TU WIEN Informatics

Comparing machine learning models for predicting the future power output of photovoltaic systems in Austria

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

David Schmalzer
Registration Number 11809617

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner

Vienna, 1st October, 2022

David Schmalzer | Wolfgang Kastner

Erklärung zur Verfassung der Arbeit

David Schmalzer

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Oktober 2022

David Schmalzer

Acknowledgements

First of all I would like to thank my advisor Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner who made it possible for me to write about this subject. He gave me quick and valuable feedback during developing this thesis. Then, I would like to thank my parents, who always supported me during my studies. Last but not least, I want to thank Sofie Aumüller for proof-reading.

Kurzfassung

Da die Erzeugung von Strom mit Photovoltaikanlagen in den letzten 10 Jahren um das 20-fache gestiegen ist, wird es immer wichtiger in Erfahrung zu bringen, wie viel Strom in naher Zukunft erzeugt werden kann. Besonders für Photovoltaikanlagen, die an das Stromnetz angeschlossen sind, ist es von Vorteil zu wissen, wie viel Strom erzeugt wird, da die Leistung schwanken kann und daher im öffentlichen Stromnetz schwer zu handhaben ist. Außerdem wird der Verbrennungsmotor in Autos immer mehr durch den Elektromotor ersetzt, so dass es für Haushalte, die eine Photovoltaikanlage und einen Energiespeicher betreiben, von Interesse ist, zu wissen, wie viel Strom an einem bestimmten Tag zur Verfügung stehen wird. In dieser Arbeit werden verschiedene Ansätze des maschinellen Lernens zur Vorhersage der Leistungsabgabe einer Photovoltaikanlage verglichen. Das Ergebnis dieser Arbeit ist eine Web-Anwendung, die verschiedene Graphen zeigt, die jeweils ein anderes Modell des maschinellen Lernens repräsentieren, um den Unterschied zwischen der vorhergesagten und der tatsächlichen Leistungsabgabe in Echtzeit analysieren zu können.

Abstract

Since the generation of photovoltaic power has increased by a factor of 20 in the last 10 years, the importance of knowing how much power is being generated in the near future increased as well. Especially for photovoltaic systems which are connected to the grid, it is important to know how much power is being generated, since the power output can be volatile and is therefore hard to handle for the public power grid. Furthermore, the combustion engine is getting replaced by the electromotor in cars, thus it could become important for households operating a photovoltaic system and an energy storage, to know how much power will be available on a given day. This thesis focuses on comparing different approaches in machine learning for predicting the power output of a photovoltaic system. The result of this work is a Web application which shows different graphs, each representing a different machine learning model, respectively, to analyze the difference of the predicted power output compared to the real power output in real time.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The generation of photovoltaic power in Austria has increased by an average of 36.9% per year in the timeframe from 2005 to 2019. This resulted in a total power generation of 1.44 TWh in 2018. In total, Austria generated 65 TWh of energy, which means that 2.2% of the energy came from photovoltaic systems [1]. Due to the reduction of power generation through fossil fuels and the increasing demand of renewable energy sources in the past years, the importance of predicting the power output of photovoltaic systems has risen as well. Since photovoltaic systems are volatile concerning the power output due to weather phenomena or the position of the sun, this poses a difficult problem.

1.2 Problem Statement

When reasoning about naive photovoltaic (PV) power generation, one might assume, that the power output of PV systems in August should always be higher than in January, since temperatures and the position of the sun is pretty much always better in August (at least in Austria). However, Figure 1.1 shows a comparison of the first five days of January 2020 and August 2020 in Austria. As illustrated on January 3^{rd} 2020, the peak power output has been higher than on August 3^{rd} 2020. The plot shows data from the ENTSO-E (European Network of Transmission System Operators for Electricity), which contains data for the actual solar power generation in Austria [2]. The data is made available in 15 minute time steps. If we calculate the area under both curves with the trapezoidal rule, we get a total output of 4892 MW/h for the 3^{rd} of January and 3936 MW/h for the 3^{rd} of August. These values need to be seen with caution, since the data is available in 15 minute time steps. However, it clearly shows that the power output has been higher on the 3^{rd} of January.

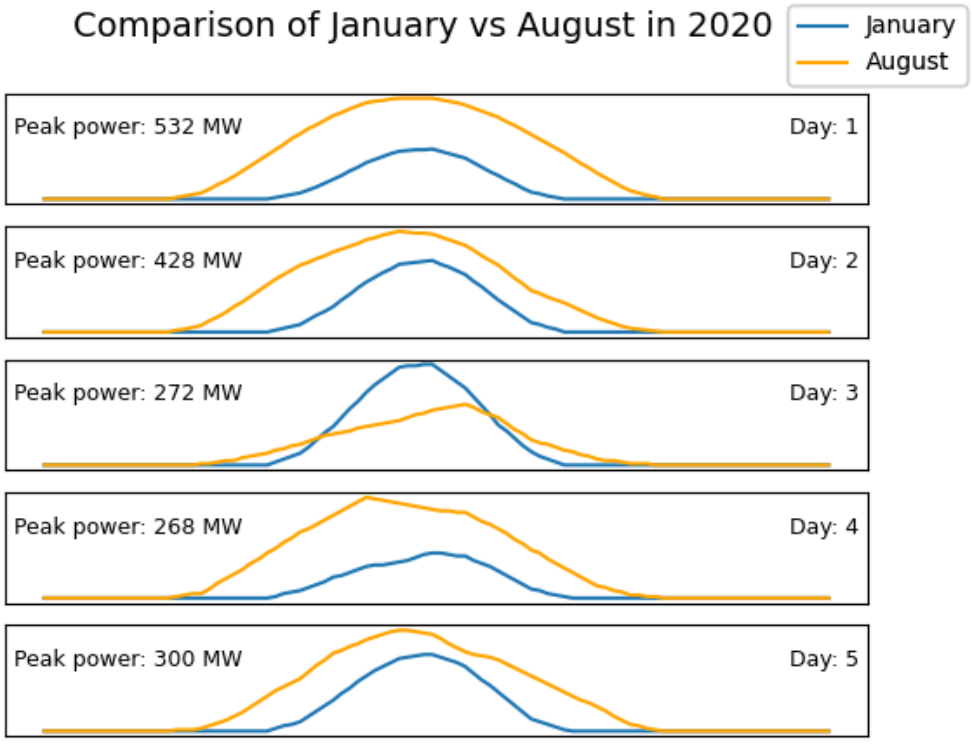

Figure 1.1: PV power output over the first five days of January and August 2020 in Austria

This begs the question about the weather condition on those two days. Therefore, one must look at the historic weather data. At the time of writing, the website wunderground.com provides free historical weather data [3]. Looking at Hörsching, Upper Austria, we can see that it has been rainy and cloudy all day. Doing the same for the weather station in Graz yields that it has been cloudy or light rain all day there. Last but not least going to Währing, Vienna, also shows that it has been raining pretty much all day. The conclusion of this is, that even if the position of the sun and the temperatures are better in August, this does not necessarily mean that the power output is higher than in January. Due to the fact that solar radiation is crucial for the power output of solar cells, it is probably useful to integrate values from solar radiation into the prediction.

Therefore, solar irradiance data from the ARAD project[4] was retrieved from the Central Institute for Meteorology and Geodynamics (ZAMG) to make further tests in this thesis. ARAD stands for Austrian Radiation and is operated by ZAMG. The general aim of ARAD is to establish a high-accuracy long-term monitoring network for solar and terrestrial surface radiation in Austria in order to assess the status as well as the temporal and spatial changes of radiative fluxes at/to the surface [5]. For this thesis, the global

radation measurements from Wien Hohe Warte were investigated. The data comes in 10 minute timesteps and range from 2015-01-01 to 2021-09-01. In order to measure the global radiation, they use the CMP21 and the CM22 Pyranometers from Kipp & Zonen[5].

1.3 Goal

The goal of this thesis is to use and compare different machine learning methods concerning their accuracy regarding the prediction of the photovoltaic power output in all of Austria. The time resolution will be 15 minutes and the prediction horizon will be a one step ahead forecast. On a large-scale environment, this can help to ensure stability of the power grid due to the fluctuating nature of PV energy. For a single household, it can support estimating how much energy can be stored in a potential battery or how much energy will be sold to the public power grid, thus displaying an estimation of earnings to the user on a given day.

In order to show the predicted and actual power output to the user, a Web application will be built. In this application, the different machine learning methods shall be displayed in separate graphs which will show the actual power generation compared to the predicted values of the different machine learning models. This Web application will be running on a Raspberry Pi 4 Model B 8GB. The Web application will continuously be calculating new predictions for all future timestamps available in 15 minute intervals.

1.4 Methodical approach

A recent study has shown that about 60% of studies in the last 15 years used LSTM (Long short-term memory) to predict PV output [6]. LSTM is then followed by RNN (Recurrent neural network) and GRU (Gated Recurrent Unit) which make up for 20% and 13% of studies, respectively. For the prediction of solar irradiance, there is a distribution of 44% using LSTM, 25% using RNN and 19% using GRU. This thesis compares the performance of LSTM and GRU models. Unlike RNN, LSTM alleviates the vanishing gradient problem [7] and thus should provide better performance. Since GRU is a modification of LSTM it is used in this thesis instead of RNN even though RNN is used more often compared to GRU. To reduce statistical outliers every model in every configuration will be trained 5 times and each model gets evaluated. The average of the results for each model will be calculated and then the average is used as final result.

The photovoltaic power generation data comes from [8], because it is mandatory for European Member State data providers and owners to submit fundamental information related to electricity generation, load, transmission and balancing for publication through the ENTSO-E Transparency Platform.

The solar irradiance data was retrieved from the ARAD project[4][5]. ARAD is a longterm project which is designed to measure solar and heat radition in Austria at 5 different

locations. In this thesis, the data from Wien Hohe Warte was used. The timerange from 2019-08-01 00:00:00 to 2020-08-01 00:00:00 was used for training. The models will then be tested with the timerange from 2020-09-01 13:45:00 to 2020-09-05 10:00:00. The performance is evaluated based on different metrics described in Section 4. To automatically train and store the models, a training framework will be built which uses data from [2]. For the evaluation, a Web application will be made which is capable to calculate the metrics defined in Section 4. Based on the results, the best performing models are selected and used in the live view of the Web application.

CHAPTER 2

State of the Art

2.1 Overview

Before diving deeper into the RNN and LSTM models, it is important to understand simpler models. The so-called feedforward neural network [9] should be the simplest model to understand. It consists of an input layer, one or many hidden layers and a single output layer. Each layer can have an arbitrary amount of neurons and is connected to all neurons in the adjacent layer, which can be seen in Figure 2.1.

