Using TensorFlow to predict future wind power output based on historic wind park data - Bremen Big Data Challenge 2018 -

Hannes Erbis (973664), Marty Schüller (979891)

Osnabrück University

Implementing ANNs with Tensorflow

Winter semester 2021/22

Project report

Abstract

To maintain a functional power supply chain it is crucial to forecast future power outputs of generators. In this project we retroactively partake in the Bremen Big Data Challenge 2018 in which competitors are ought to predict the energy output of a wind park given weather measurements like wind speed. Our approach utilized three different artificial neural network architectures: regular feedforward networks, LSTMs and GRUs. All models are realized with TensorFlow. Our results suggest that regular 'vanilla' feedforward networks consisting of four to five connected dense layers with 32 to 1042 neurons in each layer in combination with the use of an Autoencoder to reduce the dimensionality of the data yield the most promising results. It is, however, important to note that a properly optimized LSTM or GRU could produce an even better outcome. Furthermore, we find out that using an Autoencoder to reduce the dimensionality of the feature space can have a positive impact on training performance regarding the speed at which the loss is minimized during backpropagation. Our best model is able to forecast power output of the wind park for the second half of 2017 with a cumulative absolute percentage error of 0.07302 after training on weather data from 2016 to mid 2017.

Our source code is available at https://github.com/HannesOS/Predicting-future-wind-power-output-based-on-historic-wind-park-data

Contents

List of Figures

1 Introduction

Global warming is one of the most pressing challenges humans are facing in our modern world. Research suggests that rising global temperatures are emerging partly due to anthropogenic factors which cause a net increase of greenhouse gases in the atmosphere (Natural Resources Defence Council, 2021). The energy sector plays by far the biggest and most pivotal role in this, as more than 70 percent of global emissions arise from energy-related causes (Climate Watch, 2018). In this context it is not only important how much energy is being produced by humans but also how this energy is being generated. Currently, fossil fuels account for more than 80 percent of global energy production (BP p.l.c., 2020). Research shows, however, that fossil fuels are one of the main culprits of anthropogenic greenhouse gas emissions (European Comission, n.d.). It is therefore crucial to replace fossil fuels with 'cleaner' energy. As it turns out, the production of wind energy is connected with a significantly smaller environmental footprint and can therefore serve as an alternative to environmentally destructive methods (Guezuraga, Zauner and Pölz, 2012; Jones, Pejchar and Kiesecker, 2015). It should, however, be pointed out that the output of wind power generators heavily depends on the given weather conditions (Deutscher Wetterdienst, n.d. Lledó et al., 2019). Wind speed and wind direction, for example, can have a critical effect on the productiveness of power generators. Evidently, being able to estimate future energy supplies is paramount in maintaining a healthy energy supply chain and transition to sustainability. Therefore, it is necessary to be able to predict the future energy outputs of wind parks. This issue is the main motivation behind this project, in which we try to solve the problem proposed in the Bremen Big Data Challenge (2018). The Challenge ran from March 1 until March 31, 2018. In it, a data set consisting of weather measurements taken from inside a wind park in 15 minute intervals as well as the corresponding wind power output at the time of measurement is being presented. The goal of this challenge was to fit a model to this training data that is capable of anticipate future energy outputs given new wind park weather measurements. In our project we try to apply our TensorFlow (Abadi et al., 2015) skills to retroactively partake in this challenge. We compare different models and techniques to optimize our performance. Our ideas and findings are presented in this project report.

2 Data

The provided data of the Bremen Big Data Challenge 2018 consisted of measurements made inside an unspecified wind park and was stored inside csv files. Each row of the data set corresponded to one measurement. Measurements have been taken throughout 2016 and the first half of 2017 in 15 minute intervals. In total there were 19 columns, of which one corresponded to the specific time of the respective measurement and, in case of the training data, one to the wind power output at that time. Figure 2.1 illustrates example dynamics of the energy output over the course of a few days.

