern that occurred in the past or current trend.

5.5.2 Recurrent Networks

As a next step we tried three basic variants of recurrent neural networks, namely simple recurrent network (SRNN), GRU and LSTM all described in Section 1.2.1. Recurrent neural networks are usually used with time series data and therefore it makes sense to analyze their applicability to sales forecasting.

The recurrent models consisted of an input layer, hidden layers consisting of recurrent units and a dense layer that processes the output from recurrent units into the actual prediction. We combined all three variants of recurrent units with our preprocessing methods, tuned the parameters and learning process and we even tried dropouts (see Section 1.3) in several cases where the model seemed to be overfitting. We have tried many architectures differing in number of hidden layers and neurons in hidden layers. Again, we decided to work mostly with a single layer of recurrent units. Adding more layers produced only negligible improvement in overall errors but it significantly affected the learning speed.

We discovered that recurrent networks did not behave very well when trained using simple gradient descent algorithm, RMSProp algorithm gave noticeably better results. We also found out that these networks take considerably longer time to train than FNNs. The training speed of LSTM and GRU was around 10x slower on average but it varied a lot depending on the architecture and number of

hidden neurons. The training speed of SRNN was around 2x slower than training speed of FNNs.

The smallest error values achieved for each preprocessing method can be found in Table 5.2. There was a surprising difference between results with look back of 28 and 91. In the first case, the best results were achieved using WR preprocessor, however this preprocessor produced the worst results in case of 91 days long look back.

Compared to FNN, SRNN did not provide much different results. We examined some of the time series that produced the largest error difference between these two models and realized, that many times the difference was only caused by a vertical shift in prediction (see Figure 5.5).

	look back	RMSE	**SMAPE**	MIBRE
min-max normalization (MMN)	28	324.3	**0.2461**	0.3538
z-score normalization (ZN)	28	325.3	**0.2423**	0.3486
normalized weekday difference (NWD)	28	330.3	**0.2507**	0.3574
weekday ratio (WR)	28	327.9	**0.2404**	0.3472
min-max normalization (MMN)	91	313.8	**0.2376**	0.3432
z-score normalization (ZN)	91	297.4	**0.2342**	0.3397
normalized weekday difference (NWD)	91	305.5	**0.2445**	0.3522
weekday ratio (WR)	91	314.3	**0.2457**	0.3543

Table 5.2: The best mean error achieved by SRNN on validation data for each preprocessing method.

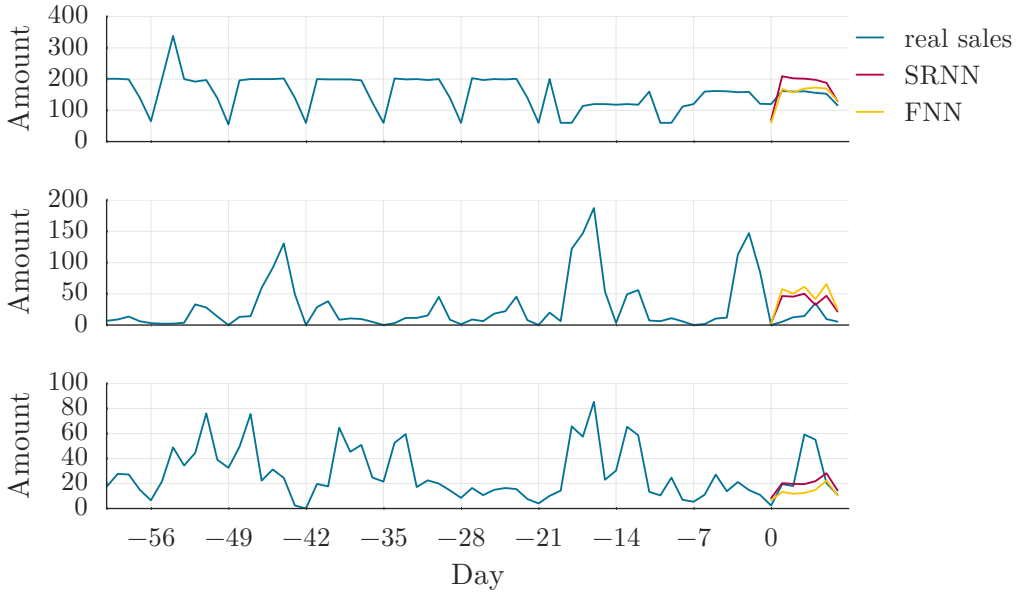


Figure 5.5: Example time series with significant error difference between FNN and SRNN model predictions.

The LSTM network which is more complex than SRNN produced only slightly better results (see Table 5.3). Considering that the training speed is much worse, the overall performance of this network is not very good. Here we experimented with retaining network state between individual input samples corresponding to subsequent parts of the time series. The idea was to allow the model to exploit patterns occurring in a much longer period of time than 28 or 91 days that we used as a look back size. These experiments however did not lead to superior results.

	look back	RMSE	**SMAPE**	MIBRE
min-max normalization (MMN)	28	332.4	**0.2470**	0.3551
z-score normalization (ZN)	28	323.2	**0.2392**	0.3451
normalized weekday difference (NWD)	28	329.5	**0.2471**	0.3529
weekday ratio (WR)	28	323.8	**0.2348**	0.3398
min-max normalization (MMN)	91	303.5	**0.2413**	0.3488
z-score normalization (ZN)	91	291.6	**0.2378**	0.3444
normalized weekday difference (NWD)	91	317.2	**0.2451**	0.3513
weekday ratio (WR)	91	264.5	**0.2340**	0.3395

Table 5.3: The best mean error achieved by LSTM on validation data for each preprocessing method.

The GRU network showed the best results out of all recurrent neural network

models. Even though GRU models are less complex than LSTM models, their superiority has been also observed in other experiments [79]. Similarly as before, the best results were achieved using WR preprocessor with softplus activation function in the output layer and ZN preprocessor with DHT activation function. The error comparison can be seen in Table 5.4.

	look back	RMSE	**SMAPE**	MIBRE
min-max normalization (MMN)	28	317.6	**0.2497**	0.3582
z-score normalization (ZN)	28	318.3	**0.2375**	0.3429
normalized weekday difference (NWD)	28	325.7	**0.2515**	0.3579
weekday ratio (WR)	28	332.4	**0.2381**	0.3443
min-max normalization (MMN)	91	288.5	**0.2414**	0.3466
z-score normalization (ZN)	91	289.3	**0.2343**	0.3399
normalized weekday difference (NWD)	91	308.5	**0.2496**	0.3561
weekday ratio (WR)	91	235.5	**0.2293**	0.3332

Table 5.4: The best mean error achieved by GRU on validation data for each preprocessing method.

We inspected the predictions of the best GRU network and compared them to predictions of FNN and LSTM network. As can be seen in Figure 5.6 the GRU network often exploited features of the series that were not discovered by any of the two other networks. This is the desired behavior of deep neural network models.