Now, we can take a look at a single neuron. In Figure 2.2, we can see that there is an

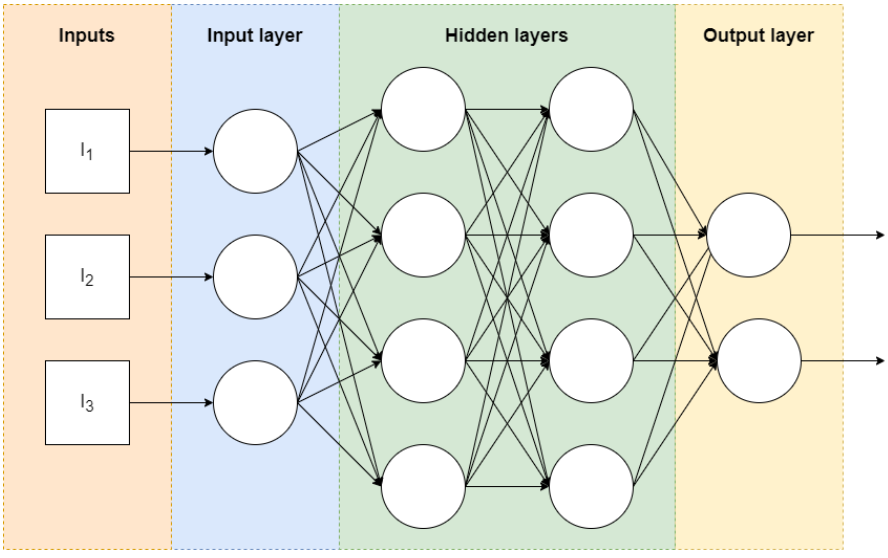

Figure 2.1: Feedforward neural network

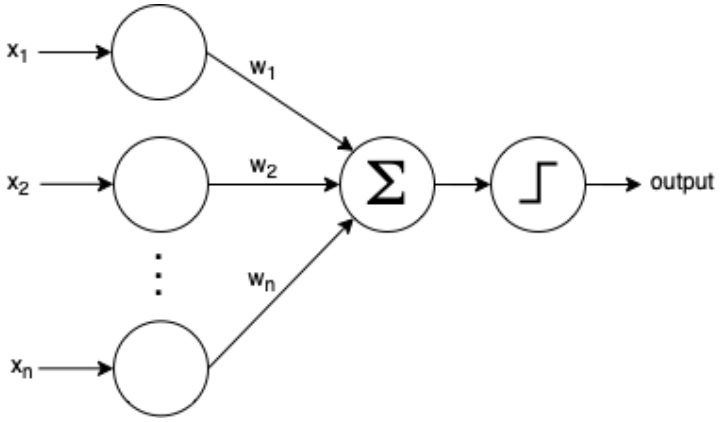

Figure 2.2: Single neuron of a neuronal network [10]

arbitrary amount of inputs labeled x_1 to x_n. Each of the inputs gets multiplied with its corresponding weight which are labeled w_1 to w_n. The resulting values get summed up and are then passed on to the activation function. Depending on which activation function is chosen and the input to the activation function, the neuron triggers and passes on its activation values to the next layer, or the activation function is not triggered and the neuron does not fire. A simple activation function is the sigmoid function, which is defined as follows:

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

If x approaches negative infinity the result will be 0, if x approaches positive infinity the result will be 1. For all other values the result is going to be something in between 0 and 1. This proved to work well for binary classification problems, since the function will trigger and therefore return one if the input is above 1, otherwise it will return 0.

Another more common activation function is the Rectified Linear Unit (ReLU) function. It is defined as follows:

$$f(x) = max(0, x)$$

which is zero for any input smaller or equal to 0 or the input value x for any number greater than 0. Due to the fact that the derivation of this function is always 0 or some constant the vanishing gradient problem is not an issue.

During the training of a machine learning model, each of the weights and the bias of a neuron are adjusted in order to get better results. A bias is a value which gets added to the input of a given neuron, to ensure that at least some neurons in a layer fire in the case that the combination of input signals and weights are not triggering the activation functions of the neurons.

2.1.1 Cost Function

The cost function is the measurement of how far off the predictions of the network is compared to the ground truth [11]. Usually the function is defined as follows:

$$C(w,b) = \frac{1}{2n} \sum_{x} ||y(x) - a||^2$$

Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x [11]. C is representing the quadratic cost function also known as the mean squared error. Note, that we divide by $2n$ just because this will cancel out with the power of 2 when taking the derivative of the cost function for gradient descent. With the cost function, it is defined that the network is performing better, if $C(w,b)$ is approaching 0. The further away the cost function is from 0 the worse the network is performing. Of course, this is dependent on the inputs to the cost function, so we want to find weights and biases which minimize the cost function. In order to minimize the cost function, we are going to use gradient descent.

2.1.2 Gradient Descent

Consider the function $f(x,y) = x^2 + y^2$. As humans, we can easily see that this function has its global minimum at $x = 0$ and $y = 0$. However a computer would have to solve this analytically. Since we are dealing with neural networks which can have thousands or millions of weights and biases as input to the cost function this would result in a huge amount of computation which would take extremely long and is not feasible. To help understanding the following equations, we can imagine a ball rolling down to the lowest point of this function. This point is as mentioned in $x = 0$ and $y = 0$ and illustrated in Figure 2.3.

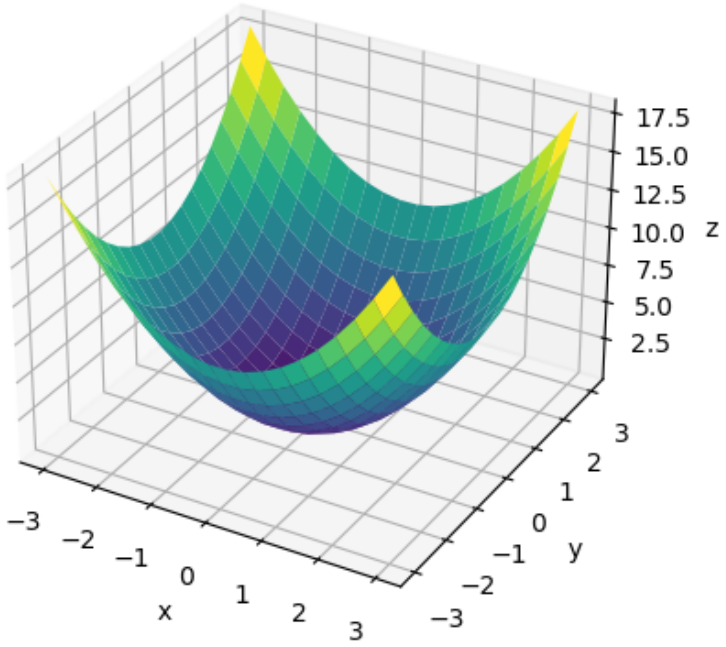

Figure 2.3: Sample plot for gradient decent

Mathematics tells us [11] that moving the ball in the direction v_1 by Δv_1 and v_2 by Δv_2 can be expressed by

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

which shows the changes to the cost function C. Now, the question arises how to choose Δv_1 and Δv_2 to make ΔC negative. In order to accomplish this, we define some notations. Let $\Delta v \equiv (\Delta v_1, \Delta v_2)^T$ where T is the transpose operation and let the gradient of C be the vector of the partial derivatives of v_1 and v_2 which is written as:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

This allows us to rewrite the formula for ΔC as:

$$\Delta C \approx \nabla C \cdot \Delta v$$

Suppose we take

$$\Delta v = -\eta \nabla C$$

η represents the learning rate which is a small positive parameter. Now, we can substitute Δv in the equation above and we get $\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2$. Since $||\nabla C||^2$ is always positive and $-\eta$ always negative it is guaranteed that $\Delta C \leq 0$ as long as we are

in the limits of the approximation $\Delta C \approx \nabla C \cdot \Delta v$. Now, we can take $\Delta v = -\eta \nabla C$ to calculate a value for Δv, then move the ball by this amount:

$$v \to v' = v - \eta \nabla C$$

Doing these steps repeatedly until we get to a minimum is called gradient descent.

2.1.3 Backpropagation

Basically, backpropagation is the synonym for $\frac{\partial C}{\partial w}$ where C denotes the cost function and w any weight or bias in the network. Backpropagation tells us how fast the cost changes if biases or weights are changed [11]. Consider the following neuronal network illustrated in Figure 2.4.

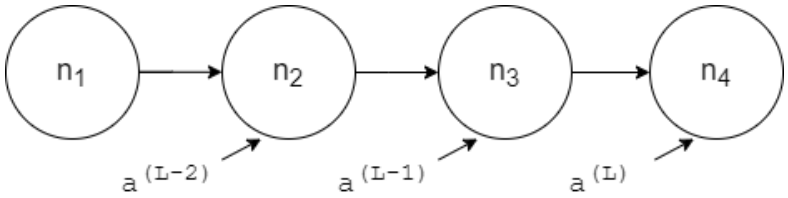

Figure 2.4: Example network with 4 neurons

The $a^{(L)}$ [11][12] labels describe the activation value of the specific neuron. L describes the total amount of layers in the network. Note that the expression $(L - n)$ is not an exponent, it is just a tool for indexing to know which neuron is currently looked at. Furthermore, let y denote the desired value of the last activation. Now let's take a look at the neuron n_4 and let's assume that the actual output of n_4 is 0.75 for a given training example. Also let's assume that the desired output of the network for this training example should be $y = 1$. The cost C_0 for a single training example is defined as $C_0 = (a^{(L)} - y)^2$ which in this case would be $(0.75 - 1)^2$. The activation $a^{(L)}$ is defined as $a^{(L)} = \sigma(w^{(L)} a^{(L-1)} + b^{(L)})$ where $\sigma()$ is some activation function like ReLU or $tanh$. For simplicity, we call the term within the activation function $z^{(L)}$. Now, the question arises how C_0 can be minimized as far as possible. This can be done by adjusting the weights and biases.

First, we will focus on understanding how the adjustment of the weight can affect the cost function. Mathematically expressed we want to know $\frac{\partial C_0}{\partial w^{(L)}}$. We know that a small change to the weight $w^{(L)}$ changes the result of $z^{(L)}$ (1), which changes the result of $a^{(L)}$ (2), which directly changes the result of C_0 (3). Now. we can use this information and create 3 expressions:

(1): $\frac{\partial z^{(L)}}{\partial w^{(L)}}$ (2): $\frac{\partial a^{(L)}}{\partial z^{(L)}}$ (3): $\frac{\partial C_0}{\partial a^{(L)}}$

These three expressions are actually the chain rule. Multiplying these expressions together gives us the desired information.

$$\frac{\partial C_0}{\partial w^{(L)}}$$

To see what the actual derivatives of these three expressions are, recall the following
$C_0 = (a^{(L)} - y)^2$
$a^{(L)} = \sigma(z^{(L)})$
$z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$

(1): $\frac{\partial z^L}{\partial w^{(L)}} = a^{(L-1)}$ (2): $\frac{\partial a^L}{\partial z^{(L)}} = \sigma'(z^L)$ (3): $\frac{\partial C_0}{\partial a^{(L)}} = 2(a^{(L)} - y)$

So we can write all of this as:

$$\frac{\partial C_0}{\partial w^{(L)}} = \frac{\partial z^L}{\partial w^{(L)}} \frac{\partial a^L}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} = a^{(L-1)}\sigma'(z^L)2(a^{(L)} - y)$$

Keep in mind that this expression is only for one given training example. This needs to be averaged over all the training examples. The formula for the full cost function is:

$$\frac{\partial C}{\partial w^{(L)}} = \frac{1}{n}\sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}$$

which is just for this one specific weight, which is just one component of the gradient vector ∇C.