For this challenge we were given the wind speed and wind direction at heights of 48, 100 and 152 meters. Additionally, the probabilistic wind speeds at 100 meters are given in 10 to 90 percentiles in steps of 10. Furthermore, the available capacity of the wind park was also measured for each time step. Since the wind park did not actually measure the wind properties every 15 minutes but rather every hour, the missing values were gained through interpolation. Interpolated data rows are identified with the respective value in the 'interpolated' column. This value is either 1 or 0 depending on if the corresponding data is interpolated or not. As of March 2022, the data as well as additional information and explanations about it can be accessed at https://bbdc.csl.uni-bremen.de/index.php/2018h/23-aufgabenstellung-2018. Since the goal was to fit a model to this data, which is able to predict future power outputs, the second half of the 2017 data was being reserved as a testing set and did not include power outputs at the respective time steps. Now, the task for the competitors was to find

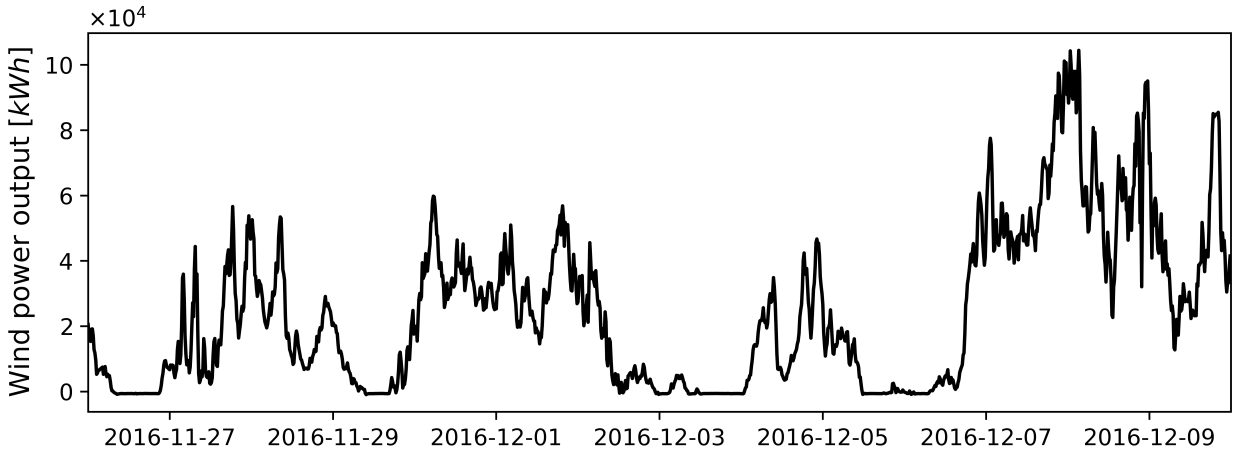

FIGURE 2.1: Wind power output of the unnamed wind park from November 27 until December 10, 2016. The time series can be obtained from the training data set used in the Bremen Big Data Challenge (2018).

	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	Output
F1	1	0.982	0.958	0.126	0.123	0.121	0.98	0.981	0.982	0.982	0.982	0.982	0.982	0.982	0.981	0	0.006	0.889
F2	0.982	1	0.992	0.093	0.094	0.097	0.998	0.999	1	1	1	1	1	0.999	0.998	0	0.057	0.889
F3	0.958	0.992	1	0.072	0.075	0.079	0.99	0.991	0.992	0.992	0.992	0.992	0.992	0.991	0.99	0	0.077	0.875
F4	0.126	0.093	0.072	1	0.948	0.918	0.091	0.091	0.092	0.093	0.093	0.094	0.094	0.095	0.096	-0.002	-0.029	0.117
F5	0.123	0.094	0.075	0.948	1	0.967	0.092	0.093	0.093	0.094	0.095	0.095	0.095	0.096	0.097	-0.001	-0.029	0.114
F6	0.121	0.097	0.079	0.918	0.967	1	0.095	0.096	0.096	0.097	0.097	0.097	0.098	0.098	0.099	-0.001	-0.026	0.112
F7	0.98	0.998	0.99	0.091	0.092	0.095	1	0.999	0.999	0.999	0.998	0.997	0.997	0.996	0.995	0	0.055	0.886
F8	0.981	0.999	0.991	0.091	0.093	0.096	0.999	1	1	0.999	0.999	0.999	0.998	0.997	0.996	0	0.055	0.887
F9	0.982	1	0.992	0.092	0.093	0.096	0.999	1	1	1	0.999	0.999	0.999	0.998	0.997	0	0.056	0.888
F10	0.982	1	0.992	0.093	0.094	0.097	0.999	0.999	1	1	1	1	0.999	0.999	0.998	0	0.056	0.888
F11	0.982	1	0.992	0.093	0.095	0.097	0.998	0.999	0.999	1	1	1	1	0.999	0.998	0	0.057	0.889
F12	0.982	1	0.992	0.094	0.095	0.097	0.997	0.999	0.999	1	1	1	1	0.999	0.999	0	0.057	0.889
F13	0.982	1	0.992	0.094	0.095	0.098	0.997	0.998	0.999	0.999	1	1	1	1	0.999	0	0.057	0.889
F14	0.982	0.999	0.991	0.095	0.096	0.098	0.996	0.997	0.998	0.999	0.999	0.999	1	1	1	0	0.057	0.889
F15	0.981	0.998	0.99	0.096	0.097	0.099	0.995	0.996	0.997	0.998	0.998	0.999	0.999	1	1	0	0.058	0.889
F16	0	0	0	-0.002	-0.001	-0.001	0	0	0	0	0	0	0	0	0	1	0	0
F17	0.006	0.057	0.077	-0.029	-0.029	-0.026	0.055	0.055	0.056	0.056	0.057	0.057	0.057	0.057	0.058	0	1	0.07
Output	0.889	0.889	0.875	0.117	0.114	0.112	0.886	0.887	0.888	0.888	0.889	0.889	0.889	0.889	0.889	0	0.07	1