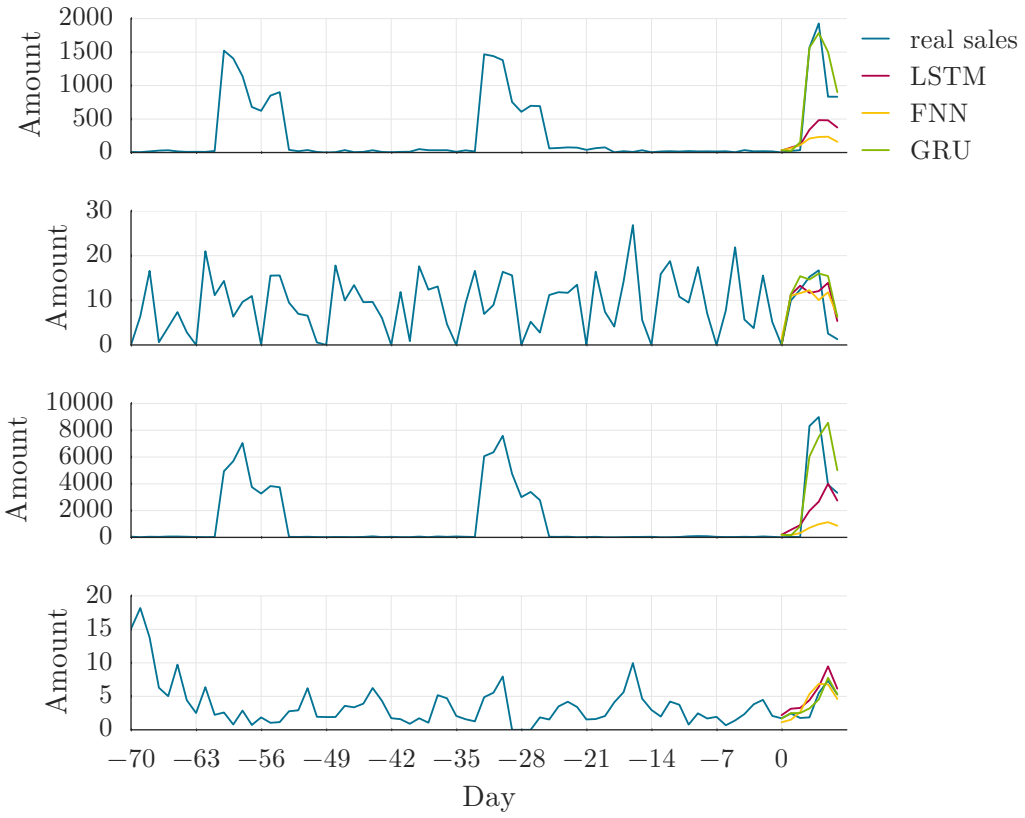


Figure 5.6: Example time series where the best GRU network significantly outperforms both the best LSTM network and the best FNN.

5.5.3 Convolutional Networks

The CNNs are known for their ability to extract various important features from input data. We trained such a network on our data, extended by a dense layer on the top. This dense layer should combine the features revealed by convolutional layers and perform prediction,

Although the CNNs also outperformed all our benchmark models, they did not produce results as good as FNN or RNN models did. We experimented with different activation functions, various number of convolutional layers, kernel sizes and numbers of kernels. We found out that in our case, increasing number of convolutional layers actually did not lead to significantly better results. What improved the error was increasing the number of convolution kernels, but only to some point. After reaching a specific number of kernels, different for each preprocessing method, the error did not improve any more.

The best results were achieved using ZN preprocessor with 91 days long look back. This network consisted of a single convolutional layer with 16 convolutional kernels of size 7 and a max-pooling layer with pooling regions of size 2. The over-

all error comparison of convolutional networks combined with our preprocessing methods can be found in Table 5.5.

	look back	RMSE	**SMAPE**	MIBRE
min-max normalization (MMN)	28	335.3	**0.2713**	0.3835
z-score normalization (ZN)	28	323.0	**0.2457**	0.3526
normalized weekday difference (NWD)	28	325.2	**0.2665**	0.3761
weekday ratio (WR)	28	319.3	**0.2423**	0.3481
min-max normalization (MMN)	91	316.9	**0.2661**	0.3775
z-score normalization (ZN)	91	305.7	**0.2383**	0.3437
normalized weekday difference (NWD)	91	308.5	**0.2631**	0.3730
weekday ratio (WR)	91	302.4	**0.2384**	0.3443

Table 5.5: The best mean error achieved by convolutional models on validation data for each preprocessing method.

5.5.4 Hybrid Networks

Convolutional networks by themselves did not reach the performance of recurrent networks. Nevertheless, we believe that feature extraction using convolutional neural networks could be effective when combined with some form of recurrence that keeps track of the long term relations. Therefore, in addition to traditional approaches, we suggested a hybrid model combining convolutional layers with recurrent cells.

We tried several possible options of combining the convolutional and recurrent approaches including two separate models combined in the last dense layer or recurrently processing features detected by the convolutional layers. The method that seemed the most promising was a network consisting of a layer of GRU memory cells and convolutional layers processing the whole sequence output from the recurrent layer. We will refer to this network as GRU-convolutional neural network (GRU-CNN). This architecture combines the advantages of ability to capture long-term relations in input as well as convolutional feature extraction.

We experimented with various architectures of this type and various network sizes. The best results were achieved with a single GRU layer with 50 neurons. More neurons caused networks to overfit independently of the rest of the network. Upon this recurrent layer we tried to build various convolutional architectures. However, once again the best results were achieved with a single convolutional layer with 16 kernels of size 7, followed by a max-pooling layer with regions of size 2 and a fully-connected layer.

The best results by each preprocessing method are summarized in Table 5.6.

As can be seen, with this architecture we were able to slightly outperform the pure GRU network which has shown the best results so far. Figure 5.7 shows example time series where GRU-CNN significantly outperformed the best GRU network.

	look back	RMSE	**SMAPE**	MIBRE
min-max normalization (MMN)	28	326.9	**0.2478**	0.3552
z-score normalization (ZN)	28	323.3	**0.2434**	0.3498
normalized weekday difference (NWD)	28	328.0	**0.2515**	0.3585
weekday ratio (WR)	28	315.6	**0.2416**	0.3480
min-max normalization (MMN)	91	299.8	**0.2355**	0.3403
z-score normalization (ZN)	91	276.9	**0.2281**	0.3326
normalized weekday difference (NWD)	91	306.4	**0.2428**	0.3486
weekday ratio (WR)	91	272.4	**0.2337**	0.3391

Table 5.6

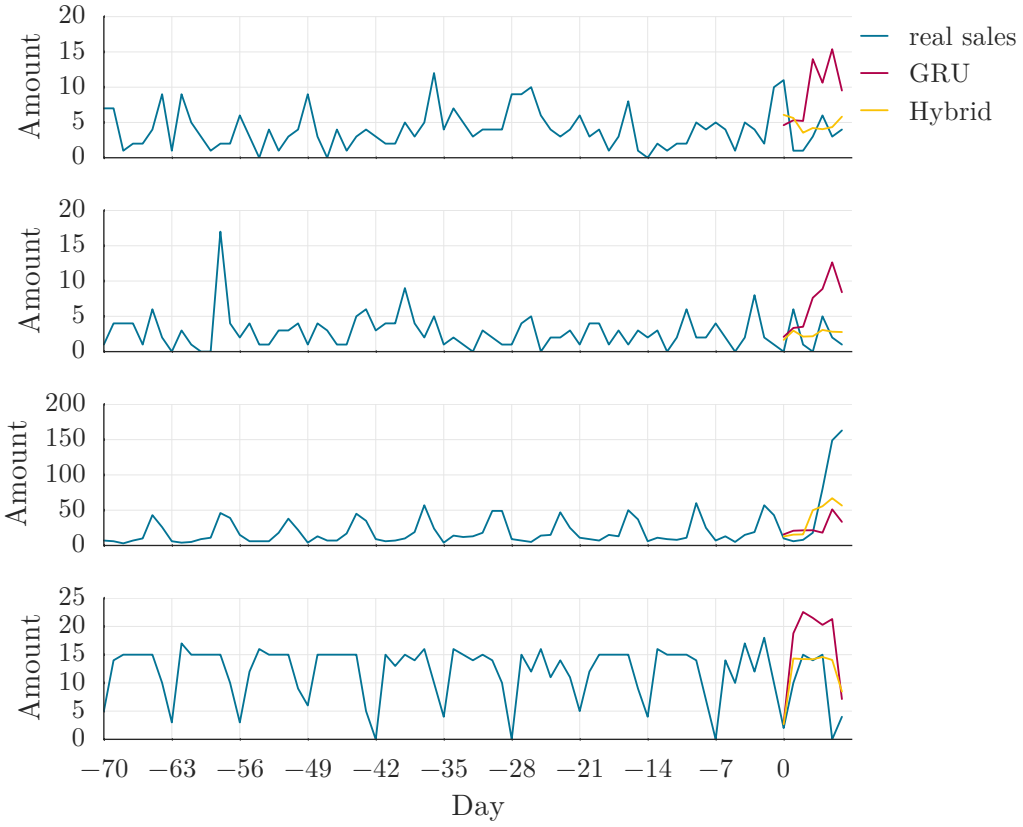


Figure 5.7: Example time series where the best GRU-CNN significantly outperforms the best GRU model, which has been the best model so far.