Looking at changes for the bias, it is necessary to exchange the first term $\frac{\partial z^L}{\partial w^{(L)}}$ with $\frac{\partial z^L}{\partial b^{(L)}}$. From $\frac{\partial a^L}{\partial z^{(L)}}$ we see that the derivative is 1 which results in losing the first term of the original equation

$$\frac{\partial C_0}{\partial b^{(L)}} = \sigma'(z^L)2(a^{(L)} - y)$$

From the term $z^{(L)} = w^{(L)}a^{(L-1)} + b^{(L)}$, we can see that we tackled every term besides $a^{(L-1)}$ which is the activation value of the previous layer. This is analogous to the calculation for the bias above. We are also going to exchange the first term, this time with $\frac{\partial z^L}{\partial a^{(L-1)}}$, and calculate the new derivative which results in:

$$\frac{\partial C_0}{\partial a^{(L-1)}} = w^{(L)}\sigma'(z^L)2(a^{(L)} - y)$$

Since we can not influence the value of the previous activation we can at least store it and iterate this idea backwards to previous layers. Hence, the term backpropagation is derived.

Since we looked at a network where there is only one neuron per layer it is necessary to look at this with a more realistic example. In order to support multiple neurons per

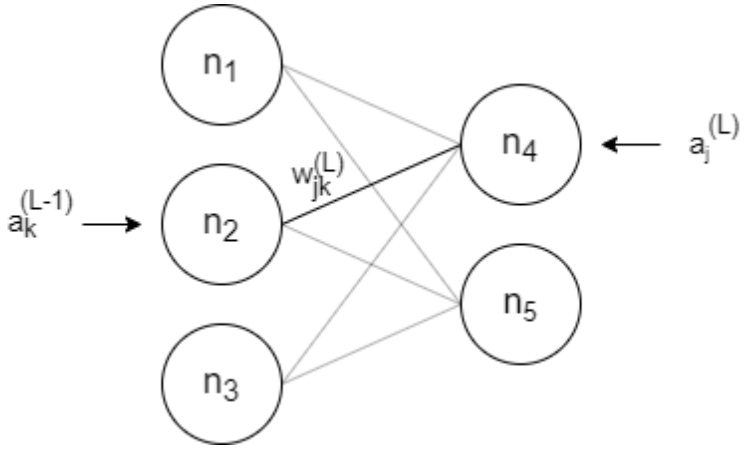

Figure 2.5: Example network with multiple neurons per layer

layer, the equations mostly do not change too much but it is necessary to introduce some more indexing. Consider this network and ignore greyed out connections for now:

Since this network has 2 layers the concrete description for the weight connecting n_2 and n_4 would be w_{01}^1 if indexing of layers and number of neuron in each layer starts with 0. Generally speaking, it says it is the weight from the k^{th} neuron in the $(l-1)^{th}$ layer to the j^{th} neuron in the l^{th} layer.

The first thing that changes in the mathematics is the definition of the cost. Since we have multiple neurons in the last layer, we have multiple outputs in our case called y_0 and y_1 hence the cost is defined as $\sum_{j=0}^{n_L-1}(a_j^{(L)}-y_j)^2$. Looking at neuron n_4, we have to take into account that there are 3 weights and activation values from n_1, n_2 and n_3 which we have to consider in the formula for $z_j^{(L)}$ which in this case would be $z_0^{(0)}$. This can be generalized for each neuron in a layer by $z_j^{(L)} = w_{j0}^{(L)}a_0^{(L-1)} + w_{j1}^{(L)}a_1^{(L-1)} + w_{j2}^{(L)}a_2^{(L-1)} + b_j^{(L)}$, thus we rewrite $a^{(L)} = \sigma(z^{(L)})$ to $a_j^{(L)} = \sigma(z_j^{(L)}$. With this knowledge, we can now rewrite the formula which tells us how sensitive the cost is to a specific weight to more generalized form as well

$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^L}{\partial w_{jk}^{(L)}}\frac{\partial a_j^L}{\partial z_j^{(L)}}\frac{\partial C_0}{\partial a_j^{(L)}}$$

The formula which tells us the sensitiveness about the previous layers' activation however changes a bit more

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \sum_{j=0}^{n_L-1}\frac{\partial z_j^L}{\partial a_k^{(L-1)}}\frac{\partial a_j^L}{\partial z_j^{(L)}}\frac{\partial C_0}{\partial a_j^{(L)}}$$

This is due to the fact that a neuron in layer $L-1$ can influence two or more neurons in the next layer. For example in the network illustrated in Figure 2.5, we can see that the

activation value of neuron n_2 has influence on neuron n_4 and n_5 which means it has also an impact on the activation value of n_4 and n_5. This is the reason for taking the sum over the layer L. The formula for the bias stays the same.

2.2 RNN

A Recurrent Neural Network (RNN) introduces cycles in the network, which allows them to handle the aspect of time. Other applications could be speech synthesis or music generation. Basically, RNNs are good at solving problems, which rely on knowing the current and all previous inputs. RNNs use so-called feedback loops which are used to learn from sequences which can also vary in length. RNNs also require the input to be 3 dimensional. These dimensions can be seen as follows for a time series problem:

- quantity of samples
- quantity of timesteps in a single sample
- quantity of values observed per timestep, also known as features

This is often seen as a triple in the form: [samples, timesteps, features]. On the left-hand side of Figure 2.6 the basic architecture of a RNN can be seen [13]. It has an input x_t and an output h_t. A takes the input and produces the output. The arrow going from A to A allows the RNN to pass information from one point in the network to the next. This can be seen, if the network is unrolled which is on the right-hand side of Figure 2.6. This might be looking confusing at first, but it can be thought of copies of the same network which pass information to its successor. This architecture is what makes this type of neural network so good for tasks like speech synthesis.

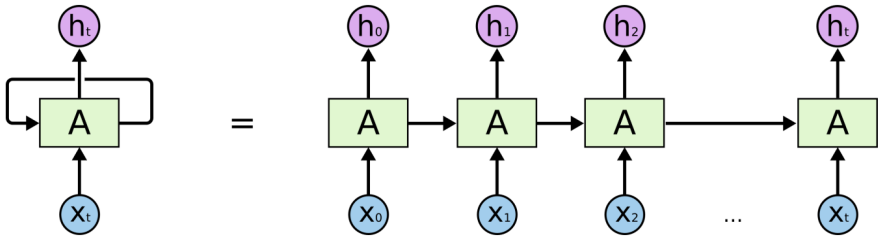

Figure 2.6: Unrolled RNN [13]

However there are two major issues when it comes to RNNs:

1. Learning: Learning in RNNs is done with backpropagation through time, which is basically just unrolling the network and then applying normal backpropagation for learning. Due to the unrolling, the network becomes very deep, which increases the chance of the vanishing or exploding gradient problem to occur. This issue is discussed in Section 2.2.1.

2. Long-Term dependencies: The network will not be able to connect relevant information which is far apart, thus failing to give the correct output. Imagine a network which should predict the last word of a sentence. A sentence like 'Planets are rotating around the *sun*.' it should be pretty obvious that the result is going to be 'sun'. Now consider a sentence like 'I am currently studying for my motorcycle license and I really enjoy driving so far, thus I will probably buy a *bike*'. In order to know what will probably be bought, it is necessary to look all the way back to the word 'motorcycle', which will be difficult for the RNN. This issue was examined in more detail by Hochreiter [14], which ultimately resulted in the introduction of Long Short-Term Memory (LSTM) in 1997 [15]. LSTM Networks are discussed in more detail in Section 2.3.

2.2.1 Vanishing and exploding gradient problem

In [11], the author examines the speed of learning per hidden layer in the network, summarized in Figure 2.7.

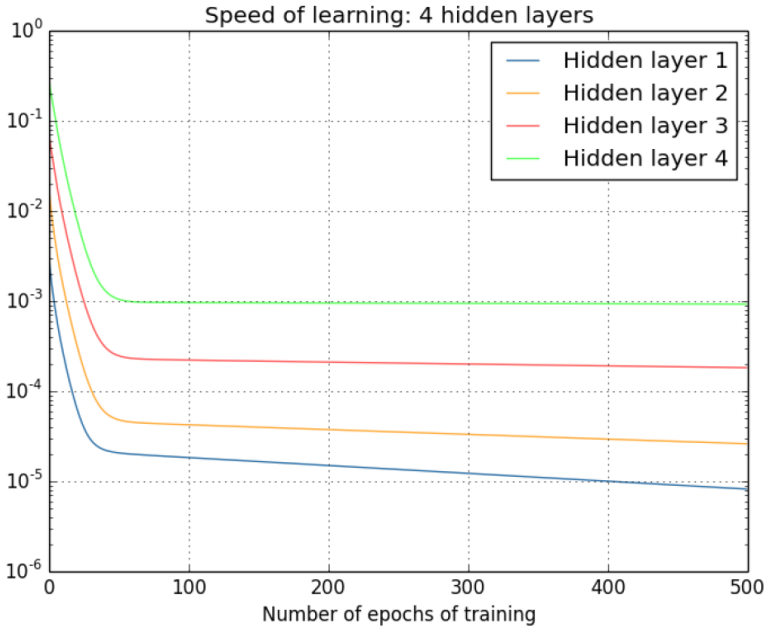

Figure 2.7: Learning speed per hidden layer [11]

Note that hidden layer 4 represents the last hidden layer which is before the output layer, which means that layer 4 is the first layer where backpropagation is applied. We can see that earlier layers in the network are learning slower than later ones. The cause of this is the vanishing gradient problem, which means that the gradient described in Section 2.1.2 becomes smaller with every hidden layer.