FIGURE 2.2: Pearson correlations between the features and target in the Bremen Big Data Challenge 2018 training data. F1, ..., F17 stand for the features of the data set beginning with the wind speed at 48 meter height (F1) and ending with the installed capacity (F17). Output corresponds to the power output of the wind park. Correlations above a value of 0.9 are marked with a red color, as they indicate a very high correlation. We see that a significant proportion of the data is strongly correlated.

the missing power outputs for the last part of the time series. After the challenge concluded, the respective real energy output values for the test data have also been published, which we will use to assess the usefulness of our model architectures.

Before we build any models, it is sensible to first have a look at the data. We notice that lots of variables inside the data frame are very highly correlated (figure 2.2), indicating that we might not need all features to make useful predictions. Therefore, we can get rid of some of the variables. We notice that the features 'available capacity' and 'interpolated' have no positive impact on the training performance. In fact, including them in the training would result in a significantly slower process and a slightly worse model fit. Hence, we removed those features from our data. Moreover, it seems that, especially due to the strong correlations between many of the features, reducing the dimensionality of the data seems like a good idea. To reduce the data dimensionality of the model, we made use of an Autoencoder which produces an embedding layer which illustrates a lower-dimension version of the original input features (see chapter 3).

It is also important to note that the 'date' column will also be ignored in the training of our models as it not very relevant to the problem and can lead to confusing results. Now, we might think that normalizing our features to bring them into ranges of -1 and 1 would be a good step as it should make training easier and does not cause some features to have a disproportionately high impact on the training (Sola and Sevilla, 1997) but as it turns out, contrary to our beliefs, normalizing the data does not effect training positively but rather leads to a slightly worse performance in our case. Therefore, we did not normalize

the data, unless otherwise stated. Pre-processing and validation splitting of the data set was realized within the Keras 'model.fit()' function (see chapter 3). There we set a batch size of 128 and validation set which consists of 20 percent of the training data. Additionally, whenever we were using a Recurrent Neural Network (**RNN**) we needed to make sure that we sliced the data into time windows where the first n time steps of each window are given as features while the $(n+1)$th time step will be considered the target. Here we set the respective window size to 5 but this certainly could be optimized. To realize this we used *tensorflow.keras.preprocessing.sequence.TimeseriesGenerator* which is a generator and functions slightly different than a regular data set (TensorFlow, n.d.[b]). Whenever we did not deal with RNN-based networks, we also wanted to shuffle our data to get rid of potential hidden patterns. Since we want to make use of the 'model.fit()' function provided by Keras we do not need to specify a TensorFlow data set but rather just pass the features and targets as lists into the fitting function.