5.5.5 Analysis of Small Dataset Experiments

The errors of the best models (chosen by SMAPE value) of each type computed on the validation set can be seen in Table 5.7. These error values cannot be interpreted as model accuracy, since the models were tuned and chosen based on these errors. However, this processes gave us the best architecture for each type of model. Parameters of these architectures are summarized in Table 5.8.

Training time of different models strongly depends on the specific architecture. It took a few minutes to train a FNN, dozens of minutes to train CNN and a few hours to train recurrent and hybrid networks on the small dataset. Despite these differences, after the models were trained, they were all able to predict the 400 validation series in a few seconds.

	RMSE	**SMAPE**	MIBRE
GRU-CNN	276.9	**0.2281**	0.3326
GRU	235.5	**0.2293**	0.3332
FNN	264.9	**0.2303**	0.3336
CNN	297.2	**0.2330**	0.3378
LSTM	264.5	**0.2340**	0.3395
SRNN	297.4	**0.2342**	0.3397

Table 5.7: The best mean results achieved on validation data for each method. The best results were chosen and ordered by SMAPE value. Note that the network parameters were tuned using validation data, so the results do not represent model performance in general.

	FNN	SRNN	LSTM	GRU	CNN	GRU-CNN
preprocessor	ZN	ZN	WR	WR	ZN	ZN
look back	91	91	91	91	91	91
layers	1	1	1	1	2	3
hidden act.	sigmoid	tanh	tanh	tanh	ReLU	tanh, ReLU
output act.	DHT	DHT	softplus	softplus	DHT	DHT
neurons	100	100	100	100	16×7	$50 + 16 \times 7$
algorithm	RMSProp	RMSProp	SGD	RMSProp	SGD	SGD
learning rate	0.0005	0.0005	0.1	0.001	0.01	0.05

Table 5.8: Description of the best model of each type of network. Layers refer to the number of hidden layers, hidden act. and output act. denote activations in hidden and output layers respectively. Neurons denote the number of hidden neurons, algorithm is the learning algorithm. Note that both LSTM and GRU neurons have additional inner cells with their own activation function. In both cases a hard sigmoid was used - a linear approximation of sigmoidal activation.

As can be seen in Table 5.7 the best errors produced by each model type are very similar. These values are mean errors across the whole dataset, but we also analyzed the errors more deeply. We computed the SMAPE value for each method and time series from validation set and studied the correlation coefficients between errors produced by each method. The results of this analysis can be seen in heat map dendrograms in Figures 5.8 and 5.9. Most of the errors are significantly correlated and it might be noticed that the models in dendrograms are mostly grouped by the same method of preprocessing. This suggests that preprocessing method is the main factor that affects which types of series would be forecasted with lower accuracy.

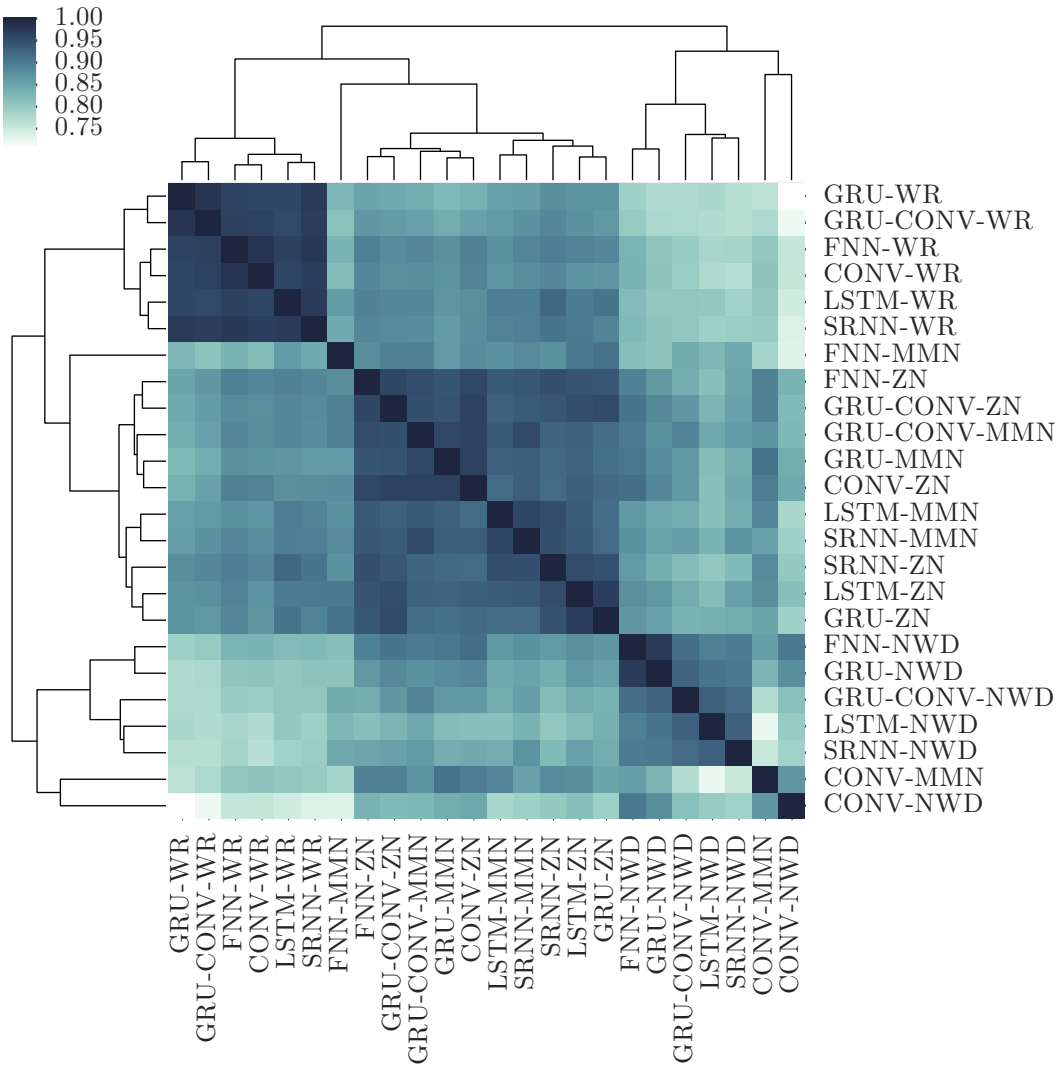


Figure 5.8: Heat map dendrogram showing correlations between errors produced by different models with look back ok 28 days. The errors were computed on time series from validation set. Each model description is in the format X-Y where X is a type of model and Y is preprocessing method.