To understand why this problem occurs, consider the network shown in Figure 2.8 and

the following equation which describes the changes in the cost function C with respect to changes made with b_1

$$\frac{\partial C}{\partial b_1} = \sigma'(z_1) \times w_2 \times \sigma'(z_2) \times w_3 \times \sigma'(z_3) \times w_4 \times \sigma'(z_4) \times \frac{\partial C}{\partial a_4}$$

To see why this equation is correct refer to [11] page 160. Assume that the activation function σ in this case is the sigmoid function and the weights are initialized using a normal distribution with mean 0 and standard deviation of 1. Due to the chosen normal distribution the weights will satisfy $|w_j| < 1$ most of the time. The maximum of the derivative is at $\sigma'(0) = \frac{1}{4}$. So multiplying these terms will mostly result in $w_j\sigma'(z_j) < \frac{1}{4}$. Since we have four of these terms in the equation and the result of one such multiplication is almost guaranteed to be less than 1 the result tends to get smaller with every multiplication, or in other words the result gets smaller the deeper we go into the network.

On the other hand, there is the exploding gradient problem, which can also be described with an example using the network shown in Figure 2.8 even if the example is a bit made up. Assume $w_2 = w_3 = w_4 = 100$. For the biases, we want that $z_j = 0$, thus $\sigma' z_j = 1/4$. z_j is defined as $z_j = w_j a_{j-1} + b_j$ so we choose for the bias $b_j = -100 \times a_{j-1}$. Note that a_j denotes the output of the previous neuron. Using the assumed values $w_j\sigma'(z_j)$ becomes $100 \times 1/4 = 25$. Multiplying these values is going to let the gradient become very large very quickly. This is why it is called exploding gradient.

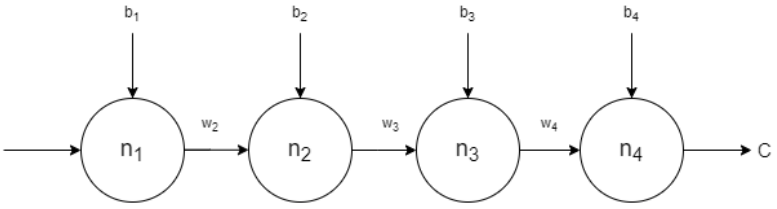

Figure 2.8: Example network for the vanishing gradient problem

2.3 LSTM

Long Short-Term Memory (LSTM) was introduced in 1997 by Hochreiter and Schmidhuber [15]. The advangtage of LSTM networks is that they are capable of remembering information for a longer period of time [13]. A standard RNN will have a simple structure which could be a single *tanh* layer which takes the input from the preceding part of the network, the current input x_t and creates an output h_t and feeds it to succeeding part of the network. However, in an LSTM network there are four layers which can be seen in Figure 2.10. The layers are the three σ symbols and the *tanh* function.

The main part is the so-called cell state which is represented by the straight line going from C_{t-1} to C_t. Then there are 3 gates which are represented by a sigmoid function and a pointwise multiplication. These gates are able to regulate how much information

is passed through. A value of 1 means let all information through, a value of 0 means let no information through. The internal structure of the LSTM can be explained in 4 steps.

1. The first layer (from left to right) is the forget gate layer. It takes the inputs h_{t-1} x_t and outputs a number between 0 and 1 for each number in C_{t-1}. This is expressed by $f_t = \sigma W_f \cdot [h_{t-1}, x_t] + b_f$ where W_f denotes the weight for the forget gate, h_{t-1} the output of the preceding block, and b_f the bias for the forget gate. These notations are also being used for the input gate and the output gate following in the next steps, only the subscript will change accordingly.

2. The next step is made of two parts, the first one is the input gate expressed by $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$. The second part is a layer which finds a vector of new candidates, $\tilde{C}$, for the cell state which is defined as follows: $\tilde{C}_t = tanh(W_C \cdot [h_{t-1,x_t} + b_C])$. In the following step the result of the input gate and the vector $\tilde{C}$ get combined.

3. The third step is updating the cell state C_t-1. This is done by multiplying f_t from the first step with C_{t-1}. With this we actually forget everything we wanted to forget in the first step. Now we add the result of $i_t * \tilde{C}_t$ which represents the new values scaled by how much each state value is updated. C_t is then simply defined by $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$.

4. In the last step the output is decided, which depends on the cell state, but the cell state C_t will be filtered by *tanh* first such that the values are scaled between -1 and 1. Also the output gate calculates what we are going to output as follows $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$. This leaves us with h_t being defined as $h_t = o_t * tanh(C_t)$

LSTM was designed to alleviate the the problem of the vanishing gradient discussed in Section 2.2.1. DiPietro R. et al. [7] explain why LSTM and GRU models handle the vanishing gradient better than ordinary RNN models in the following way: Looking at the equation $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$ which describes the new cell state it is important to see that there is only one path between C_t and C_{t-1} which is modelled by the forget gate f_t. However in the context of some time τ where $t > \tau$ we can have exponentially many paths from C_t to $C_{t-\tau}$ but one of these paths is just the elementwise multiplication of forget gates. This represents another gradient component which is the product of diagonal Jacobians with diagonal elements corresponding to the forget gates. Which means that the gradient can still vanish exponentially over τ, however this is not the case if the forget gates have elements that are close to 1, because the base will also stay close to 1. This is the reason, why the bias of forget gates is often initialized to some positive number like 1 or 2.

Additionally to the "plain" LSTM model there exist various other models for example LSTM models which introduce peepholes [16] or hybrid CNN-BiLSTM [17] models and many more variations.

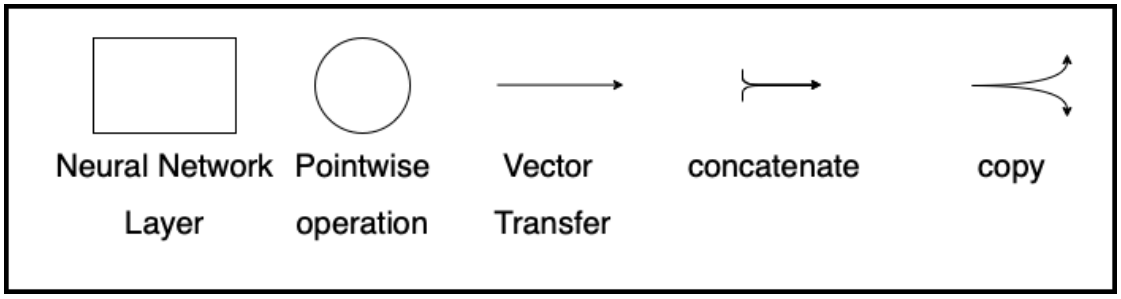

Figure 2.9: LSTM Operators [13]

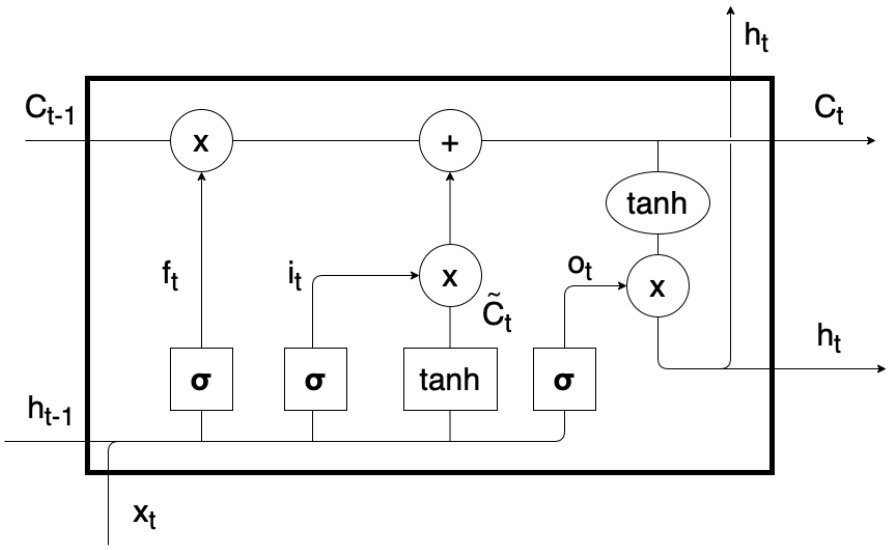

Figure 2.10: Detailed LSTM architecture [13]

2.4 GRU

Gated Recurrent Units (GRUs) were introduced in 2014 by Kyunghyun Cho et al. [18]. In their paper, they compared two machine learning models, one being an RNN Encoder-Decoder and the other being their new *gated recurisve convolutional neural network*. Their new network replaces the Encoder part in the first model.

GRUs are a special case of LSTM networks [19]. It was found that performance in speech signal modeling was similar to LSTM. Also the training time was shorter compared to LSTM networks and it has fewer parameters since GRUs do not have output gates. Essentially GRUs use two inputs at a given time, which are the previous input $h(t-1)$ and the input vector $x(t)$. Like mentioned earlier. GRU uses only 3 gates which are the following:

$z(t)$ describes the update gate vector

$r(t)$ describes the reset gate vector
$\tilde{h}_t$ describes the resulting memory
$h(t)$ describes the resulting memory

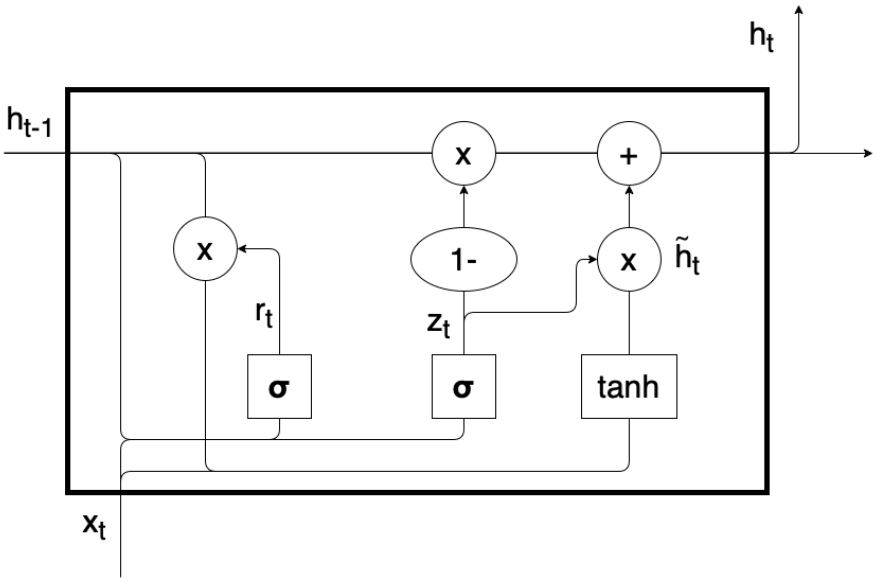

Figure 2.11: Detailed GRU architecture [13]

Mathematically these gates can be expressed in the following way [13]:

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h_t} = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h_t}$$

Compared to LSTM, GRU combines the forget and input gates to the single update gate z_t [13]. Also the cell state and hidden state are merged in combination with other smaller changes.