3 Methods

There were two main types of models we compared regarding their performance. The first architecture was a simple 'vanilla' feedforward neural network. It consists of multiple hidden dense layers and an output layer with one neuron. The hidden layers all have a 'ReLU' activation while the output neuron has a regular linear activation. We chose the aforementioned activation functions as they gave us the best results. The linear activation at the output neuron is due to the nature of the problem proposed in the Bremen Big Data Challenge, as we needed to predict real continuous numbers rather than classes or categories. This means that we were dealing with a regression problem rather than a classification task.

To have a comparison we also implemented two architectures of Recurrent Neural Networks (RNNs), namely Long short-term memory (LSTM) (Hochreiter and Schmidhuber, 1997) and Gated recurrent unit (GRU) (Cho et al., 2014). They function fundamentally different to a regular feedforward neural network in the sense that they consider the data from previous n time steps as features and the (n+1)th time step as a target. These architectures can be particularly useful when dealing with time series (TensorFlow, n.d.[c]). Here, we did not restrict ourselves to single LSTM or GRU cells but also wanted to explore what happens if we stack multiple of those networks on top of each other. This was a little bit difficult though, as the time required for learning would explode if multiple layers are being stacked. Additionally, we included an optional dropout layer for both the RNNs and the feedforward networks, in the hopes of increasing generalizability (Srivastava et al., 2014).

Furthermore, we experimented with dimensionality reduction by using an Autoencoder. The Autoencoder takes the feature data, which has a dimension of 17, and reduces it down to a dimension of 5 in our case. This can be done by training the Autoencoder and setting the embedding size to 5. The network then learns how to reduce the dimension of the data to said dimensionality (Rajan, 2021). Here we simply fixed the number of layers and the number of layer units in the Autoencoder to essentially arbitrary values. For further performance gains, it would be sensible to optimize these hyperparameters. To attain the lower dimensional data we simply propagated our features through the encoder, which is the first part of the Autoencoder. As a result we received the encoded, lower dimensional data. We anticipate this method to improve the generalizability of our model and maybe even boost training speed. All of our models are subclasses that inherit from *tf.keras.Model* (TensorFlow, n.d.[a]). The *call()* functions are always defined as a *tf.function* to enable graph mode and speed up the training process (Dash, 2020).

A considerable amount of the actual training was being automated by TensorFlow.

Since all of our models inherit from *tf.keras.Model*, we could use the *model.compile()* and *model.fit()* functions to train any of our architectures (TensorFlow, n.d.[a]). Keras then did all the backpropagation for us. As discussed in chapter 2, we reserved 20 percent of the entire training data for validation and define a batch size of 128. In our cases this gave us the most promising training performance. When using a feedforward network, the data would also be shuffled. As an optimizer, we chose Adam (Kingma and Ba, 2014) with a learning rate of 0.001. During the training we monitored the loss (mean squared error), the root mean squared error as well as the cumulative absolute percentage error (CAPE) defined as:

$$CAPE = \frac{\sum_{t=1}^{T} |Predicted_t - Actual_t|}{\sum_{t=1}^{T} Basis}, \tag{3.1}$$

where t stands for a time step, T for the maximum amount of time steps and $Basis$ for the installed capacity which is set to remain constant at a value of 122400kW (Bremen Big Data Challenge, 2018). All of the above metrics are being tracked for both the training and the validation sets.

To circumvent overfitting we passed a specific callback to the function. This callback was an early stopping mechanism which stopped the training after the CAPE of the validation data set stopped decreasing for 300 epochs in a row. After those 300 epochs, the network would roll-back to the weights where the model had the best validation CAPE. Therefore, we practically did not define a maximum amount of epochs, as the callback takes care of the stopping condition. After a network is fully trained we serialized it using *model.save()* and also saved a summary of the model and logs of the training to further understand our newly trained network. Furthermore, we evaluated each neural network and also give first predictions using the test data.

As established above, we did not fix all of the hyperparameters of our models. Instead, we wanted to be able to optimize them by trying out various combinations and tracking the respective performance. For each model we tried out different amounts of layers, different amount of neurons per layer as well as various dropout rates for the dropout layer. A dropout rate of 0 was also valid and resulted in no dropout layer being added to the network. For each architecture, the model with the lowest validation CAPE was chosen as the best network for the respective type. This left us with 6 different 'best' models, two for each architecture, i. e. one with using the raw competition data and one using the encoded lower dimensional data attained from the Autoencoder.