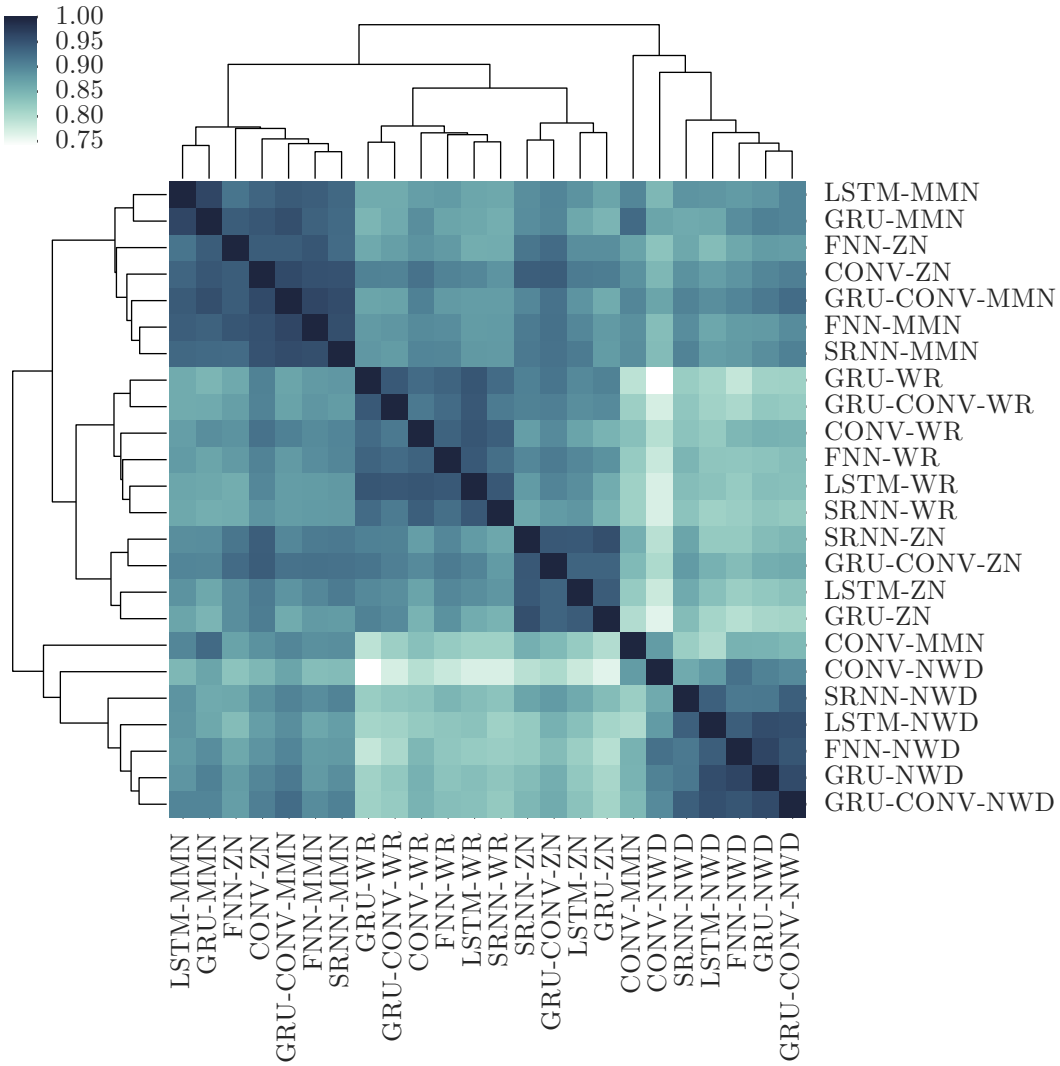
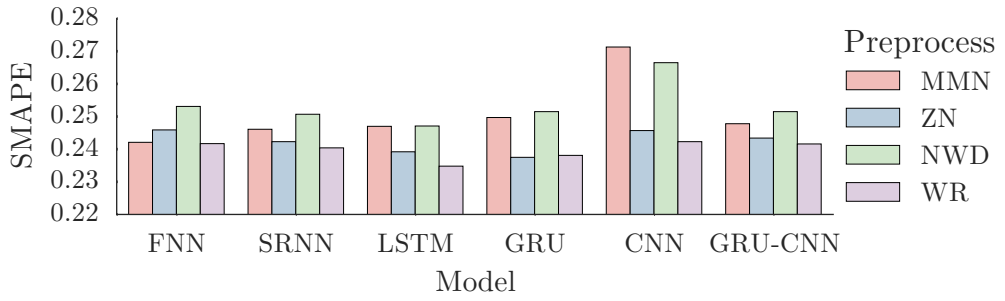
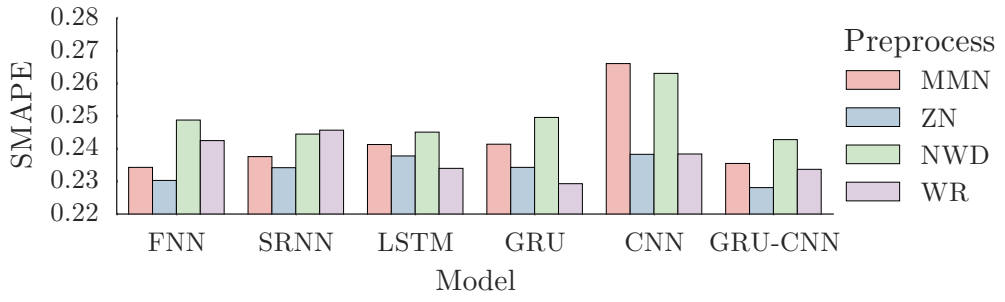


Figure 5.9: Heat map dendrogram showing correlations between errors produced by different models with look back of 91 days. The errors were computed on time series from validation set. Each model description is in the format X-Y where X is a type of model and Y is preprocessing method.

Among the preprocessing methods, ZN and WR usually produced the lowest error. A comparison of SMAPE values for different preprocessing methods and models can be seen in Figure 5.10.



(a) 28 days long look back



(b) 91 days long look back

Figure 5.10: SMAPE by preprocessing method and model type.

5.6 Large Dataset Experiments

Studies [73, 80] from the last few years show that deeper neural networks lead to even better results. Deeper models consist of significantly larger number of parameters that must be trained. To be able to handle such a large architecture and avoid network overfitting, more data must be introduced.

We used our large Multistore dataset to train deeper architectures. The larger the architecture and the training set are, the longer it takes to train the network. Training most of these models took many hours. We have reached more than 12 hours with some of our models and were not able to increase the training speed anymore, because of limited memory of our GPU. Thus we were not able to conduct such a large-scale experiment as we did with the small dataset. We only experimented with models that produced the best results on the small dataset. We tuned their parameters using the validation set and then compared their performance to the models trained using the small dataset and to our benchmark models. The performance was compared using the test set extracted from the large dataset. This data was never seen by any of the models and was never used before to tune any parameters or compare the models.

For experiments on the large dataset we chose FNN, GRU network and GRU-CNN because these models produced the best performance on the small dataset. First of all, we used the same network architectures as were used in small dataset experiments. The only difference was in the amount of training data. Then we experimented with extending the models by adding more neurons and more layers.

According to the expectations, the FNN did not seem to improve with multiple hidden layers. However we were able to lower the error on validation set by adding more hidden neurons to a single hidden layer. The best results were achieved with 200 hidden neurons. In comparison, adding one hidden layer to the GRU network led to improvement in accuracy. The lowest error value on validation set was achieved with architecture consisting of a hidden layer with 100 GRU cells and second hidden layer with 50 GRU cells. The best GRU-CNN architecture that we found consisted of a single GRU layer with 100 GRU cells and a single convolutional layer with 32 kernels of size 7 and ReLU activation function. Upon this layer a max-pooling was done before fully-connected layer. A summarization of the best architecture of each type of model can be found in Table 5.9.