CHAPTER 3

Implementation

3.1 Used Technologies

For the development of the application, various technologies have been used. First of all, Keras [20] was used to train all of the models. Keras is a deep learning API written in Python, running on top of the machine learning platform TensorFlow. TensorFlow [21] is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in machine learning (ML), and developers easily build and deploy ML powered applications.

For faster training, the data [2] has been stored in an influxDB [22] to retrieve the data faster. Also potential missing datapoints in the CSV data have been interpolated linearly before inserting them into the database. After the training of the models is done, a webserver running on Node.js is providing the trained models.

With all the aforementioned technologies the Web application which is built in angular can download the desired model. After that the Web application can either evaluate a model in a specific time range from the data or simulate a live prediction on the data [2].

To show the simulation or evaluation of a model to the user the plotly.js [23] library has been used. Plotly.js is built on top of d3.js and stack.gl and is free as well as open source. It is also possible to evaluate all of the models available and export the data as Excel files which can be used to validate the results in this thesis.

The data which has been used comes from [2] and represents the total photovoltaic power generation of Austria in the timeframe from 2015-01-01 to 2020-12-31 in 15 minute timesteps.

3.2 Architecture

This section describes the architecture used to generate and evaluate the machine learning models. The architecture can be seen in Figure 3.1.

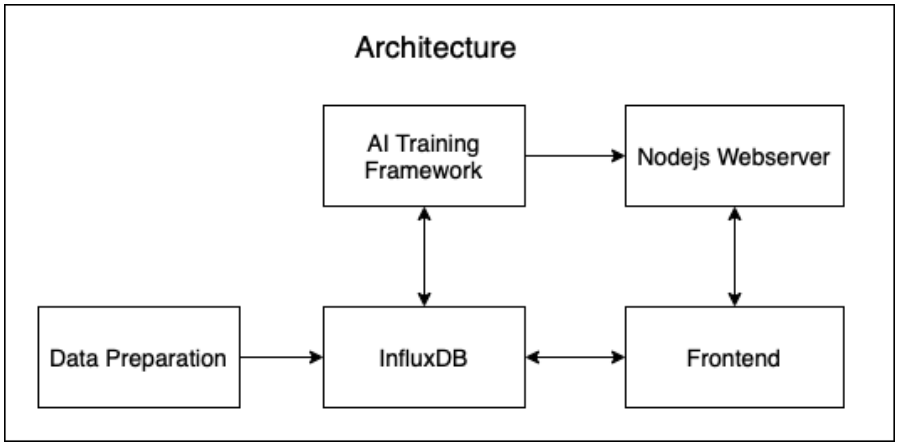

Figure 3.1: Architecture of the implemented application

3.2.1 Data Preparation

The Data Preparation component is implemented with Python and can store two datasets to the influxDB. First of all, it is used to store the relevant data from [2] in the influxDB. The relevant data includes the date/time column and the AT_solar_generation_actual column. In case timesteps are missing in the data, the missing timesteps are interpolated linearly. The timestamps are in 15 minute intervals. The timesteps are from minute 00 to 15 to 30 to 45 and so on.

The second set of data is from [5] and ranges from the year 2015 to 2021. Every year camet in its own file so the data was merged to one file. Furthermore, the data was provided in 10 minute intervals so it was necessary to fit the timesteps in the 15 minute intervals from the first dataset. This was also done with linear interpolation within the minutes of 10 to 20 and 40 to 50 to get minute 15 and minute 45 of each hour, respectively. The relevant data from the ZAMG dataset [5] is the cGLOm column. This column represents the global radiation which is direct radiation and diffuse radiation combined. The interpolated data of the second dataset got also inserted to influxDB.

3.2.2 influxDB

influxDB [22] is a timeseries database which allows for fast read and write access. It was used because it is possible to query the data more easily compared to a CSV file which is loaded via pandas [24] with Python. Also influxDB provides a useful webinterface which allows to filter data quickly and it can display the queried data in a graph. Furthermore, influxDB can run inside a docker container which emphasizes platform independence.

Unlike conventional databases, influxDB uses flux instead of SQL which is a functional data scripting language which is also used for querying data. This scripting language has been used throughout the project. This is an example of the flux query language:

```
from(bucket: "photovoltaic_bucket")
    |> range(start: -1d)
    |> filter(fn: (r) => r._measurement == "mem"
    and r._field == "cGLOm")
```

This query selects measurements from the photovoltaic_bucket in the last day and then filters for a measurement called "mem" and a field which is called "cGLOm". The bucket can be seen as a table like in ordinary SQL databases and the filter function acts like a "where" clause. The influxDB is listening on port 8086.

3.2.3 AI Training Framework

The AI Training Framework is the core part of this thesis. It is implemented in Python and has multiple purposes. First of all, it makes sure that the timeseries data is present in the influxDB, otherwise the data gets inserted before anything else happens. This is done by using the data preparation part of the project. After that it is possible to train models. It is possible to train either LSTM or GRU models, and for each one can choose if the training happens with or without solar irradiance data [5]. Also it can be decided, if just a single model should be generated or if a set of models should be generated. If the single model is chosen then the user has to fill in a config in the terminal. The single training option allows to configure the following parameters:

- start date - specifies the startdate for the training date range.
- end date - specifies the enddate for the training date range.
- epochs - specifes the number of times the gradient descent algorithm goes through the model.
- batch size - specifes the number of samples the gradient descent algorithm goes through, before internal parameters of the model are updated.
- cut time from dataframe - lets the user choose if specific parts of the date range should be removed for training (could be used to cut out nighttime to speed up training time).
- foldername - specifies the name of the folder the generated model is going to be stored.

After parameter selection, the model is trained and stored. Also the configuration and duration of the training is saved in the folder of the model. To save the model two different methods are used. The first one being tensorflowjs version for Python which can save a Keras model and as second option the model.save() function provided by the Keras model itself. Exact implementation details of the LSTM and GRU models are discussed in the LSTM and GRU sections below.

Furthermore, it is possible to generate a set of models automatically. The set consists of models with the following epochs: 20, 40, 50, 60, 70, 100, 150 and the following batch sizes: 32, 64, 128. This makes 21 different models. In order to reduce statistical errors during training and therefore reduce outliers in the results each possible configuration for a model gets generated 5 times which makes a total of 105 models.

3.2.4 Node.js Webserver

The Node.js webserver component is used as a static file server which provides the models trained by the AI training framework. The Node.js webserver is serving the models on port 81 and has 5 different endpoints which provide the models for lstm, lstm with solar irradiance data, gru and gru with irradiance data, respectively. The last endpoint is used to show the live simulation. To run the webserver on the Raspberry Pi permanently, the npm package forever [25] is used. Forever is a simple CLI tool for ensuring that a given script runs continuously.

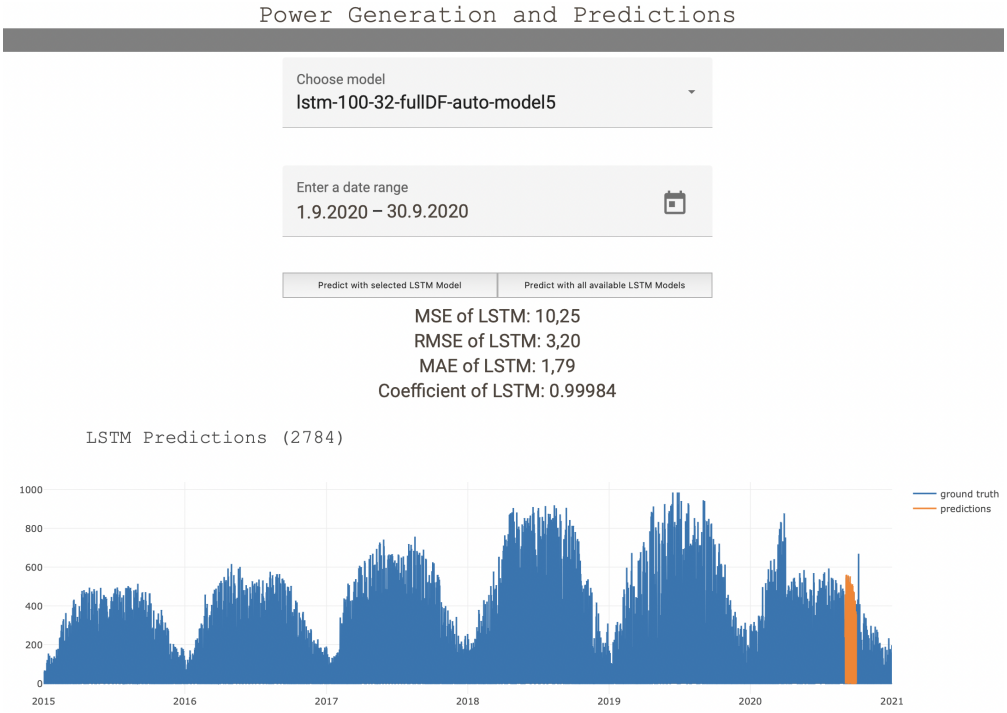

Figure 3.2: Frontend after evaluation of a single model

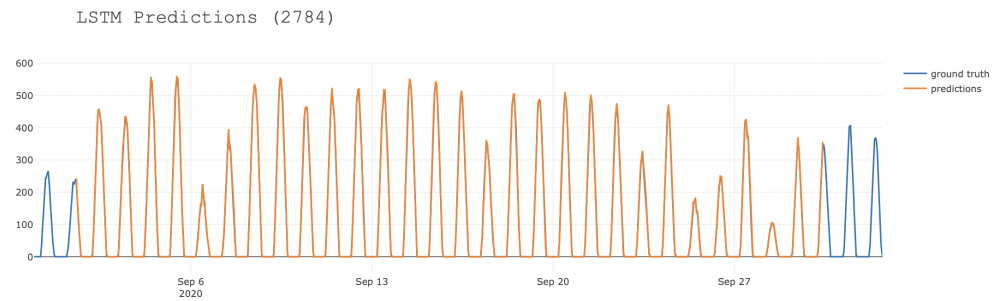

Figure 3.3: Interactive graph zoomed in to predicted month

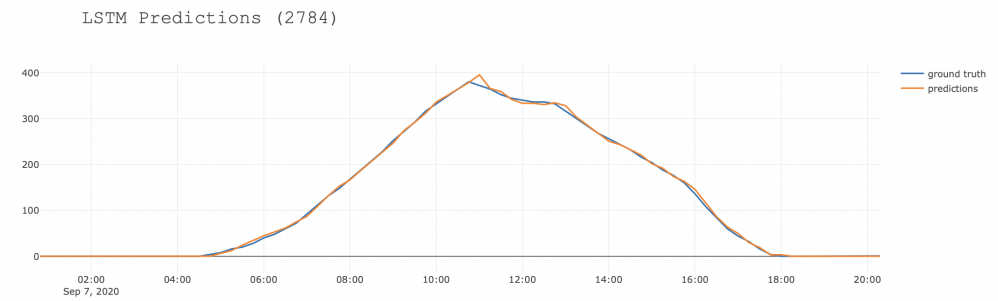

Figure 3.4: Interactive graph zoomed in to a day of the predicted daterange

3.2.5 Frontend

The last part is the frontend which is the second core part of the application. It is used to evaluate trained models. For the evaluation, the desired model and desired date range can be selected. After some time, depending on the length of the date range, the results are displayed on the Web page and a graph is plotted which shows the groundtruth and the predictions made by the model in the desired date range. It is possible to zoom into the graph to see the differences of prediction and groundtruth in detail. Additionally, it is possible to evaluate all models of the same kind at the same time. Therefore, the frontend has a total of 4 routes corresponding to lstm, lstm with irradiance, gru and gru with irradiance respectively. When evaluating in batches the results are rendered into a table. As soon as the batch evaluation is finished the table can be exported as Excel file.