To keep track of the training and hyperparameter optimization process , the training progress was being tracked, printed out and serialized in a csv file. This included tracking the epoch, time spent per epoch, loss (mean squared error), root mean squared error and CAPE of both the training and validation data. This way we could understand the entire process. Furthermore, once a network has been fully trained, we saved a summary in which we serialized the number of epochs needed for training, how the model and its layers are

structured, what parameters have been used and the key metrics at the optimum.

Out of curiosity we also tried a classic non-ANN machine learning approach on this problem. For this we compared our findings with results we attained from training a Random Forest model on the same data. To simplify things, we used the Random Forest regressor and a 5-fold cross-validation function provided by scikit-learn (Pedregosa et al., 2011; scikit-learn, 2011a; scikit-learn, 2011b).

In order to speed up the training process, we utilized our GPUs with the help of CUDA (NVIDIA, Vingelmann and Fitzek, 2020) and cuDNN (NVIDIA, Vingelmann and Fitzek, 2021) to help with the backpropagation. Unfortunately this did not work out properly for the RNN based models as it would force us to use a 'tanh' activation function in the respective layer, which did not produce satisfying results. We therefore kept using a 'ReLU' activation and used our CPUs for the corresponding calculations instead.

4 Results

After training over 100 models and choosing the best one from each architecture based on the validation data which is split from the training data, we can now compare their effectiveness against each other. Table 4.1 gives an overview over the performance of each model on the test data. Out of all the models we experimented with, the regular feedforward network which makes use of the encoded with reduced dimensionality proved to be the strongest and most efficient one. With it we are able to forecast the energy output of the second half of 2017 with a cumulative absolute percentage error (CAPE) of 0.07302. In our case it took approximately 50 to 250 epochs to optimize the weights of the feedforward network. The training stopped after the validation CAPE did not decrease for 300 epochs in a row. After that, it would roll back to the optimal weights found regarding the validation CAPE. The best feedforward network we found makes use of the lower dimensional data acquired from the Autoencoder and consisted of four hidden layers with 32, 256, 512 and 1024 units in the layers respectively and did not include a dropout layer. Low amounts of layers resulted in remarkably worse forecasting power of the model on the test data. Figure 4.1 shows how the prediction of our best model compares to the actual wind park power output for a given time frame, and we notice that the feedforward network is able to understand the

TABLE 4.1: Best model predictions for each neural network architecture. We compared our best regular feedforward networks (FFNN), LSTMs and GRUs. The column 'encoded' illustrates whether or not the data has been propagated through an Autoencoder first, to reduce its dimensionality. The column 'encoded' denotes how many layers, excluding the output layer, have been used in the respective network. For the RNN based model it stands for the amount of LSTM or GRU units that have been stacked on top of each other. The 'Layer units' columns describes the amount of neurons in each of the layers respectively. 'Epochs' gives the average amount of epochs required for completely training the network. It is important to note that epochs take significantly longer to complete for the RNN based models compared to the feedforward networks. In the column 'Test CAPE' we present our cumulative absolute percentage error we achieved with this model.

Model	**Encoded**	**#Layers**	**Layer units**	**Epochs**	**Test CAPE**
FFNN	No	4	[32, 256, 512, 1024]	181	0.073373
FFNN	Yes	4	[32, 256, 512, 1024]	154	0.073026
LSTM	No	1	256	384	0.076966
LSTM	Yes	1	256	340	0.076900
GRU	No	1	256	399	0.075939
GRU	Yes	1	256	357	0.076132