	FNN	GRU	GRU-CNN
preprocessor	ZN	WR	ZN
look back	91	91	91
layers	1	2	3
hidden act.	sigmoid	tanh	tanh, ReLU
output act.	DHT	softplus	DHT
neurons	200	$100 + 50$	$100 + (32 \times 7)$
algorithm	RMSProp	RMSProp	SGD
learning rate	0.01	0.001	0.01

Table 5.9: Description of the best model of each type of network. Layers refer to the number of hidden layers, hidden act. and output act. denote activations in hidden and output layers respectively. Neurons denote the number of hidden neurons, algorithm is the learning algorithm. Note that GRU neurons have additional inner cells with their own activation function. In both cases a hard sigmoid was used - a linear approximation of sigmoidal activation.

5.6.1 Analysis of Large Dataset Experiments

The average RMSE computed on the test set is lower than average RMSE values computed on the validation set extracted from small dataset. The reason is that RMSE strongly depends on the scale of the series. Since time series in large dataset contained lower values, RMSE became lower. Contrarily, value of

SMAPE is significantly higher when computed on the test set. We analyzed this behavior and found that several time series in the test set ended with zero sales (see Figure 5.11) which did not happen in the small dataset. This caused SMAPE for the product to be 1 which is the highest possible value in case of non-negative values as the sales are. In practice, this is a very specific situation usually caused by stock-outs (which could be prevented using good forecasts) or by intentionally selling-out the product which is known beforehand so the forecast would not lead to incorrect decisions. Due to the problems with zero sales we introduce a modified version of SMAPE error denoted as SMAPE-NONZERO. This error value does not include days where actual amount sold is zero. Based on this value, we will compare the overall model performance.

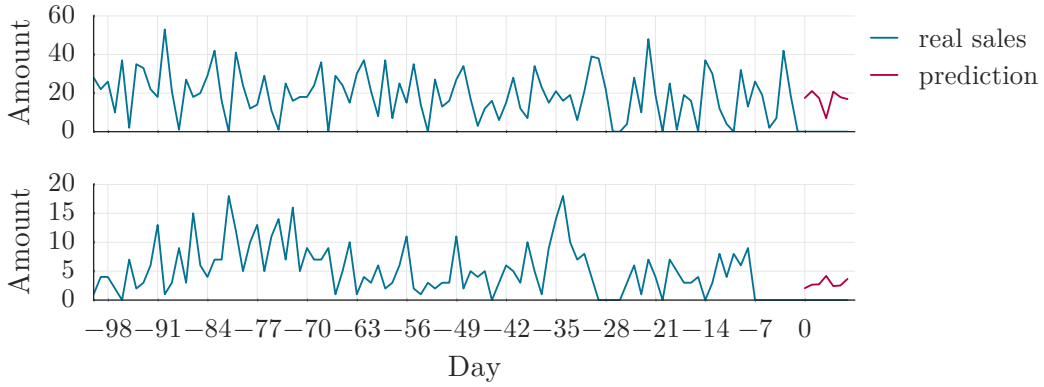


Figure 5.11: Example time series from the test set that end with zero sales. The models do not expect this behavior.

As can be seen in Table 5.10 all the methods show very similar accuracy. The weekday average benchmark showed the poorest performance. It would probably be more accurate if the average only considered last few values or a weighted average was used. We however proposed this benchmark to check whether our models with preprocessing based on weekday means did not learn to naively predict weekday mean. Since all the models easily outperformed this benchmark, one can see that it did not happen. Random forests method also did not outperform our models. We trained the estimator with 100 trees using 120,000 training samples. Adding more training samples or more trees did not lead to better results. The accuracy of exponential smoothing is comparable to most of our results.

Performance of the models trained on small dataset is quite surprising. Even though GRU-CNN showed the best results on the validation set, its accuracy computed on the test set was worse than accuracy of LSTM or GRU network. The most likely reason is that results of GRU-CNN on validation set were biased, since the network parameters were tuned using this dataset. The network performed

very well on this data, but it was not able to generalize.

	Dataset	RMSE	SMAPE	MIBRE	**SMAPE-NONZERO**
Exponential Smoothing		44.2	0.3706	0.4149	**0.2377**
Random Forests		95.3	0.3220	0.4219	**0.2445**
Weekday Average		60.2	0.3268	0.4272	**0.2498**
GRU	small	42.7	0.3144	0.4122	**0.2359**
LSTM	small	43.5	0.3153	0.4134	**0.2369**
FNN	small	44.3	0.3230	0.4221	**0.2414**
GRU-CNN	small	41.6	0.3210	0.4176	**0.2419**
SRNN	small	43.3	0.3226	0.4221	**0.2422**
GRU	large	42.4	0.3105	0.4078	**0.2316**
FNN	large	40.1	0.3153	0.4128	**0.2353**
GRU-CNN	large	44.3	0.3185	0.4170	**0.2386**

Table 5.10: Model performance evaluated on the test set. Column Dataset indicates which dataset the model has been trained on.

Based on the SMAPE-NONZERO values in the above table, we have made following observations:

1. The GRU network trained on large dataset outperforms other neural network models, including FNN, trained on large dataset.

2. The models trained on large dataset show performance superior to the corresponding models trained on small dataset.

3. The GRU network trained on large dataset seems to outperform all our benchmarks.

To determine whether these results are statistically significant, we trained the GRU network, GRU-CNN and FNN for three times with different weight initializations both on large and small dataset. Since training of GRU network and GRU-CNN takes many hours, we were not able to manage a larger examination. Each time we tested our models on the same test set, however the size of our test set (2000 samples) is quite large to provide a reliable estimate of accuracy. The resulting means and standard deviations of SMAPE-NONZERO can be found in Table 5.11.

	Dataset	Mean	Standard deviation
GRU	small	0.23698	0.00087
FNN	small	0.24154	0.00014
GRU-CNN	small	0.24316	0.00092
GRU	large	0.23198	0.00027
FNN	large	0.23602	0.00067
GRU-CNN	large	0.23846	0.00094

Table 5.11: Mean and standard deviation of SMAPE-NONZERO based on three runs with different random weight initializations. Column Dataset indicates which dataset the model has been trained on.

We now determine whether our observations are statistically significant according to the statistics in Table 5.11. We will use 98% as our desired confidence.

To confirm the first observation, we had to compare error distribution of GRU network to error distributions of CNN and GRU-CNN (all trained on the large dataset) and find the confidence interval for the difference between their means. We used Welch's t-test to find the confidence interval. Based on our sample, the error of GRU network is lower than the errors of FNN and GRU-CNN by somewhere between

- 0.00143 and 0.00665 in case of FNN
- 0.00248 and 0.01048 in case of GRU-CNN

with 98% confidence.

To confirm the second observation, we needed to compare error distribution of each GRU, FNN and GRU-CNN models trained on large dataset to their counterpart trained on the small dataset. Again, we examined the 98% confidence intervals for the difference between their means. According to our data, the error differences between model trained on the small dataset and model trained on the large dataset were with 98% confidence in following ranges:

- (0.00138, 0.00863) for the GRU network
- (0.00246, 0.00857) for the FNN
- (0.00123, 0.00819) for the GRU-CNN

Therefore, according to our results, the models trained on the large dataset perform better than models trained on the small dataset with 98% confidence.

To verify the third observation, we have to test the errors of GRU network agains the errors produced by the benchmark models. The exponential smoothing and weekday average models are deterministic. We therefore need to check,

whether it is proper to claim, that mean error of GRU model is lower than 0.2377 and 0.2498. We state the null hypothesis that the mean error is equal to 0.2377, which is the error produced by exponential smoothing, against an alternative hypothesis that the error is lower. With t-statistics of -29.96 and $\frac{p\text{-value}}{2} = 0.00056 < 0.02$ we reject the null hypothesis in favor of the alternative hypothesis. Our last benchmark, random forests algorithm is not deterministic since it is only trained using a subset of data. We therefore run the algorithm for three times, using different random subset of data each time, to compute the mean and standard deviation of its SMAPE-NONZERO value. The mean error of random forests was 0.24432 with standard deviation 0.00015. Based on our results, this error is higher than the error of GRU network by somewhere between 0.01138 and 0.01330 with 98% confidence.