Another core part of the frontend is the live prediction of photovoltaic power output. The last 7 ground truth values are fed into the model the user has to select. Then, the application continuously predicts the photovoltaic power output of the next timestep. A single timestep is, like the groundtruth, in 15 minute intervals. For the proof of concept developed the timestep is reduced to 5 seconds to see the live graph in action. Every 5 seconds a prediction for the next 15 minute interval is predicted and the time gets increased by 15 minutes. So the time is speed up.

Lastly every component of the application is platform independent and has been tested

on Windows, macOS and Raspberry Pi OS.

3.3 LSTM

Figure 3.5 shows the structure of the LSTM model used in this thesis. $x_1, x_2, ...x_t$ are the ground truth values which are given to the model. This is followed by two dense layers where ud_1 and ud_2 describe the size of the output of the respective dense layer. In this model, t was set to 200, ud_1 and ud_2 to 100 and 7, respectively.

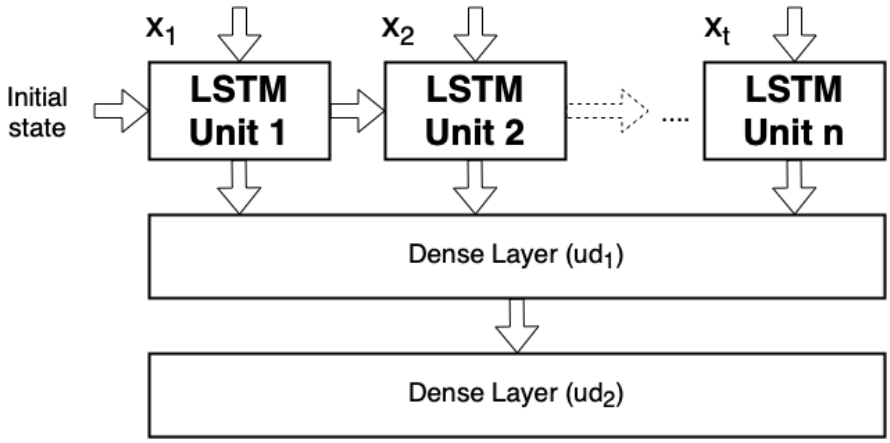

Figure 3.5: Structure of the LSTM model used for evaluation [26]

The LSTM Units use a rectified linear unit as activation function. Due to the structure of the LSTM model, the data needed to be put into a fitting shape. The input shape to the LSTM Units is [number_of_input_samples, number_of_timesteps, number_of_features]. In practise it could look like [35040, 7, 1] where 35040 is the number of quarter hours in a whole year which is calculated with 96×365. 7 is the number of timesteps which are fed into the model at a time and 1 is the number of features. In the case where the solar irradiance data is also used the shape becomes [35040, 7, 2], since for every timestep the photovoltaic power output and the solar irradiance data are fed into the model. As loss function the mean squared error was used. The optimizer which was used is adam [27]. The optimizer is used to adjust the weights and biases in a neural network.

3.4 GRU

The structure of the GRU models is almost identically to the LSTM models. Looking at Figure 3.5, the only difference is that the LSTM units are replaced with GRU units. This was done to ensure the best comparability possible between the two models.

CHAPTER 4

Results

Three different machine learning models have been taken into account. Each of the models uses a different count of epochs and batchsize. To reduce outliers, every model has been trained and evaluated 5 times. After that the results have been normalized using

$$x_{norm} = \frac{(x - x_{min})}{(x_{max} - x_{min})}$$

Then, the average of the results has been calculated. Regarding the evaluation the terms RMSE (root mean squared error), MAPE (mean relative percent error), MAE (mean absolute error) and COEFF (pearson correlation coefficient) are defined as follows:

$$MSE = \frac{1}{n}(\sum_{i=1}^{n}(x_i - y_i)^2)$$

$$RMSE = \sqrt{\frac{1}{n}(\sum_{i=1}^{n}(x_i - y_i)^2)}$$

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|x_i - y_i|$$

$$MAPE = \frac{100\%}{n}\sum_{i=1}^{n}|\frac{x_i - y_i}{x_i}|$$

$$COEFF = \frac{\sum_{i=1}^{n}((x_i - \overline{x})(y_i - \overline{y}))}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2 \sum_{i=1}^{n}(y_i - \overline{y})^2}}$$

All models have been trained in the same timeframe ranging from 2019-08-01 00:00:00 to 2020-08-01 00:00:00. Also, each model has been evaluated in the same timeframe ranging from 2020-09-01 13:45:00 to 2020-09-05 10:00:00.

Every model has been trained on the following setup:

- CPU: AMD Ryzen™ 7 3700X
- RAM: 32GB DDR4 3200Mhz Corsair VENGEANCE® with XMP enabled
- Mainboard: MAG B550M MORTAR WIFI
- SSD: Samsung 980 1 TB PCIe 3.0 NVMe
- Operating System: Windows 10

Note that it was also tried to train the models on a MacBook Air with the M1 chip, but these results were note representable, since in the longer runs, especially when training models above 100 epochs, the laptop struggled with thermal throtteling.

The results below are all single step ahead forecasts which predict the next 15 minute timeframe. All of the results have been made two times. One time with only one feature as input which was just the previous time series. The second table of results always shows the results with two input features where the first one is the previous time series and the second one the solar irradiance from Wien Hohe Warte.

4.1 LSTM

Since the data in [26] got normalized, the results from Table 4.1 can be normalized using $x_{min} = 0$ and $x_{max} = 544$ since 544 is the maximum value in the evaluation timeframe. $x_{max} = 944$ would be the maximum value for the training data. For the best result of Table 4.1, the normalized values using $x_{max} = 944$ are $MSE = 0,010624$ $RMSE = 0,003353$ $MAE = 0,001939$ $COEFF = 0,001059$. Comparing the results with [26] is hard because the authors use a single 4kWp photovoltaic array which is way more prone to weather phenomena compared to this thesis which uses photovoltaic data from all of Austria combined.

4.1.1 LSTM with solar irradiance

It is immediately visible that the LSTM models trained with the solar irradiance data as additional feature performs worse than the models from Table 4.1 and Table 4.3. The best model which is trained with solar irradiance has an MSE which is 20,7× worse compared to the 100E-32B LSTM model without solar irradiance data. To understand the decrease in performance, we take a look at the actual data and the irradiance data during the testing timeframe in Figure 4.2. The lightblue line shows the current photovoltaic power output and the purple line shows the current solar irradiance. The

LSTM Model results					
Model with configuration	MSE	RMSE	MAE	COEFF	Time
20E-32B	10,36	3,22	1,84	0,99973	117,0s
20E-64B	12,40	3,51	2,15	0,99972	74,0s
20E-128B	12,27	3,49	2,09	0,99972	68,0s
40E-32B	11,58	3,40	1,99	0,99972	233,0s
40E-64B	11,45	3,38	1,99	0,99972	158,8s
40E-128B	10,19	3,19	1,88	0,99973	136,2s
50E-32B	11,44	3,37	1,96	0,99973	301,8s
50E-64B	12,42	3,46	2,04	0,99972	203,8s
50E-128B	11,22	3,35	1,98	0,99972	187,4s
60E-32B	10,94	3,31	1,97	0,99972	373,8s
60E-64B	13,57	3,68	2,17	0,99971	255,0s
60E-128B	11,17	3,33	1,98	0,99972	229,6s
70E-32B	11,51	3,38	1,92	0,99972	429,4s
70E-64B	13,03	3,57	2,06	0,99972	285,8s
70E-128B	12,28	3,49	2,03	0,99972	256,2s
100E-32B	**10,03**	**3,17**	**1,83**	**0,99973**	**626,0s**
100E-64B	10,30	3,20	1,83	0,99972	423,2s
100E-128B	10,25	3,20	1,87	0,99973	374,8s
150E-32B	13,98	3,68	2,21	0,99973	954,8s
150E-64B	11,15	3,33	1,92	0,99972	631,2s
150E-128B	10,42	3,22	1,85	0,99973	539,8s
Average	11,52	3,38	1,98	0,99972	326,65s

Table 4.1: Performance of different LSTM models

red boxes mark interesting spots, because the solar irradiance is going down significantly while the photovoltaic poweroutput stays unaffected. Note that the current photovoltaic output is measured in MW and the current solar irradiance is measured in W/m^2. Since photovoltaic output stays mostly unaffected by drops of the solar irradiance the question arises how the model behaves. This becomes clear when looking at an arbitrary day in the testing timeframe. For the example in Figure 4.1 the timerange from 2020-09-01 13:45:00 to 2020-09-03 10:00:00 was predicted and the day 2020-09-02 was used as illustration in Figure 4.1. The best LSTM model which was trained with solar irradiance, which is 20E-64B, was used to do the prediction, which can be seen in Table 4.2. The model 20E-64B is the average of 5 generated models, for the predicition in Figure 4.1 the first model of the 5 generated models was used. The MSE of the prediction was 160,25, but the interesting part is how the graph of the groundtruth vs. predicted data looked like. This can be seen in the top graph of Figure 4.1. Looking at the top graph there is a clear shift of the orange line to the right when compared to the blue ground truth line. So it was tested what happens if the prediction line is shifted by one timestep to the left. This yielded an interesting result, because the prediction line almost perfectly matches

the groundtruth. This finding suggests that the solar irradiance feature which was fed to the model during learning was not a good feature for the model to rely on, so the inputs of this feature are weighted down until it gets almost ignored. Due to that feature being ignored it seems as if the models then decided to take the last value of groundtruth which got fed into the model as the next result for the prediction. This would explain why the prediction line almost perfectly fits the ground truth line after shifting. The results when shifting the prediction are very close to perfect. MSE: 0,07 RMSE: 0,26: MAE: 0,15 and a coefficient of 0,99627.