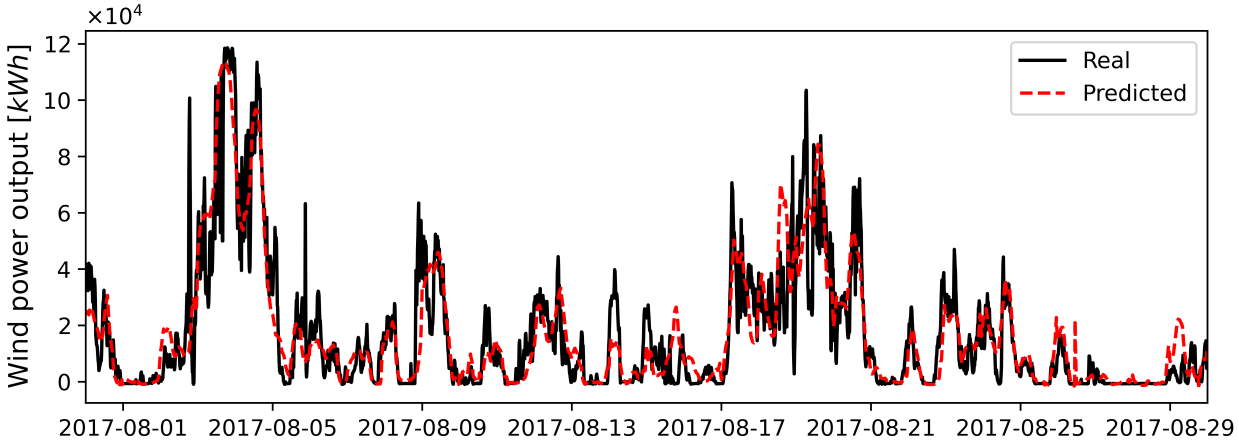

FIGURE 4.1: Real and predicted wind power output over the course of one month. For the predictions the standard feedforward network with autoencoded features has been used.

underlying dynamics between weather characteristics and wind power output very well. Unfortunately, the application of the other architectures was not as successful. For our best LSTM model we were able to achieve a CAPE of 0.0769 while our best GRU gives us a CAPE of approximately 0.0759 on the test data. These numbers are certainly higher than anticipated, as RNNs are known to perform well on time series data (TensorFlow, n.d.[c]). We need to emphasize that this underwhelming performance is most definitely a consequence of suboptimal model structures, hyperparameters or training specifications. This is mainly because the RNNs took a significantly longer time to train regarding both the number of epochs until the model hit its optimum as well as the time required per epoch compared to the vanilla feedforward network. This practically made it impossible for us to properly optimize those models. In chapter 5 we discuss this problem in more detail.

The Autoencoder we trained is able to reproduce the input feature data using an embedded version of itself which has been compressed to mere five dimensions with a mean squared error of approximately 0.11 for the training and 0.19 for the unseen testing data, which might not be optimal. For this project, however, it was out of our scope to also properly optimize the Autoencoder. In our case, the use of Autoencoders and therefore the dimension reduction of the feature space proved to be a useful tool to reduce training time needed to converge to the optimum. It also allowed us to consider a less complex network with higher generalizability which enabled us to attain our best fitting model.

Generally speaking, our findings consistently showed that the inclusion of a dropout layer into the different networks resulted in a worse prediction on the test data. So this measure to reduce overfitting and improve generalizability was not met with success for us, as it usually increased the test CAPE by 0.25 to 1 percent for the different dropout rates we considered, which we always keep between 0 and 0.5.

Overall, our neural networks performed better than the Random Forest regressor combined with 5-fold cross-validation we trained using scikit-learn. With the best regressor we achieved a test CAPE of 0.082847 but it is important to keep in mind that this could be

further improved with better settings. However, it was not one of our main priorities to optimize this model, and we thus hold onto these results.

5 Discussion

In our project we compared different kinds of artificial neural network architectures with different compositions of layers to solve the Bremen Big Data Challenge 2018. In total, we tested three different model architectures. Moreover, we experimented with dimensionality reduction of the data using an Autoencoder to increase training performance. For each architecture we optimized the hyperparameters, i. e. the amount of layers, neurons per layer and dropout rate based on the validation data which is split from the training data. Choosing the best model from each of our architectures left us with six models. One architecture, namely the regular feedforward network consisting of multiple connected dense layers, stood out to be significantly more effective and efficient than the others. The best feedforward network we could find consisted of four hidden dense layers followed by an output neuron and uses data with previously reduced dimensionality. Logically, the number of layers needed should be higher if we do not use the encoded data attained from the embedding layer of our Autoencoder, as the feature space would be bigger. For further analysis it would make sense to consider the effects of using data, which dimensionality has not been reduced, on a more complex neural network. With our best model we can predict the wind power output of the wind park for the second half of 2017 fairly well, with a CAPE of only 0.07302. If we had competed in the Bremen Big Data Challenge 2018, we would have reached rank 10 using our trained feedforward network. To our surprise, the LSTMs and GRUs did not work out as well as we hoped to. We think, however, that if we properly optimize our RNN based models, we could be able to achieve results that are comparable if not better than those we attained from the feedforward network. The reason we were struggling with this, was that in order to fully train our LSTMs and GRUs it takes an enormous amount of time. One main problem was that we could not utilize our GPUs during training, since the cuDNN does not support the RNN based models that satisfy our needs. GPU accelerated calculations are only available for LSTMs and GRUs if we use 'tanh' activation, which unfortunately does not work out for us, as the loss does not seem to converge to a satisfying minimum. Hence, we simply used a 'ReLU' function instead and put up with the enormous training time without GPU support. Training can be slightly sped up if the dimensionality of the feature space have been reduced by our Autoencoder but it is still very unpractical. Another critical point, where we could potentially boost the models' effectiveness is the Autoencoder itself. With respects to the short time, it was not a big priority of us to properly optimize the Autoencoder, meaning we just fixed all of the hyperparameters. We think that, with a better model configuration, we could have achieved