5.7 Rossmann Dataset Experiments

The Rossmann data is quite different from Multistore data and it provides an opportunity to inspect versatility of our models. The experiments conducted using Rossmann dataset are also attached on the CD and serve as an illustration of our experiments.

Here we introduce another error function called RMSPE which was used in the Kaggle competition. This error function is given by

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\frac{y_i - \hat{y}_i}{y_i}\right)^2}$$

where any day and store with zero sales is ignored in scoring. Ignoring days with zero sales is again reasonable because it generally means that the store is closed which is known in advance.

The best score that was achieved in the competition was 0.10021. Our results are not completely comparable to this result for two reasons. First of all this score was computed on a different data which is not available. Second, we do not include any other indicators in our prediction such as store type, closest competition, holidays, promo actions etc.

We trained the best architectures of each model type using Rossmann dataset for demonstration purposes, however in this section we will focus on the models that already showed the best performance on the Multistore dataset. We proposed a triple-evaluation of these models:

1. Use model trained on the full Multistore dataset. This model would only be trained using samples from Multistore dataset and it would never see any samples from the Rossmann dataset before evaluation.

2. Use model trained on the Rossmann dataset. This model would have randomly initialized parameters and then would be trained on samples from Rossmann dataset.

3. Use model pre-trained on the full Multistore dataset and then trained using the Rossmann dataset. This model would be initialized with parameters learnt using Multistore dataset and then will be trained more using samples from Rossmann dataset.

The error of the first option tells us whether our models are universal enough to predict very different sales time series. The second option shows us whether the models are suitable for aggregated data. The last option examines the power of supervised pre-training.

5.7.1 Analysis of Rossmann Dataset Experiments

The Rossmann dataset experiments were made over each type of model. Here we will only analyze the results of the best models identified using the Multistore dataset. The RMSPE errors achieved by these models are summarized in Table 5.12. Values of SMAPE, MIBRE and SMAPE-NONZERO were substantially smaller than the same errors on the Multistore dataset. This corresponds to the fact that aggregated series contain less noise. On the contrary, RMSE error was much higher since it is on the same scale as data and aggregated data involves higher values.

	Variant	Model	RMSPE
FNN	1.	FNN_1x200_ZN_91_pretrained_only	0.23448
FNN	2.	FNN_1x200_ZN_91	0.13549
FNN	3.	FNN_1x200_ZN_91_pretrained	0.12640
GRU	1.	GRU_100_50_WR_91_pretrained_only	0.24039
GRU	2.	GRU_100_50_WR_91	**0.12629**
GRU	3.	GRU_100_50_WR_91_pretrained	0.13730
GRU-CNN	1.	GRU_CNN_WR_91_pretrained_only	0.22671
GRU-CNN	2.	GRU_CNN_WR_91	0.13761
GRU-CNN	3.	GRU_CNN_WR_91_pretrained	0.13723

Table 5.12: Model performance evaluated on the test set. Column Variant indicates the type of evaluation - 1. model trained on the Multistore dataset, 2. model trained on the Rossmann dataset, 3. model pre-trained on the Multistore dataset and then trained using the Rossmann dataset. Column Model indicates model name used in the experiments, it can be found by this name on the attached CD.

As Table 5.12 shows, GRU network produced the lowest error on this dataset too. Models that were only trained using the Multistore dataset performed significantly worse. In spite of that, models with weights pre-trained using Multistore dataset and then trained using the Rossmann dataset resulted in the lowest error for two out of three models, namely FNN and GRU-CNN. This result indicates that such a pre-training might lead to increase in accuracy. To confirm this hypothesis it would be necessary to repeat the experiments for more times.

6. Extensions

Based on the results of our experiments we propose several extensions that might improve the performance of our deep neural network models and lead to more accurate forecasts. In this chapter we describe these extensions, present the results of the extensions that we tried and suggest some of them for future work.

6.1 Fine-tuning

The first extension that we tried is fine-tuning. Just before predicting the future values of each time series, we would train the network for a while using the past values of the same series. This allows to introduce more historical data in our predictions without increasing the size of look back, which significantly increases the network architecture and consequently the time that is necessary for training.

A disadvantage of this extension is that it significantly prolongs the time spent on predicting. We experimented with multiple training epochs of fine-tuning and tried to fine-tune two of our models, namely GRU network and GRU-CNN. In all the experiments, fine-tuning resulted in larger error values. We did not experiment with various learning rates a lot, but according to what we tried, this extension is not very useful.

6.2 Ensembling

Ensemble methods use multiple learning algorithms, combined by averaging or voting, to obtain better predictive performance. A diversification is quite important when combining models, which means that ensemble of very similar techniques is likely to perform worse than more diverse ensemble. According to our analysis in Section 5.5.5 various neural network models produced highly correlated errors. The main differences were achieved by using different data preprocessing. Consequently, it seems appropriate to combine models that use different preprocessing or combine them with completely different forecasting methods.

We selected two of our models trained on large Multistore dataset, namely the best FNN network which uses ZN preprocessor and the best GRU network which uses WR preprocessor. In addition, we decided to experiment with predictions of random forests and exponential smoothing as well. We have examined all the combinations of these models. Even though we only did a simple averaging of these models, we identified three combinations that led to better results than each of the individual models from the combination. These combinations, together

with their error value can be found in Table 6.2. Error of the individual models is shown in Table 6.1.

	SMAPE-NONZERO
Exponential Smoothing	0.2377
Random Forests	0.2445
GRU	0.2316
FNN	0.2353

Table 6.1: Error of the individual models computed on the test set.

	SMAPE-NONZERO
GRU + FNN	0.2307
GRU + FNN + Exponential Smoothing	0.2306
FNN + Exponential Smoothing + Random Forests	0.2349

Table 6.2: Combinations of models that produced lower error value than each individual model from the combination. The error is computed on the test set.

6.3 Future Work

There are many possible extensions that can be done to our models and it was not possible to try each one of them. In this section we suggest several other extensions that might be done in future work.

Since the most important factor that distinguished our models was preprocessing, it makes sense to introduce more methods of preprocessing and try to combine them with each model. One of the possible preprocessing is STL decomposition that decomposes the series into seasonal, trend and error component. It might be easier for the models to forecast each part separately. Another option is to forecast seasonal and trend component with more simple forecasting method and only try to find patterns and predict the error component using deep neural networks.

As might be seen in the results of our experiments on the small dataset (see Section 5.5), the experiments with look back of 91 days produced significantly better results than experiments with look back of 28 days. This suggests that increasing the look back size might improve the prediction accuracy. This can be, however, only done to some extent, since increasing the network input size increases the number of parameters that need to be trained.

The analysis of large Multistore dataset confirms the correlation between price and amount that was sold. Based on this fact, it might help to provide the

network with information about price. The information about expected price change should help the network to predict for example increased sale in case of promo action. Moreover, the retailers usually know about the price change beforehand, so it is perfectly valid to assume that this information is available.

To improve the forecast accuracy it might be also helpful to indicate special days such as holidays or days with active promo actions. A network with inputs extended in such a way should be able to find patterns how these indicators affect the sales in the corresponding day and other surrounding or correlated days.

In this thesis we analyzed deep neural networks for daily sales forecasting. Another option is forecasting aggregated weekly or monthly sales. These series are very different from daily sales time series. They comprise of different features and contain significantly less noise. Aggregated sales are also easier to predict using standard statistical methods than daily sales.