LSTM Models with solar irradiance results					
Model with configuration	MSE	RMSE	MAE	COEFF	Time
20E-32B	209,72	14,48	9,65	0,99634	117,4s
20E-64B	**208,12**	**14,43**	**9,61**	**0,99638**	**81,8s**
20E-128B	211,20	14,53	9,68	0,99633	69,0s
40E-32B	209,56	14,48	9,62	0,99635	233,2s
40E-64B	209,95	14,49	9,64	0,99635	167,0s
40E-128B	209,82	14,48	9,65	0,99634	147,0s
50E-32B	209,77	14,48	9,63	0,99635	293,2s
50E-64B	209,85	14,49	9,65	0,99634	208,6s
50E-128B	211,15	14,53	9,69	0,99634	206,2s
60E-32B	209,89	14,49	9,64	0,99634	369,2s
60E-64B	210,71	14,52	9,65	0,99634	256,4s
60E-128B	209,64	14,48	9,64	0,99634	231,2s
70E-32B	210,03	14,49	9,63	0,99634	435,2s
70E-64B	209,98	14,49	9,64	0,99634	299,8s
70E-128B	210,71	14,52	9,66	0,99634	285,0s
100E-32B	209,21	14,46	9,62	0,99635	635,8s
100E-64B	209,76	14,48	9,63	0,99634	397,0s
100E-128B	211,32	14,54	9,65	0,99632	370,8s
150E-32B	209,66	14,48	9,63	0,99634	918,8s
150E-64B	210,00	14,49	9,63	0,99634	565,6s
150E-128B	210,12	14,50	9,64	0,99634	569,4s
Average	209,99	14,43	9,64	0,99634	326,56s

Table 4.2: Performance of different LSTM models trained with solar irradiance data

One reason why the dips in solar radiation are not present in the photovoltaic power output chart might be due to the fact that Vienna has very little photovoltaic systems compared to other federate states of Austria. The distribution can be seen in [28]. Also when looking at the total power output of the photovoltaic systems in [28], we can see that Vienna is not really relevant compared to other federate states.

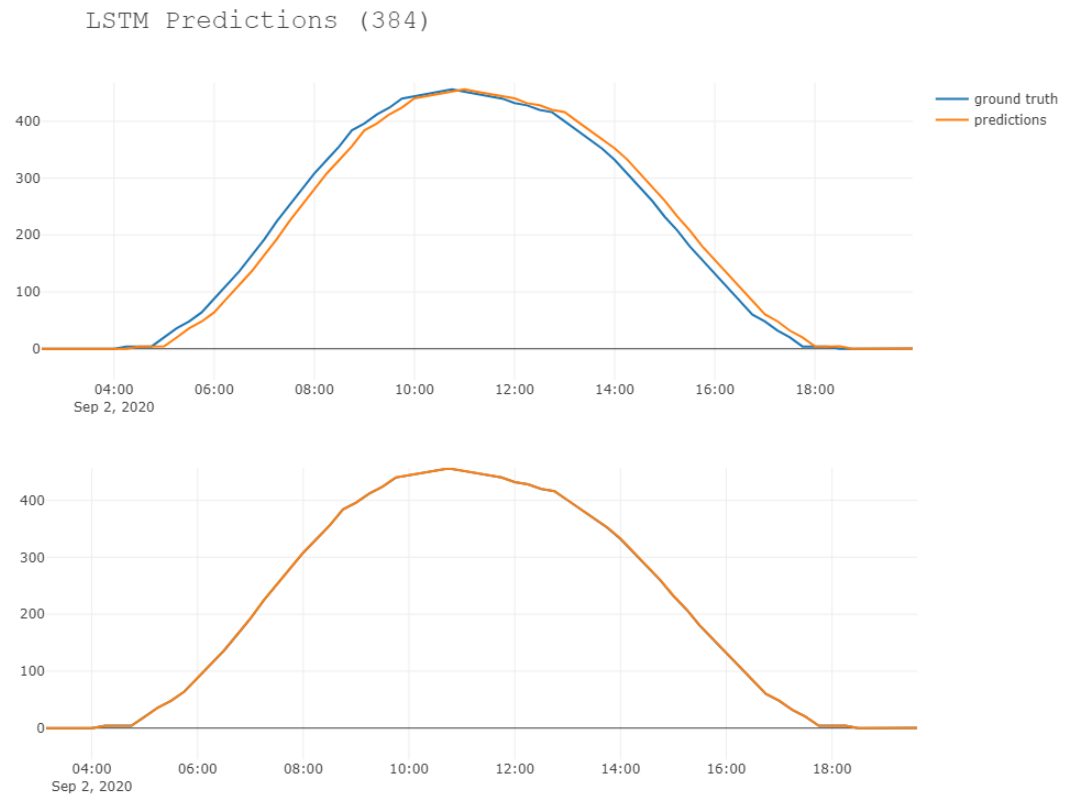

Figure 4.1: Solar model original result vs. shifted result

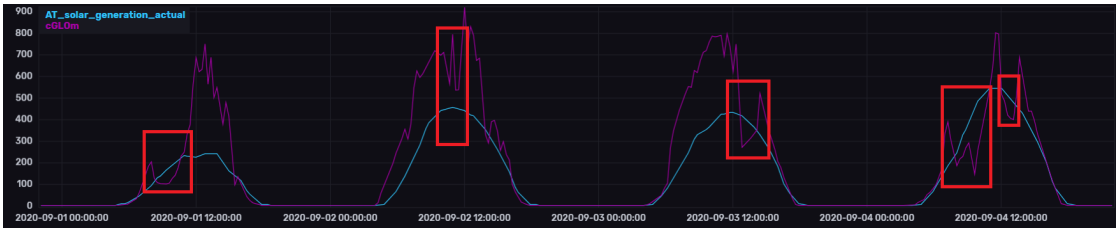

Figure 4.2: Current photovoltaic output and solar irradiance compared

4.2 GRU

For the GRU models in Table 4.3, we can see a clear winner which is the GRU-150E-64B model. The MSE of this model is 2,52 below average of all GRU models. Compared to the LSTM models, its MSE is better by 1,07. Comparing the training times between the best GRU model and the best LSTM model we get an average of 826,5 seconds for the GRU-150E-64B model and an average of 326 seconds for the LSTM model. This means GRU-150E-64B needs 253% more time to train compared to LSTM-70E-32B, on

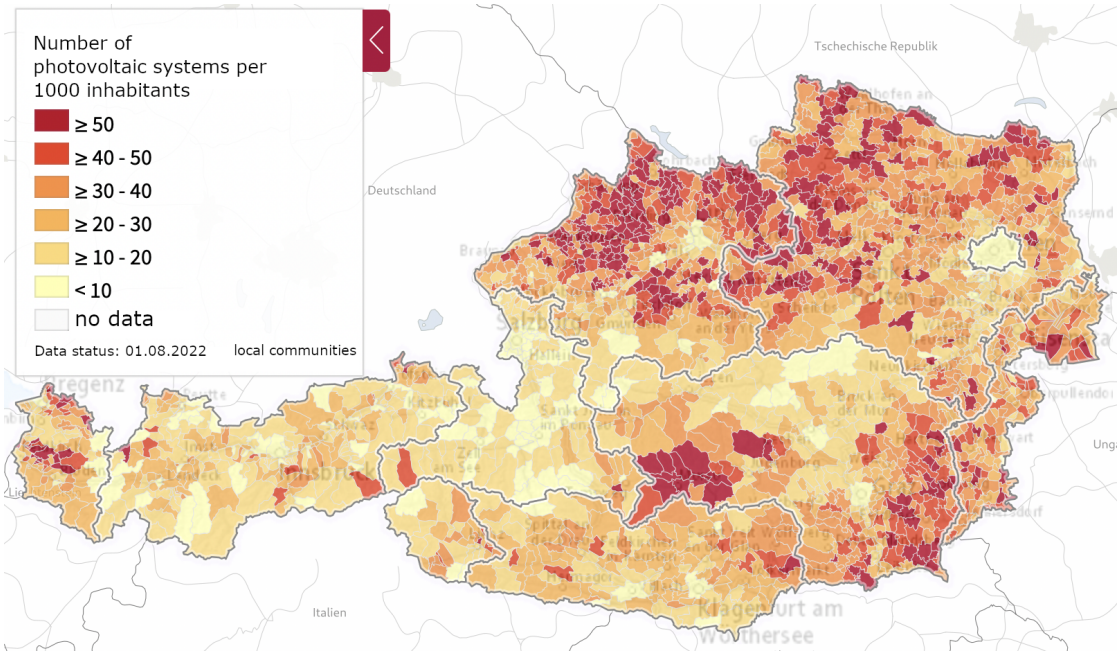

Figure 4.3: Photovoltaic systems per 1000 residents in local Austrian communities

the other hand its performance regarding the MSE is 11% better. So the takeway is: if training time does not matter then the GRU-150E-64B should be used, otherwise the LSTM-100E-32B.

Looking at all GRU models, we see that they have a better performance on average compared to the average of all LSTM models.

4.2.1 GRU with solar irradiance

Similar to the results from the LSTM models with solar irradiance the performance is almost identically bad. Compared to the GRU models which were trained without solar irradiance data, the average performance is 18.31× worse. Comparing the best performance of GRU and GRU with solar irradiance data 20E-64B performs $23,30\times$ worse than 150E-64B. Looking at the visuals of the results, the prediction vs. ground truth chart is also almost identical to the shifting observed in Figure 4.1. Due to this fact, it seems that the GRU models which were trained with solar irradiance data learned the same behavior as the LSTM models trained with solar irradiance data. A screenshot of the web application can be seen in Figure 4.4

4.3 Web application on Raspberry Pi

Like stated in the goals section, the Web application has been built and deployed on a Raspberry Pi 4 Model B 8GB. The Web application got deployed via a Node.js server

GRU Model results					
Model with configuration	MSE	RMSE	MAE	COEFF	Time
20E-32B	10,95	3,30	1,95	0,99972	151,6s
20E-64B	13,67	3,65	2,15	0,99973	87,4s
20E-128B	11,73	3,42	2,08	0,99972	74,0s
40E-32B	10,83	3,29	1,89	0,99972	267,2s
40E-64B	12,69	3,53	2,06	0,99970	182,4s
40E-128B	11,42	3,37	1,99	0,99972	151,8s
50E-32B	13,38	3,63	2,16	0,99972	361,2
50E-64B	11,74	3,41	1,99	0,99973	251,6s
50E-128B	11,52	3,38	2,04	0,99972	210,4s
60E-32B	10,24	3,20	1,87	0,99972	434,4s
60E-64B	12,18	3,47	2,04	0,99973	300,6s
60E-128B	11,77	3,42	2,03	0,99971	266,2s
70E-32B	16,55	3,97	2,35	0,99973	551,2s
70E-64B	10,66	3,26	1,84	0,99973	364,6s
70E-128B	10,34	3,21	1,88	0,99972	298,4s
100E-32B	10,09	3,17	1,82	0,99973	748,4
100E-64B	10,64	3,26	1,90	0,99974	511,4s
100E-128B	10,36	3,21	1,84	0,99973	440,8s
150E-32B	11,39	3,36	1,93	0,99972	1095,6s
150E-64B	**8,96**	**2,99**	**1,72**	**0,99974**	**711,2s**
150E-128B	9,92	3,15	1,79	0,99973	614,0s
Average	11,48	3,375	1,97	0,99973	437,30s

Table 4.3: Performance of different GRU models

running on the Raspberry Pi. The required influxDB and the Node.js webserver which serves the models is running on a local machine in the network. In the liveview, the user is able to view the 4 best models of each table in the results of this thesis. The 4 models predict the current groundtruth step, which is in 15 minute intervals, and the predictions get rendered in the graph. The user is able to speed up or slow down the simulation and can stop and resume it. Currently the maximum is about 8 to 10 predictions per second while rendering the graph and viewing the graph via teamviewer.