exceptionally better results when using the Autoencoder. Obviously, this statement also holds for the other models, as we think that a good hyperparameter choice is crucial if we want to end up with a useful and powerful artificial neural network.

Working on this project was very interesting and educational. Not only are we very motivated about learning everything about TensorFlow and artificial neural networks in general, but the specific topic we chose also seemed very meaningful to us and had a clear goal. The organization and communication inside our team was exceptional and definitely boosted our teamwork and helped us grow. During the project we have learned lots of new coding techniques and gained tons of knowledge that we think can be useful for future projects. As is usually the case with coding projects, there were a few moments of slight frustration here and there, but nothing that has stopped us from trying harder and coming up with new solutions.

This project marks the end of our participation in the course 'Implementing ANNs with Tensorflow' in the winter semester 2021/22 at Osnabrück University. All in all, we are really happy to have taken this course as it was really interesting to learn about all the different neural network architectures and their applications. Furthermore, we think the organization and structure of this course, for example the way how the homework was handled, was exceptional. We would like to thank all of the Tutors and everybody who worked on this course for this awesome learning experience.

A Software information

All technical work of this project was done in Python version 3.7 (Van Rossum and Drake Jr, 1995). The relevant packages and toolkits as well as their version numbers and applications are described in table A.1. Our source code is available at https://github.com/HannesOS/Predicting-future-wind-power-output-based-on-historic-wind-park-data.

Table A.1: Used Python packages and toolkits and their applications

Package name	**Version**	**Applications**	**Reference**
CUDA	11.2	GPU acceleration	(NVIDIA, Vingelmann and Fitzek, 2020)
cuDNN	8.1	GPU acceleration	(NVIDIA, Vingelmann and Fitzek, 2021)
Matplotlib	3.5.0	Data visualisation, creating figures	(Hunter, 2007)
Numpy	1.22.2	Mathematical operations, data structures	(Oliphant, 2006; Van Der Walt, Colbert and Varoquaux, 2011)
Pandas	1.4	Data reading and processing	(McKinney, 2010)
scikit-learn	1.0.2	Random forest	(Pedregosa et al., 2011)
Tensorflow	2.8	Artifical neural networks	(Abadi et al., 2015)

Bibliography

Abadi M. et al. (2015). *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. URL: https://www.tensorflow.org/.

BP p.l.c. (2020). *Statistical Review of World Energy*. URL: https://www.bp.com/en/global/corporate/energy-economics/statistical-review-of-world-energy.html (visited on 03/22/2022).

Bremen Big Data Challenge (2018). *Bremen Big Data Challenge 2018*. URL: https://bbdc.csl.uni-bremen.de/index.php/2018 (visited on 03/22/2022).

Cho K. et al. (2014). *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*. DOI: 10.48550/ARXIV.1406.1078. URL: https://arxiv.org/abs/1406.1078.

Climate Watch (2018). *Historical GHG Emissions*. URL: https://www.climatewatchdata.org/ghg-emissions?breakBy=sector&chartType=percentage&end_year=2018&source=CAIT&start_year=1990 (visited on 03/22/2022).

Dash S. (2020). *How to use tf.function to speed up Python code in Tensorflow*. machine learning plus. URL: https://www.machinelearningplus.com/deep-learning/how-use-tf-function-to-speed-up-python-code-tensorflow/ (visited on 03/24/2022).