Conclusion

In this thesis we showed that forecasting sales using deep neural networks might help to increase the accuracy of short-term forecasts when done properly. Even though the improvement in accuracy is quite small, it may help to significantly reduce the costs in retail business.

We collected and processed a lot of sales data which we then used to experiment with various deep neural network models. Our goal was to train a universal model, that would be able to predict sales of a product without being trained on its history.

Using a smaller subset of the data we examined over 180 different architectures covering traditional feedforward neural networks, simple recurrent networks, LSTM networks, GRU networks, convolutional networks and novel architectures combining recurrent elements with convolutional layers. We studied their performance with various learning settings and preprocessing methods, including unconventional preprocessing techniques such as forecasting the ratio to the mean sales of corresponding day of week. This technique, together with standard z-score normalization usually produced the best results.

The best architectures that we found using the smaller subset of data were then extended and trained using the whole dataset. We have shown that adding more data, which also enables to expand the architectures without problems with overfitting, helps to increase accuracy of forecast. The experiments on small dataset suggested that the hybrid model that we proposed makes predictions with highest accuracy. The final testing however showed, that the lowest error was achieved by GRU network trained on the full dataset in combination with preprocessing that computes the ratio to the mean sales of corresponding day of week. This network outperformed all the other neural network models and also all our benchmarks, including a very sophisticated implementation of exponential smoothing that involves 15 different exponential smoothing techniques, which is a great achievement.

We have also experimented with using trained models to forecast different kind of sales time series to test its ability to generalize. This data came from different kind of stores, involved different types of products and the sales were aggregated over the whole store. We found that the models that were trained on very different data did not perform very well. However, such a pre-trained model seems to be a good starting point for further training on completely different sales data.

As a reaction to the results of our experiments, we suggested several extensions

and some of them we have already tried. We discovered that the accuracy of models can be further improved by combining models. We showed that it is reasonable to combine models that use different data preprocessing, since it is the main factor that affects the diversity of forecast errors.

Training a deep neural network may take many hours, however once a network is trained, the actual forecasting is very fast. We believe that a trained model might be used for quite a time, since these models are not limited to a specific time range. The accuracy might be however increased by introducing more recent data from time to time.

Based on our experiments, we suggest that recurrent neural networks, specifically GRU network, are the best architecture for this type of problem. We advise to use this network with preprocessing that computes the ratio to the mean sales of corresponding day of the week. According to our analysis, this setup is superior to traditional forecasting techniques and we believe that using this technique may help to remarkably reduce the expenses related to supply chain. The best results might be however achieved by ensembling this technique with other approaches such as feedforward neural networks.

Forecasting sales with deep neural networks has a weakness of being a black box algorithm. This is a very discouraging factor in business and it may prevent them from being used in practice since retailers often prefer simple forecasts instead of more accurate techniques. Despite this property, forecasting daily sales of frequently sold products using deep neural networks is a perfectly valid alternative to traditional forecasting techniques. The forecasting is fast and deep neural networks are able to exploit features that are ignored by traditional methods and consequently produce forecasts with higher accuracy.

Bibliography

[1] Daniel Corsten and Thomas Gruen. Desperately seeking shelf availability: an examination of the extent, the causes, and the efforts to address retail out-of-stocks. *International Journal of Retail & Distribution Management*, 31(12):605–617, 2003.

[2] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[3] Javed Khan, Jun S Wei, Markus Ringner, Lao H Saal, Marc Ladanyi, Frank Westermann, Frank Berthold, Manfred Schwab, Cristina R Antonescu, Carsten Peterson, et al. Classification and diagnostic prediction of cancers using gene expression profiling and artificial neural networks. *Nature medicine*, 7(6):673–679, 2001.

[4] Barry Lennox, Gary A Montague, Andy M Frith, Chris Gent, and Vic Bevan. Industrial application of neural networks—an investigation. *Journal of Process Control*, 11(5):497–507, 2001.

[5] José Ernesto Rayas-Sánchez. Em-based optimization of microwave circuits using artificial neural networks: The state-of-the-art. *IEEE transactions on microwave theory and techniques*, 52(1):420–435, 2004.

[6] Christian Spreckelsen, Hans-Jörg Mettenheim, and Michael H Breitner. Real-time pricing and hedging of options on currency futures with artificial neural networks. *Journal of Forecasting*, 33(6):419–432, 2014.

[7] Stephen Judd. Learning in networks is hard. In *Proceedings of the First International Conference on Neural Networks*, pages 685–692. IEEE San Diego, CA, 1987.

[8] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[9] M. Minsky. *Perceptrons*. M.I.T. Press, 1969.

[10] Simon Haykin and Neural Network. A comprehensive foundation. *Neural Networks*, 2(2004), 2004.

[11] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[12] Teuvo Kohonen. The self-organizing map. *Proceedings of the IEEE*, 78(9):1464–1480, 1990.

[13] Seppo Linnainmaa. The representation of the cumulative rounding error of an algorithm as a taylor expansion of the local rounding errors. *Master's Thesis (in Finnish), Univ. Helsinki*, pages 6–7, 1970.

[14] DE Rumelhart GE Hinton RJ Williams and GE Hinton. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[15] Tijmen Tieleman and Geoffrey Hinton. Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, 4:2, 2012.

[16] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, 12:2121–2159, 2011.

[17] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[18] David H Hubel and Torsten N Wiesel. Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106–154, 1962.

[19] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. Master's thesis, 1991.

[20] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[21] Yann LeCun, Corinna Cortes, and Christopher JC Burges. The mnist database of handwritten digits, 1998.

[22] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. 2006.

[23] Santiago Fernández, Alex Graves, and Jürgen Schmidhuber. Sequence labelling in structured domains with hierarchical recurrent neural networks. pages 774–779, 2007.

[24] Dan Cireşan, Ueli Meier, Jonathan Masci, and Jürgen Schmidhuber. A committee of neural networks for traffic sign classification. pages 1918–1921, 2011.

[25] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[26] Quoc V Le. Building high-level features using large scale unsupervised learning. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8595–8598. IEEE, 2013.

[27] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, pages 248–255. IEEE, 2009.

[28] Sebastian Otte, Dirk Krechel, Marcus Liwicki, and Andreas Dengel. Local feature based online mode detection with recurrent neural networks. In *Frontiers in Handwriting Recognition (ICFHR), 2012 International Conference on*, pages 533–537. IEEE, 2012.

[29] Emanuel Indermühle, Volkmar Frinken, Andreas Fischer, and Horst Bunke. Keyword spotting in online handwritten documents containing text and non-text using blstm neural networks. In *Document Analysis and Recognition (ICDAR), 2011 International Conference on*, pages 73–77. IEEE, 2011.

[30] Emanuel Indermuhle, Volkmar Frinken, and Horst Bunke. Mode detection in online handwritten documents using blstm neural networks. In *Frontiers in Handwriting Recognition (ICFHR), 2012 International Conference on*, pages 302–307. IEEE, 2012.

[31] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[32] Javier Gonzalez-Dominguez, Ignacio Lopez-Moreno, Hasim Sak, Joaquin Gonzalez-Rodriguez, and Pedro J Moreno. Automatic language identification using long short-term memory recurrent neural networks. In *INTERSPEECH*, pages 2155–2159, 2014.

[33] Raymond Brueckner and Bjorn Schulter. Social signal classification using deep blstm recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 4823–4827. IEEE, 2014.

[34] Jeffrey L Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[35] Christoph Goller and Andreas Kuchler. Learning task-dependent distributed representations by backpropagation through structure. In *Neural Networks, 1996., IEEE International Conference on*, volume 1, pages 347–352. IEEE, 1996.

[36] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[37] Patrick Doetsch, Michal Kozielski, and Hermann Ney. Fast and robust training of recurrent neural networks for offline handwriting recognition. In *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*, pages 279–284. IEEE, 2014.

[38] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.

[39] Søren Kaae Sønderby and Ole Winther. Protein secondary structure prediction with long short term memory networks. *arXiv preprint arXiv:1412.7828*, 2014.

[40] Erik Marchi, Giacomo Ferroni, Florian Eyben, Leonardo Gabrielli, Stefano Squartini, and Bjorn Schuller. Multi-resolution linear prediction based features for audio onset detection with bidirectional lstm neural networks. pages 2164–2168, 2014.

[41] Felix A Gers, Jürgen Schmidhuber, and Fred Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[42] Felix A Gers and Jürgen Schmidhuber. Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pages 189–194. IEEE, 2000.

[43] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[44] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

[45] Kunihiko Fukushima. Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological cybernetics*, 36(4):193–202, 1980.

[46] Yoon Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[47] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *Advances in neural information processing systems*, pages 2933–2941, 2014.

[48] Lutz Prechelt. Automatic early stopping using cross validation: quantifying the criteria. *Neural Networks*, 11(4):761–767, 1998.

[49] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 4*, pages 950–957. Morgan Kaufmann, 1992.

[50] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[51] Richard Bellman. Adaptive control processes: A guided tour. 1960.

[52] Kevin L Priddy and Paul E Keller. *Artificial neural networks: an introduction*, volume 68. SPIE Press, 2005.

[53] Harold Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of educational psychology*, 24(6):417, 1933.

[54] George EP Box and Gwilym M Jenkins. Time series analysis: Forecasting and control. In *Holden-Day series in time series analysis*. Holden-Day, 1976.

[55] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice*. OTexts, 2014.

[56] Robert G Brown. *Exponential smoothing for predicting demand*, volume 5. 1956.

[57] Charles C Holt. Forecasting seasonals and trends by exponentially weighted moving averages. *International journal of forecasting*, 20(1):5–10, 2004.

[58] Peter R Winters. Forecasting sales by exponentially weighted moving averages. *Management Science*, 6(3):324–342, 1960.

[59] Peter Whitle. *Hypothesis testing in time series analysis*, volume 4. Almqvist & Wiksells, 1951.

[60] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[61] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[62] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[63] Vladimir N Vapnik. The nature of statistical learning theory. 1995.

[64] Chris Tofallis. A better measure of relative prediction accuracy for model selection and model estimation. *Journal of the Operational Research Society*, 66(8):1352–1362, 2014.

[65] Frank M Thiesing and Oliver Vornberger. Sales forecasting using neural networks. 4:2125–2128, 1997.

[66] Ilan Alon, Min Qi, and Robert J Sadowski. Forecasting aggregate retail sales:: a comparison of artificial neural networks and traditional methods. *Journal of Retailing and Consumer Services*, 8(3):147–156, 2001.

[67] Moritz Muller-Navarra, Stefan Lessmann, and Stefan Voß. Sales forecasting with partial recurrent neural networks: Empirical insights and benchmarking results. pages 1108–1116, 2015.

[68] Robert D Klassen and Benito E Flores. Forecasting practices of canadian firms: Survey results and comparisons. *International Journal of Production Economics*, 70(2):163–174, 2001.

[69] Teresa M McCarthy, Donna F Davis, Susan L Golicic, and John T Mentzer. The evolution of sales forecasting management: a 20-year longitudinal study of forecasting practices. *Journal of Forecasting*, 25(5):303, 2006.

[70] Sarah Gelper, Roland Fried, and Christophe Croux. Robust forecasting with exponential and holt–winters smoothing. *Journal of forecasting*, 29(3):285–300, 2010.

[71] Leona S Aiken, Stephen G West, and Raymond R Reno. *Multiple regression: Testing and interpreting interactions*. Sage, 1991.

[72] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European Conference on Computer Vision*, pages 818–833. Springer, 2014.

[73] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *arXiv preprint arXiv:1512.03385*, 2015.

[74] James Martens. Deep learning via hessian-free optimization. In *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pages 735–742, 2010.

[75] Ilya Sutskever, James Martens, and Geoffrey E Hinton. Generating text with recurrent neural networks. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pages 1017–1024, 2011.

[76] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3156–3164, 2015.

[77] Qing Cao, Bradley T Ewing, and Mark A Thompson. Forecasting wind speed with recurrent neural networks. *European Journal of Operational Research*, 221(1):148–154, 2012.

[78] Fabian Pedregosa, Gael Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[79] Wojciech Zaremba. An empirical exploration of recurrent network architectures. 2015.

[80] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

Acronyms

ANN artificial neural network. 5, 27, 34

ARMA autoregressive moving average. 26

CNN convolutional neural network. 16, 17, 55, 63, 66, 73

DHT doubled hyperbolic tangent. 57, 62, 66, 70

FNN feedforward neural network. 6–8, 12, 17, 18, 34, 55–63, 66, 70, 72, 73, 76–78

GRU gated recurrent unit. 16, 55, 59, 61–66, 70, 72–74, 76–78, 81

LSTM long-short term memory. 11–16, 20, 35, 55, 59, 61–63, 66, 72

MMN min-max normalization. 52, 53, 57–62, 64, 65

NWD normalized weekday difference. 53, 55, 57, 60–62, 64, 65

ReLU rectified linear unit. 20, 66, 70

SKU stock keeping unit. 37–39, 41

SRNN Elman's simple recurrent network. 12, 55

SVM support vector machine. 29

WR weekday ratio. 53, 55, 57–62, 64–66, 68, 70, 77

ZN z-score normalization. 52, 57, 60–66, 68, 70, 77

Attachments

The CD attached to this thesis contains an electronic version of this text in file *thesis.pdf*, a source code with experiments on the Rossmann dataset in folder *experiments* and a README file with requirements and detailed description of the *experiments* folder. Since the Multistore dataset is not public, we could not attach the experiments conducted on Multistore dataset.

Folder *experiments* contains:

- file requirements.txt with list of required libraries.
- Python scripts with methods for preprocessing, training and evaluation of models.
- A *raw_data* folder that contains CSV files with Rossmann data downloaded from Kaggle, IPython notebook that transforms this data into train, validation and test set. This folder also contains these datasets in the form of pickled Pandas DataFrames.
- Folder *preprocessed_data*, which includes subfolders with preprocessed data and an IPython notebook that was used for creating those subfolders.
- An *exploratory_data_analysis* folder containing a single file - an IPython notebook with exploratory analysis of Rossmann dataset. Results of this analysis were presented in Section 4.3.3
- Folder *benchmarks* with three subfolders containing our benchmark models. Each subfolder contains an IPython notebook with benchmark model and performance evaluation. The *exponential_smoothing* subfolder also contains an R script that was used for predictions.
- Folders *models_FNN*, *models_RNN*, *models_hybrid* and *models_CNN* with neural network models. Each of them contains several subfolders with models. The main file of each model is IPython notebook *model.ipynb* with model description, definition, evaluation and in most of the cases training process. The training process is missing for models that were only evaluated using pre-trained weights and were not trained on Rossmann dataset. Models that have been pre-trained on Multistore dataset also contain a *weights_pretrained.h5* file with pre-trained weights. Models that have been trained contain a file *weights.h5* with trained weights and other files related to training such as log file.

All the IPython notebooks are also included in HTML format so they can be viewed without any necessary installations. We did not include each model with each architecture and each set of parameters that we experimented with. It is though very easy to modify the settings of a model. Every *model.ipynb* file contains a cell with settings such as batch size or look back and a cell where the model and preprocessor are defined. All the model parameters and variations must be set here, except for the learning rate which might be also defined as a parameter of the *train* method, since its value might change during the training.