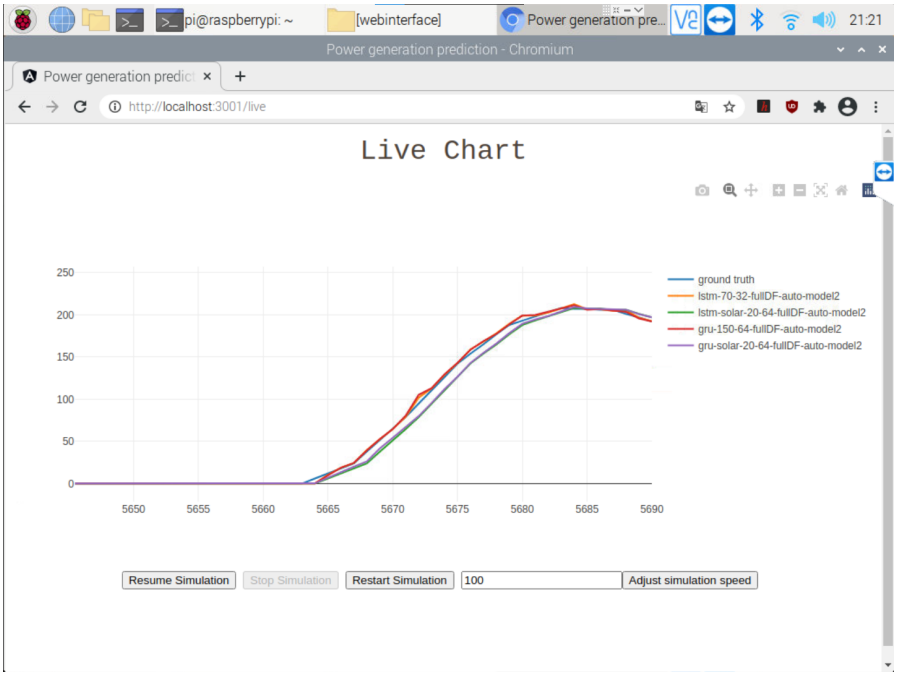

Figure 4.4: Web application running the live simulation on a Raspberry Pi 4 Model B 8GB

GRU Models with solar irradiance results					
Model with configuration	MSE	RMSE	MAE	COEFF	Time
20E-32B	208,99	14,46	9,63	0,99635	156,4s
20E-64B	**208,85**	**14,45**	**9,62**	**0,99636**	**103,4s**
20E-128B	210,75	14,52	9,64	0,99635	74,0s
40E-32B	209,62	14,48	9,62	0,99635	297,6s
40E-64B	210,18	14,50	9,64	0,99635	198,8s
40E-128B	210,51	14,51	9,65	0,99635	194,8s
50E-32B	211,09	14,53	9,65	0,99634	425,4
50E-64B	213,33	14,60	9,68	0,99636	252,0s
50E-128B	209,09	14,46	9,62	0,99635	209,2s
60E-32B	209,69	14,48	9,65	0,99635	460,0s
60E-64B	209,65	14,48	9,63	0,99635	321,8s
60E-128B	210,94	14,52	9,64	0,99635	260,0s
70E-32B	209,85	14,49	9,63	0,99634	570,4s
70E-64B	210,76	14,52	9,64	0,99635	375,0s
70E-128B	209,69	14,48	9,65	0,99634	308,8s
100E-32B	209,60	14,48	9,63	0,99634	746,4s
100E-64B	213,17	14,60	9,68	0,99635	539,8s
100E-128B	209,73	14,48	9,63	0,99634	414,2s
150E-32B	209,92	14,49	9,63	0,99634	1155,6s
150E-64B	209,24	14,47	9,62	0,99635	826,2s
150E-128B	209,63	14,48	9,63	0,99634	688,6s
Average	210,20	14,50	9,64	0,99635	408,50s

Table 4.4: Performance of different GRU models trained with solar irradiance data

CHAPTER 5

Summary

5.1 Conclusion

The aim of this thesis was to compare the performance of different types of machine learning models with different parameters for forecasting the total photovoltaic power generation in Austria. To accomplish this, 420 models were trained, 105 for each type of machine learning model, to predict the photovoltaic power output of the next timestep. The results show that the best performing model on average over 5 models is the GRU model with 150 epochs and a batchsize of 64 (GRU-150E-64B) with an MSE of 8.96. It took 711 seconds to train this model which is above the average training time of all model types. If a short training time matters then other models will suit better. Looking at the filesize of the models the GRU models are smaller compared to the LSTM models by about 158KB. The reason why the models trained with additional solar data are so much worse is due to the fact that the solar irradiance data does not add meaningful value to the photovoltaic output in this setting, which can be seen in Figure 4.2. The insight that the additional solar irradiance data from Wien Hohe Warte can worsen the performance of the models so much is an important takeway of this thesis. Another goal was to setup a Web application on a Raspberry Pi which shows the predictions of the models in a live simulation, which was successfully accomplished. Additionally. the Web application is able to benchmark all models in a custom timerange [29].

5.2 Future Work

In a future extension, more different types of machine learning models like BiLSTM, CNN-LSTM or transformers can be benchmarked. Additionally to that more features could be useful to improve the models, especially if multistep-ahead forecasting is introduced. Also there could be more variety regarding hyperparameters. Additional parameters like temperature or humidity could be useful, if this data is available from multiple locations spread all over Austria to exclude local phenomena. Furthermore, the application can be extended by allowing the user to select which models they would like to view in the live view.

List of Figures

List of Tables

Bibliography

[1] "Energie in Österreich," pp. 10, 14. https://www.bmk.gv.at/dam/jcr:f0bdbaa4-59f2-4bde-9af9-e139f9568769/Energie_in_OE_2020_ua.pdf.

[2] "European network of transmission system operators for electricity," https://transparency.entsoe.eu/generation/r2/actualGenerationPerProductionType/show?name=&defaultValue=false&viewType=TABLE&areaType=CTY&atch=false&datepicker-day-offset-select-dv-date-from_input=D&dateTime.dateTime=03.09.2021+00:00|CET|DAYTIMERANGE&dateTime.endDateTime=03.09.2021+00:00|CET|DAYTIMERANGE&area.values=CTY|10YAT-APG------L!CTY|10YAT-APG------L&productionType.values=B16&dateTime.timezone=CET_CEST&dateTime.timezone_input=CET+(UTC+1)+/+CEST+(UTC+2).

[3] "Weather hörsching, upper austria history (03. august 2020)," https://www.wunderground.com/history/daily/at/h%C3%B6rsching/LOWL/date/2020-8-3.

[4] "Arad," https://www.zamg.ac.at/cms/de/klima/klimaforschung/datensaetze/arad.

[5] M. Olefs, D. J. Baumgartner, F. Obleitner, C. Bichler, U. Foelsche, H. Pietsch, H. E. Rieder, P. Weihs, F. Geyer, T. Haiden, *et al.*, "The austrian radiation monitoring network arad–best practice and added value," *Atmospheric Measurement Techniques*, vol. 9, no. 4, pp. 1513–1531, 2016.

[6] R. A. Rajagukguk, R. A. Ramadhan, and H.-J. Lee, "A review on deep learning models for forecasting time series data of solar irradiance and photovoltaic power," *Energies*, vol. 13, no. 24, p. 6623, 2020.

[7] R. DiPietro and G. D. Hager, "Deep learning: RNNs and LSTM," in *Handbook of medical image computing and computer assisted intervention*, pp. 503–519, Elsevier, 2020.

[8] "Electricity market transparency," https://www.entsoe.eu/data/transparency-platform/.

[9] J. Patterson and A. Gibson, *Deep learning: A practitioner's approach*. O'Reilly Media, Inc., 2017.

[10] G. Keilbar, *Modelling Systemic Risk using Neural Network Quantile Regression*. PhD thesis, 08 2018.

[11] M. A. Nielsen, *Neural networks and deep learning*, vol. 25. Determination press San Francisco, CA, 2015.

[12] Sanderson, "Backpropagation calculus," 2017. https://www.3blue1brown.com/lessons/backpropagation-calculus.

[13] C. Olah, "Understanding LSTM networks," 2015.

[14] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen Netzen," *Diploma Thesis, Technische Universität München*, vol. 91, no. 1, 1991.

[15] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[16] F. A. Gers, N. N. Schraudolph, and J. Schmidhuber, "Learning precise timing with LSTM recurrent networks," *Journal of machine learning research*, vol. 3, no. Aug, pp. 115–143, 2002.

[17] M. Rhanoui, M. Mikram, S. Yousfi, and S. Barzali, "A cnn-bilstm model for document-level sentiment analysis," *Machine Learning and Knowledge Extraction*, vol. 1, no. 3, pp. 832–847, 2019.

[18] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches," *arXiv preprint arXiv:1409.1259*, 2014.

[19] Y. Wang, W. Liao, and Y. Chang, "Gated recurrent unit network-based short-term photovoltaic forecasting," *Energies*, vol. 11, no. 8, p. 2163, 2018.

[20] "About keras," https://keras.io/about/.

[21] "An end-to-end open source machine learning platform," https://www.tensorflow.org/.

[22] "influxdata," https://www.influxdata.com/.

[23] "Plotly javascript open source graphing library," https://plotly.com/javascript/.

[24] "pandas," https://pandas.pydata.org/.

[25] "forever," https://www.npmjs.com/package/forever/.

[26] A. Mellit, A. M. Pavan, and V. Lughi, "Deep learning neural networks for short-term photovoltaic power forecasting," *Renewable Energy*, vol. 172, pp. 276–288, 2021.

[27] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[28] "statatlas," https://www.statistik.at/atlas/?mapid=them_energie_klimafonds.

[29] "Sourcecode implemented for thesis," https://github.com/schmalzer/bachelorthesis-public.