Deutscher Wetterdienst (n.d.). *Weather forecasts for renewable energy - a challenge*. URL: https://www.dwd.de/EN/research/weatherforecasting/num_modelling/07_weather_forecasts_renewable_energy/weather_forecasts_renewable_energy_node.html (visited on 03/23/2022).

European Comission (n.d.). *Causes of climate change*. URL: https://ec.europa.eu/clima/climate-change/causes-climate-change_en (visited on 03/22/2022).

Guezuraga B., Zauner R. and Pölz W. (2012). "Life cycle assessment of two different 2 MW class wind turbines". In: *Renewable Energy* 37.1, pp. 37–44. ISSN: 0960-1481. DOI: https://doi.org/10.1016/j.renene.2011.05.008.

Hochreiter S. and Schmidhuber J. (Dec. 1997). "Long Short-term Memory". In: *Neural computation* 9, pp. 1735–80. DOI: 10.1162/neco.1997.9.8.1735.

Hunter J. D. (2007). "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3, pp. 90–95. DOI: 10.1109/MCSE.2007.55.

Jones N. F., Pejchar L. and Kiesecker J. M. (Jan. 2015). "The Energy Footprint: How Oil, Natural Gas, and Wind Energy Affect Land for Biodiversity and the Flow of Ecosystem Services". In: *BioScience* 65.3, pp. 290–301. ISSN: 0006-3568. DOI: 10.1093/biosci/biu224.

Kingma D. P. and Ba J. (2014). *Adam: A Method for Stochastic Optimization*. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

Lledó L. et al. (2019). "Seasonal forecasts of wind power generation". In: *Renewable Energy* 143, pp. 91–100. ISSN: 0960-1481. DOI: https://doi.org/10.1016/j.renene.2019.04.135.

McKinney W. (2010). "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference*. Ed. by Stéfan van der Walt and Jarrod Millman, pp. 56 –61. DOI: 10.25080/Majora-92bf1922-00a.

Natural Resources Defence Council (2021). *Global Warming 101*. URL: https://www.nrdc.org/stories/global-warming-101 (visited on 03/22/2022).

NVIDIA, Vingelmann P. and Fitzek F. H. (2020). *CUDA Toolkit. Version 11.2*. URL: https://developer.nvidia.com/cuda-toolkit.

— (2021). *cuDNN. Version 8.1*. URL: https://developer.nvidia.com/cudnn.

Oliphant T. E. (2006). *A guide to NumPy*. Vol. 1. Trelgol Publishing USA.

Pedregosa F. et al. (2011). "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12, pp. 2825–2830.

Rajan S. (2021). *Dimensionality Reduction using AutoEncoders in Python*. URL: https://www.analyticsvidhya.com/blog/2021/06/dimensionality-reduction-using-autoencoders-in-python/#:~:text=AutoEncoder%20is%20an%20unsupervised%20Artificial,representation%20of%20the%20input%20data. (visited on 03/24/2022).

scikit-learn (2011a). "sklearn.ensemble.RandomForestClassifier". In: URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html (visited on 03/27/2022).

— (2011b). "sklearn.model_selection.cross_val_score". In: URL: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_val_score.html (visited on 03/27/2022).

Sola J. and Sevilla J. (1997). "Importance of input data normalization for the application of neural networks to complex industrial problems". In: *IEEE Transactions on Nuclear Science* 44.3, pp. 1464–1468. DOI: 10.1109/23.589532.

Srivastava N. et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15.56, pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

TensorFlow (n.d.[a]). *tf.keras.Model*. URL: https://www.tensorflow.org/api_docs/python/tf/keras/Model (visited on 03/24/2022).

— (n.d.[b]). *tf.keras.preprocessing.sequence.TimeseriesGenerator*. URL: https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/sequence/TimeseriesGenerator (visited on 03/24/2022).

TensorFlow (n.d.[c]). *Time series forecasting*. URL: https://www.tensorflow.org/tutorials/structured_data/time_series#recurrent_neural_network (visited on 03/24/2022).

Van Der Walt S., Colbert S. C. and Varoquaux G. (2011). "The NumPy array: a structure for efficient numerical computation". In: *Computing in Science & Engineering* 13.2, p. 22.

Van Rossum G. and Drake Jr F. L. (1995). *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam.