IN5410 - Assignment 2: Machine Learning for Wind Energy Forecasting

Group 1 - davidgek, emiliehf, peterhp and siljeben

May 2024

Abstract

Wind energy forecasting is the process of using relevant power and weather data to predict future output of wind turbines. It is critical to make use of wind energy, balance the grid and ensure correct production and pricing of electricity.

In this report we explored the predictability of wind power generation using various machine learning methods, with an emphasis of relationships between wind speed, wind direction and power output.

First, we considered wind speed and wind generation. The Neural network method performed the best with an RMSE of 0.2130, with the SVR method just behind with an RMSE of 0.2137. kNN and LR performed the worst with an RMSE of 0.2163 and an RMSE of 0.2164 respectively. In the second case the wind direction was added, which slightly improved the MLR prediction, yielding an RMSE of 0.2149. Directly using the zonal and meridional wind speed components as features further improves the model with an RMSE of 0.2085, surpassing all previous methods that only utilized wind speed, but not wind direction. Hence, the wind direction provides significant additional information, if the model is able to interpret it.

In Task 3 we utilized time-series forecasting with a sliding window approach. The SVR model, with a window size of 83, achieved an RMSE of 0.1282, whereas LR, with a window size of 100, performed slightly better with a lower RMSE of 0.1222. The ANN, with a window size of 100 had an RMSE of 0.1344. The RNN model stood out with a window size of 100 and the lowest RMSE of 0.1214, highlighting its proficiency in capturing the changes and variations in the time series.

Lastly, we manually did the calculations of forward and back propagation to ensure that this method, used in neural networks, actually converges to the best solution. The errors were also calculated by a program for about 350 iterations.

Contents

1 Introduction **3**

2 Data description **4**
- 2.1 Preprocessing 4
  - 2.1.1 Sliding window technique 5

3 Background and libraries **5**
- 3.1 Forecasting and machine learning 5
- 3.2 Methods 6
  - 3.2.1 Linear Regression (LR) and Multiple Linear Regression (MLR) 6
  - 3.2.2 Support Vector Regression (SVR) 7
  - 3.2.3 k-nearest neighbors (kNN) 8
  - 3.2.4 Artificial Neural Networks (ANN) 9
  - 3.2.5 Recurrent Neural Networks (RNN) 10
- 3.3 Calculating error: Root Mean Square Error (RMSE) 10
- 3.4 Libraries 11
  - 3.4.1 sklearn 11
  - 3.4.2 pytorch 11

4 Implementation/algorithm **11**
- 4.1 Task 1 11
- 4.2 Task 2 11
- 4.3 Task 3 12
- 4.4 Task 4 12

5 Results and discussion **12**
- 5.1 Task 1: One feature and different methods 12
- 5.2 Task 2: Two features and MLR 16
- 5.3 Task 3: No features and different methods 17
- 5.4 Task 4: Specifics of propagation 23
- 5.5 Conclusions and reflections 26

1 Introduction

Wind power forecasts are critical in several socio-economically relevant decisions and calculations like electricity market price, power grid stability and energy load balance. Therefore it is vital to accurately predict the power produced by wind farms. The impact of a wrong prediction can, in a worst-case scenario, become an economic liability for the producers, TSOs and power consumers.

In the event of underestimating wind power output, energy wastage and revenue loss afflict producers. Moreover, such inaccuracies can escalate to catastrophic proportions, potentially triggering blackouts when supply fails to meet demand. The ripple effect of these inaccuracies can disrupt entire energy systems, leading to widespread economic and social consequences.

However, the main problem of wind is the intermittency. Changes in wind speed can happen quickly (within minutes), and the local variation of wind speed poses another significant obstacle. The wind turbines causes wind-wakes, which makes the airflow turbulent and even harder to predict, resulting in even more uncertainty in wind turbine output. In addition, the wind power curves, which describe the theoretical output of a turbine given wind speed, are non-linear.

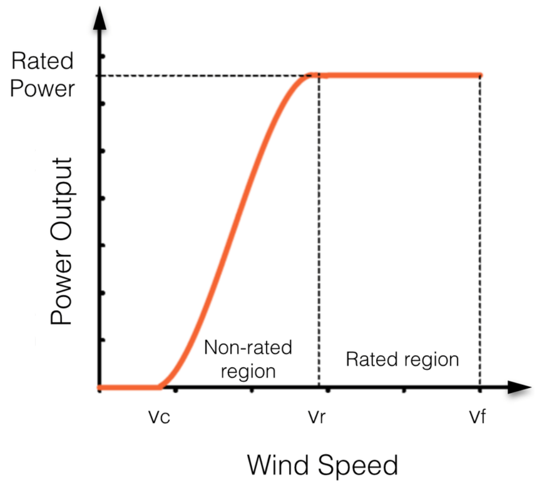


Figure 1: Typical wind turbine power curve: the turbine begins to operate at the cut-in speed V_c, then the power output increases with wind speed following a cubic curve until wind speed reaches the rated speed, where the turbine begins to operate at its rated power [1]

.

In order to perform forecasts, one needs something to predict from. That is, predictions come from some input features and their values. The amount of possible features for forecasting wind power makes the optimization hard to solve, these features are everything from the wind speed and direction to less obvious ones like temperature and barometric pressure. Not to mention that wind speed and direction again is impacted by the geography of the surrounding area, the time of year, and several other things.

In this assignment historic wind power data from two years is used to train machine learning models which forecast power production for the forthcoming month. The variables are limited, with only wind speed and wind direction as external inputs. The models used are linear regression, multiple linear regression, support vector regression, k-nearest neighbors, artificial (feedforward) neural networks and recurrent neural networks.

2 Data description

The data files in this project contain necessary information to learn, predict and validate machine learning methods for wind energy forecasting. The power data is from a wind farm in Australia, and the weather information is from the European Centre for Medium-range Weather Forecasts (ECMWF). The power data is normalized by the nominal wind farm capacity to preserve anonymity.

The dataset **TrainData.csv** consists of weather and power measurements for a time series. The measurements are hourly, from 1am on 01.01.2012 until 0 am on the 01.12.2013, resulting in 16 080 measurements in total. For each time-step, we have the following measurements which are used in the assignment:

- TIMESTAMP: Provides date and hour in the format yyyymmdd HH:MM, for instance the first timestamp is 20120101 01:00
- POWER: normalized power output from the wind farm (in range 0 to 1)
- U10: Wind speed in m/s in west-east projection at 10m above ground
- V10: Wind speed in m/s in south-north projection at 10m above ground
- WS10: Total wind speed in m/s at 10m above ground

The dataset also includes similar wind speed measurements 100m above the ground, but those are not to be used for this assignment.

The data found in **WeatherForecastInput.csv** consists of the same weather features as the previous file. The time period is the whole month of November 2013, i.e. the month after the end date of **TrainData.csv**. The data in **WeatherForecastInput.csv** will be used as input to the models for prediction of the wind power generation that month. The last dataset **Solution.csv** contains the true power measurements of the month of November 2013. This will be used for validation of the models.

2.1 Preprocessing

As the data is already relatively well behaved, we skip normalization/scaling of the data. In particular, the output is already normalized between 0 and 1. For the wind speed inputs, we consider their range from roughly -10 to +10 acceptable, as none of the considered methods are particularly sensitive to the scale of the data. Neural networks do in general perform better with normalized data, but can handle small to medium differences in scale [2].

We further evaluated converting the timestamps to decimal years, in order to have a numeric value for the date, and predict seasonal patterns in the data. With more data, collected over a longer time-frame, this strategy may be effective. However, since our data is comprised of less than two years, and therefore does not contain enough seasonal information, we did not pursue the idea further.

To avoid correlated input features, we calculate the wind direction. As we delve into the wind forecast data provided in **TrainData.csv**, it includes measurements of both the zonal component (U10) and the meridional component (V10). These two measurements are also referred to as longitude and latitude. Longitude aligns parallel to the x-axis, while latitude aligns parallel to the y-axis [3]. The coordinates serve as inputs to a trigonometric function, specifically the arctan2 function from the numpy library [4], to compute the wind direction in terms of polar coordinates.

```python
def components_to_angle(df):
    df["A10"] = np.arctan2(df["U10"], df["V10"])
    df["A100"] = np.arctan2(df["U100"], df["V100"])
    df.drop(["U10", "V10", "U100", "V100"], axis=1)
    return df
```

2.1.1 Sliding window technique

In Task 3, we need to predict future power output from just the past power output measurements. Since there may be some variance in the power output, we want to consider multiple past measurements to predict the next one. Hence, we transform the data to represent these multiple past measurements as input data, and the corresponding next power measurement as output. This is achieved using a sliding window technique to gather the last m measurements to predict the next one, if we choose a window size of m. For instance, the prediction for power measurement p_7 with a window size of 4 would utilize the past true data measurements p_3, p_4, p_5 and p_6. We will revisit the implementation of the sliding window technique in section 5.3.

3 Background and libraries

3.1 Forecasting and machine learning

Forecasting is the process of predicting future values based on historical and current data. Accurate forecasts can lead to better decision-making and strategic planning.

Machine learning is the process of using a machine to find a relation between some input values $x_1, x_2, \cdots x_n$ (features) and the value y (label) that we are interested in. Machine learning uses previous data, called training data, to find/learn correlations and patterns in the data. In practice this means that the machine creates some internal structure of the data, given an approach. The approaches vary, and the ones used in this assignment is discussed in 3.2. All methods used in this assignment goes in the category of *supervised machine learning*. This means that under the training we provide the true values for a combination of inputs.

In particular, the goal of machine learning is to learn the relation not only for the training data, but instead learn it in such a way that the model is able to predict the correct output for new, unseen data. That is the reason we differentiate between training and test data, and use test data to evaluate the model. In cases where the machine has learned the relationship between the input and output perfectly for the training data, but fails to accurately predict new, unseen data, the model is *overfitted.* When using machine learning models, overfitting should be avoided.

3.2 Methods

3.2.1 Linear Regression (LR) and Multiple Linear Regression (MLR)

LR is a method used to find a linear relationship between the input variable and the output variable. The concept of the method is to find the linear relationship that minimizes the error between the true values y of the training data and the predicted value y_{LR}.

In mathematical terms the process is as follows: Given n observations $(x_i, y_i); i = 1, 2, \dots, n$, with $x_i \in \mathbb{R}, y_i \in \mathbb{R}$ we want to find a line y/y_{LR} (see equation 1) that minimizes the error. The error is defined as the sum of the *residuals* squares, where residuals are defined as:

$$e_i = y_i - (\beta_1 x_i + \beta_0)$$

Hence, what we are trying to do is to find β_0 and β_1 that minimize

$$\min \sum_{i=1}^{n} e_i^2 = \min \sum_{i=1}^{n} (y_i - (\beta_1 x + \beta_0))^2$$

If there are several input variables, MLR is used, and the linear relationship is described in equation 2.

$$y_{\text{LR}} = \beta_0 + \beta_1 X \tag{1}$$

$$y_{\text{MLR}} = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_n X_m \tag{2}$$

When y is found with the training data, or more specifically the weights $\beta_0, \beta_1, \dots, \beta_m$, this function remains, and is used for future input values. This means that when a new data point $x = (x_1, x_2, \dots, x_m)$ is presented to the model, the output is:

$$y_{\text{pred}} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ...\beta_m x_m$$

A MLR problem can also be formulated more succinctly by collecting the training data in a matrix $X = \begin{bmatrix} \mathbf{1} & X_1^T & X_2^T & ... & X_m^T \end{bmatrix}^T$, where $\mathbf{1}$ is an m-dimensional column vector consisting of m ones, representing the bias value, and $X_{i,j}$ representing the j-th element of the i-th training sample. Then, collecting all weights as $\beta = \begin{bmatrix} \beta_1 & \beta_2 & ... & \beta_n \end{bmatrix}^T$ and all true outputs as $\mathbf{y} = \begin{bmatrix} y_1 & y_2 & ... & y_n \end{bmatrix}^T$, we formulate the least-squares problem in equation 3 as:

$$w^* = \arg\min_{w} ||Xw - y||^2 \tag{3}$$

By calculating the derivative and setting it to zero, it can be shown that the above formulation is equivalent to the following system of normal equations:

$$X^T X w^* = X^T y \tag{4}$$

The general result for any MLR problem is therefore given by the solution to this system of linear normal equations.

3.2.2 Support Vector Regression (SVR)

Support Vector Regression is a machine learning method that involves calculating the error between points to find a line y so that the fewest errors are found outside of the error-threshold of the line. SVR work with different *kernels*, which defines how the line y looks. Non-linear kernels can capture more complex relationships in the data.

To explain the concept of SVR, let us use the linear kernel. In this way, we build on our understanding on linear regression and can easily formulate and illustrate how SVR finds the optimal line.
For SVR (with linear kernel) it finds a hyperline $y = wx + b$, but ignores the errors which are less than ϵ from the line y. ϵ is a pre-defined parameter and represents the acceptable error. The ϵ-tube is the tube around the hyperline within which the error is ignored. Note that this error is vertically from the line, and not perpendicular (See Figure 2).

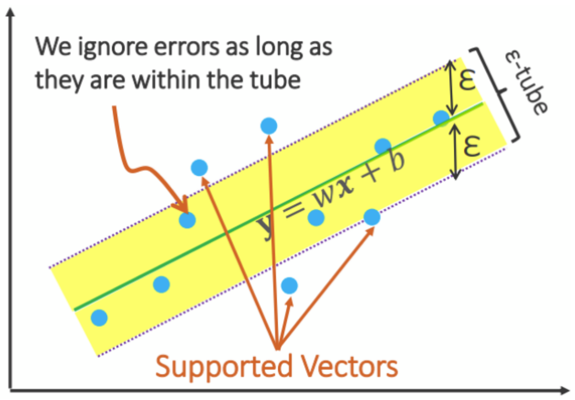


Figure 2: Illustration of SVR. Picture taken from lecture in IN5410, 21.03.2024

The L points that fall outside this tube, have an error $\xi_{il}(l = 1, \ldots, L)$ which is expressed as the distance to the tube.

$$\xi_{il} = |y_{il} - (wx_{il} + b)| - \epsilon$$

However, for points inside the tube we have that $|y_i - (wx_i + b)| - \epsilon < 0$. So to generalize we have that

$$\xi_i = \max(|y_i - (wx_i + b)| - \epsilon, 0)$$

for all points.

Before going into the minimization formulation of this method, let us have a look at some characteristics.
Recall that the formula for the perpendicular distance of two parallel lines are

$$d = \frac{|C_1 - C_2|}{\sqrt{A^2 + B^2}}$$

which gives that the distance between the two lines of the epsilon tube is

$$d = \frac{2\epsilon}{\sqrt{w^2 + 1}}$$

This means that to maximize the distance between the two lines, and thus cover more points, we need to minimize the w^2. In practice this means having the line as flat as possible. However, the errors of the points which the tube cannot cover, must also be considered and minimized.
The minimization goal ends up as

$$\min_{w,b} \frac{1}{2}\|w\|^2 + C\sum_{i=1}^{n} \xi_i$$

subject to

$$|y_i - (wx_i + b)| \leq \epsilon + \xi_i$$

where C is a tolerance-parameter. A small C gives a higher toleration of errors (can have more points outside the tube), and a higher C penalizes errors more.
The difference between LR and SVR is that SVR is less prone to overfitting. That is because with small $|w|$, the function is less sensitive to small changes in the input data.
After the line is found with the training data for SVR, the same procedure follows as for LR: the function y remains, and is used for future input values. This means that when a new data point x is presented to the model, the output is

$$y_{\text{pred}} = wx + b$$

3.2.3 k-nearest neighbors (kNN)

The k-nearest neighbors method is a method which is commonly used in classification problems, but is also effective in regression problems. When used in regression the kNN algorithm calculates the average value of the k nearest neighbors.

Given some training data $(x_i, y_i); i = 1, 2, \ldots, n$ and a new point x, the algorithm works in the following way: First it starts by calculating the distance $D(x, x_i)$ from x to all points $x_i (i = 1, 2, \ldots, n)$ in the training set. The distance is calculated using a mathematical metric, which can either be the Euclidean distance, Manhattan distance, or more generally the Minkowski distance or any other function d that fulfills the axioms for a metric space (M, d) where M is a set on which d are working. In this assignment we are using the Euclidean distance. Which is defined as

$$D(x_i, x) = \sqrt{\sum_{j=1}^{m}(x_{ij} - x_j)^2}$$

where m is the number of features (input variables). When only one feature is present, like in Task 1: One feature and different methods, the distance is just the absolute value between the points, i.e. $D(x_i, x) = |x - x_i|$,

After calculating the distances from x to all points in the training data, it takes the k closest points $(x_{i1}, x_{i2}, \ldots, x_{ik})$ in training data and their values $(y_{i1}, y_{i2}, \ldots, y_{ik})$. The output for x is calculated by the average of the values.

$$y_{\text{pred}} = \sum_{j=1}^{k} y_{ij}/k$$

3.2.4 Artificial Neural Networks (ANN)

A artificial neural network is a method inspired by the way the human brain (biological neural networks) works to understand patterns in the data. It uses a structure and calculations that mimics how the neurons in the human brain communicate to process information.

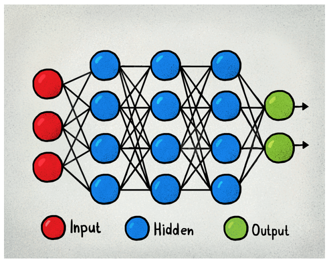


Figure 3: The structure of a neural net. Picture taken from [5]

The most simple artificial neural networks (ANN) goes under other names as well: neural network (NN) and Feed-forward neural network (FFNN). The structure of this basic network is shown in Figure 3. Each dot represent a neuron. The vertical lines are called layers. All artificial neural networks consist of one input layer (red), one or more hidden layers (blue) and one output layer (green). All the neurons in layer k connects to all the neurons in layer $k + 1$.

In each neuron a weighted linear sum of the outputs of the last layer is calculated. Then this sum goes through an activation function before the output is sent to the next layer. A non-linear activation function introduces non-linear relationships between the input and output data. There are many different types of activation functions, but in this assignment the sigmoid function is used. This is defined by

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

The popularity of use for this function comes from the fact that it is continuous, differentiable and that the derivative function can be easily expressed. The derivative is $f'(x) = f(x)(1 - f(x))$.

Training a neural network mainly includes two processes: forward propagation and back propagation. In forward propagation we use the neural net's weights and activation function(s) to get an answer when given an input. In backward propagation we are adjusting the weights by calculating the error between the output from the forward propagation and the error from the backward propagation. By re-using the calculated gradients for all weights in previous network layers, it is possible to compute the gradient extremely efficiently. Calculating a gradient is then no more expensive than doing the forward pass.

3.2.5 Recurrent Neural Networks (RNN)

Recurrent neural networks differ from regular ANNs by being able to have an internal state which persists from one forward pass to the next. This enables the RNN to save important information in its connections, and significantly improves performance for time-series forecasting tasks, where outputs typically depend on previous states of the system.

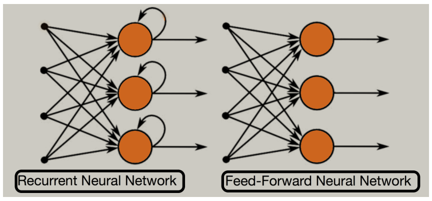


Figure 4: RNN and FFNN/ANN. Picture from [6]

Further example applications include NLP and Computer Vision, particularly in the video domain.

3.3 Calculating error: Root Mean Square Error (RMSE)

To evaluate the models, we need to compare the predicted outputs and the true values. So for each data point $(x_i, y_i)i = 1, \ldots, N$ in the test dataset, the model makes a prediction $y_{\text{pred},i} = \hat{y}_i$. To measure the error of the model, we calculate the total error that the model makes for these points. This calculation can be done several ways, like using R^2, MAE, $RMSE$ or others. In this assignment the chosen metric for calculating error is *Root Mean Square Error*, or RMSE. This is expressed as

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}$$

A low RMSE means that the models predictions are close to the true values for (most of) the inputs. A high RMSE can have several reason. The first being that the model used is wrong for the relationship of the data. For example using linear regression when the relationship is non-linear. Another reason is that there are inadequate amounts of training data for the the model to find a relationship. Lastly, a low RMSE can come from overfitting, as described in Forecasting and machine learning.

3.4 Libraries

3.4.1 sklearn

For Assignment 2, several machine learning models will be utilized through the sklearn library, as this is an open-source machine learning and data modeling library for Python. The library is designed to interoperate well with other Python libraries, such as NumPy for array vectorization, and Pandas dataframes [7]. The library provides numerous supervised and unsupervised learning algorithms, and for this assignment we applied the algorithms KNeighborsRegressor, LinearRegression, and SVR.

3.4.2 pytorch

For implementing the neural network models, namely the ANN and RNN, we use the pytorch framework [8]. It provides a dynamic computation graph for automatic differentiation and optimization, requiring us only to specify the network architecture and the training loop. It is, according to paperswithcode.com, currently most popular framework in Deep Learning research.

4 Implementation/algorithm

4.1 Task 1

In this task we used four different machine learning methods: LR, SVR, kNN and ANN/NN. These were used to find the relationship between wind power generation and wind speed, at 10m above ground level. This means that we only used the columns WS10 and POWER from the datasets.

All methods except the neural network are from the library sklearn. We used SVR with a non-linear kernel (more specifically 'rbf'/default), which can capture more complex relationships in the data. The neural network model is implemented using the pytorch framework.

We started by extracting the columns of interest from both the training data and the test data. Then we trained all models with the training data, where POWER was the label and WS10 was the (only) feature. Then we made predictions with all models for the test data. These predictions can be found in the following csv-files: **ForecastTemplate1-LR.csv**, **ForecastTemplate1-SVR.csv**, **ForecastTemplate1-kNN.csv**, **ForecastTemplate1-NN.csv**. We compared the predictions to the real values, with plots. Furthermore, the RMSE was computed for each method to evaluate and compare the different methods.

4.2 Task 2

In Task 2, we used the numpy arctan2 function to calculate an angle in radians between $-\pi$ and $+\pi$ as additional feature. We used the zonal component as first input, and the meridional component as second input. This results in an angle of zero being east, an angle of $\frac{\pi}{2}$ being north, $-\frac{\pi}{2}$ being south, and $\pm\pi$ representing west. This angle input is then used together with the wind speed as features for a MLR model, that is otherwise trained and evaluated similarly to the models in Task 1. The predicted forecast can be found in the csv-file: **ForecastTemplate2.csv**.

4.3 Task 3

Task 3 considers the wind power prediction task without providing any additional weather input features, but instead only relying on past wind power values. In order to have more context than just a single previous value, we apply the sliding window technique introduced in section 2.1.1 to provide a window of several previous inputs as features for every prediction.

This sliding window input is then used to train a feed-forward artificial neural network, and a recursive neural network. Both of those are implemented in pytorch, with the ANN using the same architecture we used in Task 1. Additionally, we will also apply it for the LR and SVR models.
The predictions generated by these models are stored in the following CSV files: **ForecastTemplate3-ANN.csv**, **ForecastTemplate3-RNN.csv**, **ForecastTemplate3-LR.csv**, and **ForecastTemplate3-SVR.csv**.

4.4 Task 4

For Task 4, we manually implement the backpropagation algorithm for the specific network provided in the assignment. NumPy is only used to randomly initialize the weights.

First, the forward pass is calculated, and the error evaluated. Then, we calculate all intermediate partial derivatives along the edges of the network. This enables us to prevent calculating the same partial derivative multiple times, for instance once for every weight. These intermediate partial derivatives are then used to calculate the derivatives of the error w.r.t. the individual weights. Finally, the weights are updated using gradient descent, and the process repeats until convergence.

5 Results and discussion

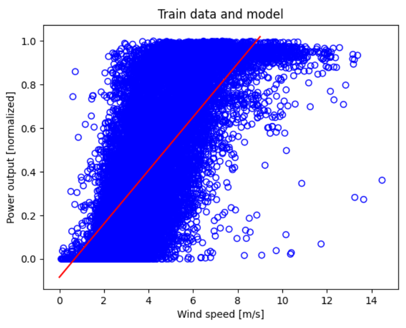
5.1 Task 1: One feature and different methods

The RMSE for all four methods are shown in table 1. All models perform pretty well with RMSE ≈ 0.21. The neural network performed the best with RMSE = 0.2130, with SVR just behind at RMSE = 0.2137. kNN and LR holds the last places with RMSE = 0.2163 and RMSE = 0.2164, respectively. We can see that the results are similar for all methods, only deviating with less than 0.01 amongst them.

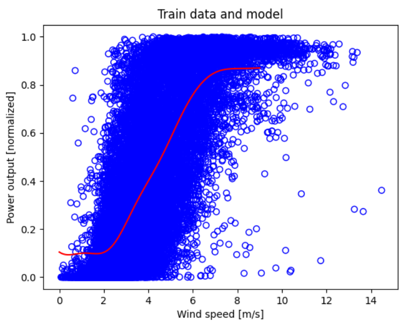
ML technique	**RMSE**
SVR	0.2137
LR	0.2164
kNN	0.2163
NN	0.2130

Table 1: Comparison of RMSE for Task 1

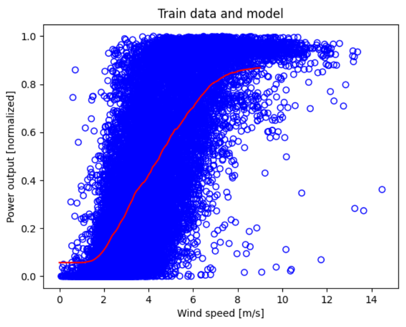
Figure 5 shows the training data and the fitted lines for each method. The linear regression model performs well in the middle range, but bad for small or large values of wind speed. For large wind speeds it even predicts a power output above 1, which is generally not possible. Furthermore, we



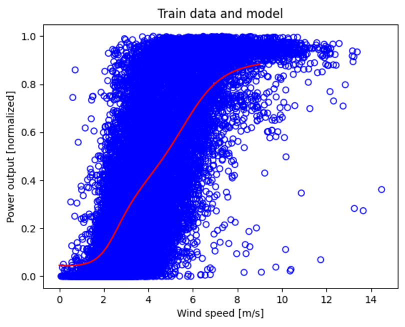
(a) LR



(b) SVR



(c) kNN



(d) NN

Figure 5: Comparing training data pairs and fitted models

see that kNN, SVR and the NN follow a non-linear curve, such as the power curve for wind turbines in Figure 1. For SVR this is due to the use of a non-linear kernel and for kNN, this is due to the fact that it relies on closest points. The NN, due to its nonlinearity and large amount of parameters, is also able to fit the nonlinear power curve very well.

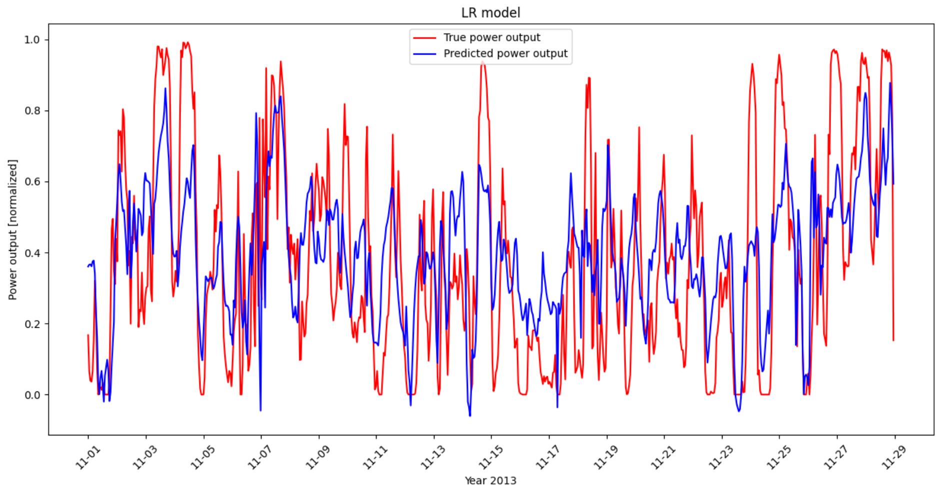


Figure 6: LR

Figure 6 shows that the predicted outputs from a LR model is a bad match to the actual outputs. It can mostly follow the shape, but fails to predict extreme values. The predictions are mostly under the true values graph when power output is more than 0.5, and mostly over when power output is less than 0.5.

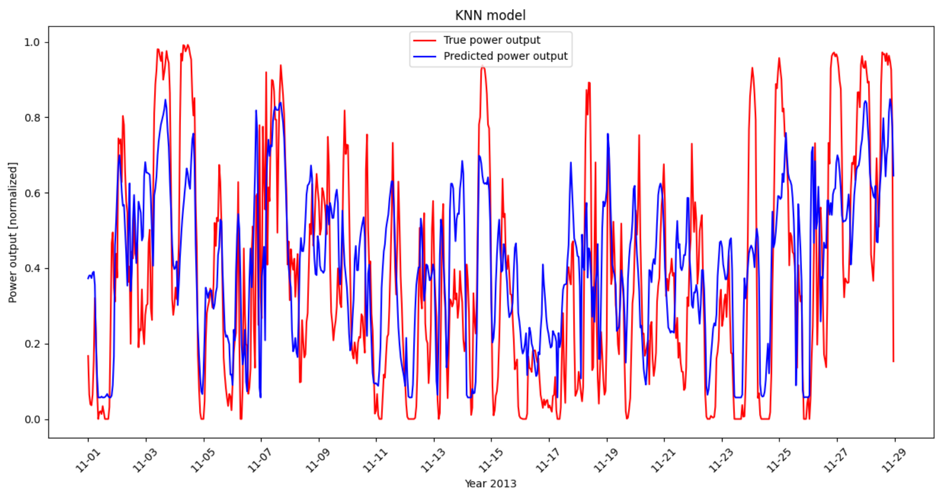


Figure 7: kNN

Figure 7 shows the disagreement between the predicted from kNN and the actual power output. The graph of kNN is comparable to LR, but kNN get some more detail in place, and have more smoother edges on the oscillations. However, the problem with accurately predicting extreme points remain. Note that we used $k = 1634$, this value was decided by bruteforcing the absolute best value (under 2000) for this specific data. Such a large k can prevent the kNN from fitting finer, high-frequency patterns on other/new data, as higher values for k limit the models expressiveness.

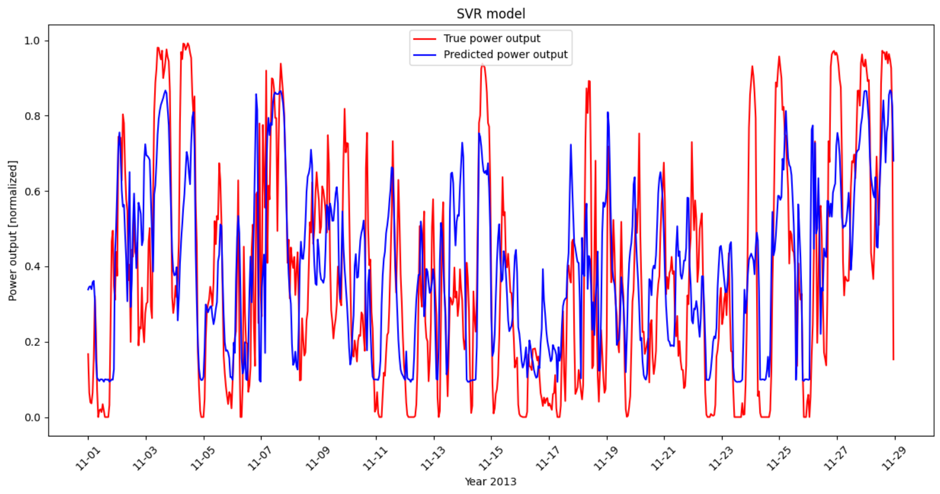


Figure 8: SVR

Figure 8 shows that SVR lie closer to the actual output than the previous two method, which is the reason for the lower RMSE. It is both better at predicting the oscillations and the extreme points.

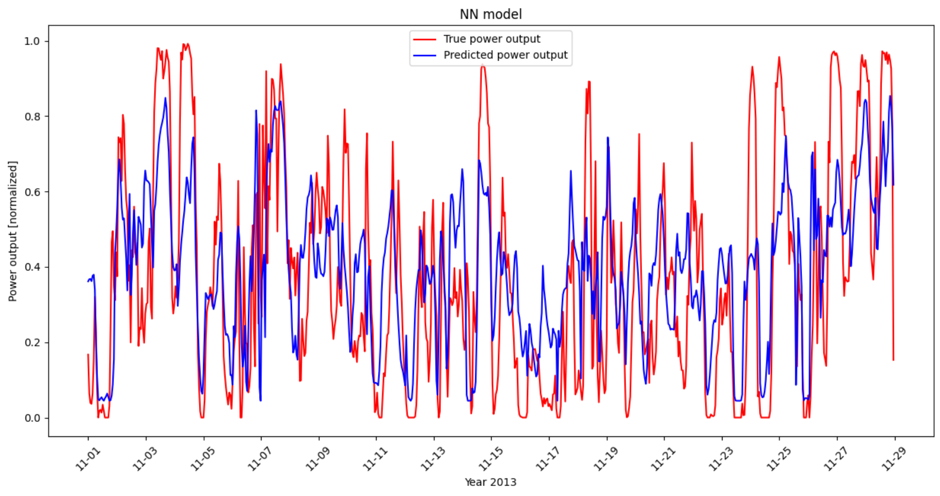


Figure 9: NN

Figure 9 shows that the NN predictions are extremely close to those of the SVR method. While it is difficult to see in the prediction plot, the predictions of the NN are even slightly better than that of the SVR, evidenced by its lower RMSE value.

5.2 Task 2: Two features and MLR

As for Task 2, prior discussions have highlighted the potential for improved model performance by incorporating a second feature: wind direction.

Upon constructing the MLR model, we assembled a multidimensional array comprising both wind speed and calculated wind direction. This array was utilized by the model to forecast power generation for the period of 11.2013. To assess the model's efficacy, we computed the RMSE between the predicted and actual power outputs. Table 2 displays the RMSE values for both the LR model from Task 1 and the MLR model implemented in this task.

ML technique	**RMSE**
LR	0.2164
MLR	0.2149

Table 2: Comparison of RMSE for LR and MLR

It becomes evident that the inclusion of wind direction may not have enhanced the model's performance notably. When getting the coefficients of the MLR, the coefficient for the angle is -0.00399 while the coefficient for the wind speed is 0.12350. This demonstrates the insignificance of the angle, specially since we know that the angle/direction will only vary between -π and π. Also note that since multiple *linear* regression is used, the output will be completely different for π and $-\pi$, even though they represent the same angle.

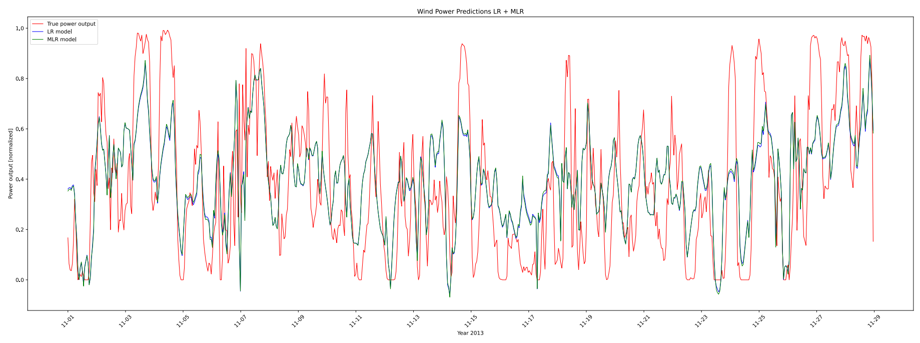


Figure 10: LR-MLR

Figure 10 show three curves; the true power generation in red, and predictions from the LR in blue and from MLR in green. The blue and green line are almost identical, which shows that the model was not able to extract much additional information out of the wind direction.

We also tested MLR with using the zonal and meridional components, resulting in RMSE of 0.2085, which is lower. However note that the U and V values implicitly contain information about the average wind speed as well, since $WS = \sqrt{U^2 + V^2}$. Thus there is correlation between the features. Specifically, when using U and V, the coefficient for the wind speed increases to 0.12808, compared to 0.12350 when using the angle. We did pursue this further, and therefore the plot is not included for this.

5.3 Task 3: No features and different methods

The primary objective for Task 3 was for the model to find relationships or patterns and forecast the subsequent power output accurately, where the POWER and TIMESTAMP variables were organized as time-series data. Employing the sliding window technique, we transformed the data to enable each model to predict the forthcoming output based on preceding data.

We devised various helper functions to facilitate the encoding of input and output data for each model using the sliding window method. The encoding process remained consistent across all ML models.
The function `get_sliding_window_input_output` generates input-output pairs from the training data, with each window serving as an input-output pair. The size of the window determines the number of data points considered in each instance.

```python
def get_sliding_window_input_output(data: np.ndarray, window_size: int):
    X = []
    for i in range(len(data) - window_size):
        X.append(data[i : i + window_size])
    return np.array(X), data[window_size:]
```

Additionally, the function `get_sliding_window_input_test_data` prepare input data for evaluating the performance of each model. In time series forecasting, ensuring continuity between the last data point of the training set and the first data point of the test set is crucial. This continuity provides the model with the necessary historical context for making accurate predictions on unseen data.

```
def get_sliding_window_input_test_data(
    test: np.ndarray, train: np.ndarray, window_size: int
):
    X = []
    for i in range(window_size):
        x = np.concatenate([train[-window_size + i :], test[:i]])
        X.append(x)
    for i in range(len(test) - window_size):
        X.append(test[i : i + window_size])
    return np.array(X)
```

Let's delve into the functionality of the functions. For illustrative purposes, we will apply a window size of 3 for the first and last rows of time-series from the **TrainData.csv**, **Solution.csv**, and the results from each sliding window function.

Timestamp	**Power**
20120101 1:00	0.2736781568
20120101 2:00	0.0867959455
20120101 3:00	0.0068114015
20120101 4:00	0.0186459868
20120101 5:00	0.0348118328
20120101 6:00	0.0219168973
20120101 7:00	0.01823263
⋮	⋮
20131031 19:00	0.1425092556
20131031 20:00	0.1012544481
20131031 21:00	0.1050465476
20131031 22:00	0.1450792567
20131031 23:00	0.1809334675
20131101 0:00	0.236826498

Table 3: The first and last rows of time-series data in TrainData.csv

Table 3 displays the power data used for training the wind power generation forecasting model. Each row represents a timestamp and its corresponding power value.

Input (X)	Output (y)
{0.2736781568 ,0.0867959455 ,0.0068114015}	0.0186459868
{0.0867959455 ,0.0068114015 ,0.0186459868}	0.0348118328
{0.0068114015 ,0.0186459868 ,0.0348118328}	0.0219168973
{0.0186459868 ,0.0348118328 ,0.0219168973}	0.01823263
⋮	⋮
{0.1425092556 ,0.1012544481 ,0.1050465476}	0.1450792567
{0.1012544481 ,0.1050465476 ,0.1450792567}	0.1809334675
{0.1050465476 ,0.1450792567 ,0.1809334675}	0.236826498

Table 4: Formatted Training Data Using Sliding Window Size as 3

Table 4 presents the training data formatted using a sliding window approach. Each row represents a set of input-output pairs, where the input consists of a sequence of power values, and the output is the subsequent power value to be predicted. The table is constructed using the dataset from Table 3.

Timestamp	Power
20131101 1:00	0.1672145142
20131101 2:00	0.0639984185
20131101 3:00	0.0390352611
20131101 4:00	0.0362316236
20131101 5:00	0.0648880342

Table 5: The first rows of time-series data in Solution.csv

Table 5 shows the solution data, which represents the actual power values observed at specific timestamps. Each row corresponds to a timestamp and its associated power value.

Input (X)	Output (y)
{0.1450792567 ,0.1809334675 ,0.236826498 }	0.1672145142
{0.1809334675 ,0.236826498 ,0.1672145142}	0.0639984185
{0.236826498 ,0.1672145142 ,0.0639984185}	0.0390352611
{0.1672145142 ,0.0639984185 ,0.0390352611}	0.0362316236
{0.0639984185 ,0.0390352611 ,0.0362316236}	0.0648880342
{0.0390352611 ,0.0362316236 ,0.0648880342}	0.15467632

Table 6: Formatted Test Data Using Sliding Window Size as 3

Table 6 demonstrates the test data formatted with a sliding window technique. Each row represents a set of input-output pairs, where the input consists of a sequence of power values, and the output is the subsequent power value to be predicted. Please observe that the first row of input values corresponds to the last 3 output values in Table 4. This exemplifies the maintaining of continuity between the last data point of the training set and the first data point of the test set.

We then evaluated each model using RMSE to assess performance and determine an optimal window size for each model that is represented in Table 7.

ML technique	**Window size**	**RMSE**
SVR	83	0.1282092712257549
LR	100	0.12223004615190537
ANN	100	0.13438464645758574
RNN	100	0.1214215778680508

Table 7: Comparison of RMSE

Initially, each model was assigned a window size of 140. Through observation, and engaging in *trial and error*, we manually determined a window size that resulted in the lowest RMSE within that range. We noted some distinct behaviors for the SVR model, particularly noting that the window size was set to a lower value.

Further, we did not expect the linear regression method to perform better than some of the other methods with higher expressiveness, like SVR or ANNs. A possible explanation is overfitting, since the more expressive a model is, the easier it can fit random noise in the training data, instead of the underlying pattern. The RNN does not suffer from this problem, instead it seems to be able to fit the underlying pattern more closely than the LR, without overfitting. This is likely due to the inductive bias of its memory state, enabling more natural predictions of time series compared to a regular ANN with sliding window inputs.

The wind power predictions are plotted as time-series data, divided into two graphs. Figure 11 displays the wind power forecast results using LR and SVR, while Figure 12 illustrates the power forecasts using ANN and RNN. Each graph includes a red curve representing the true wind energy measurements.

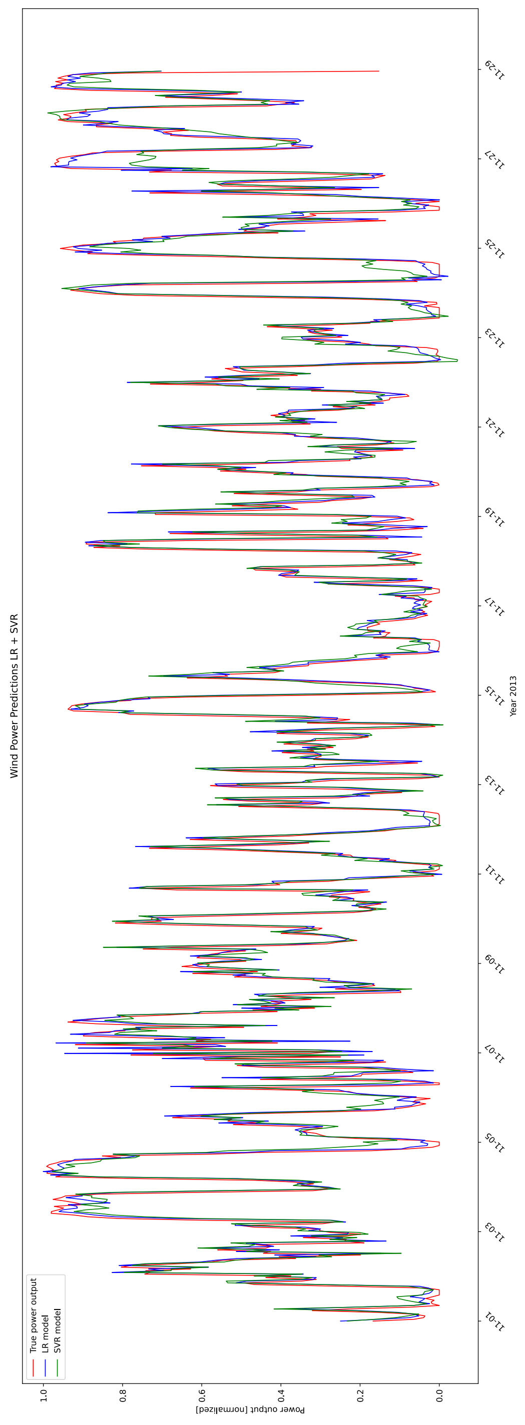


Figure 11: LR-SVG

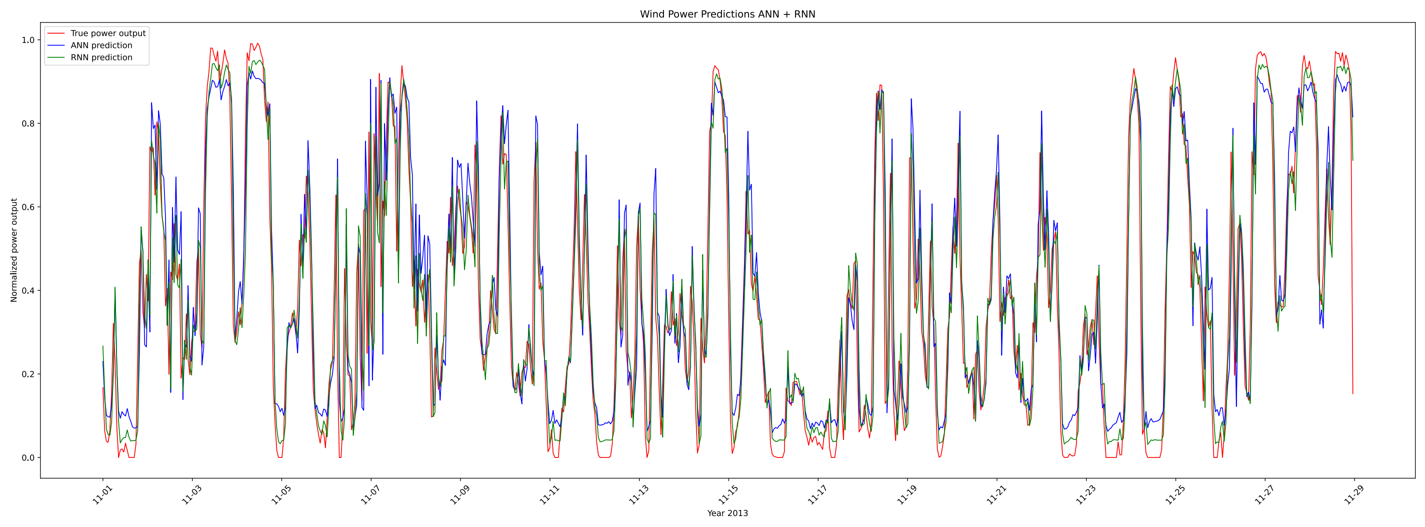


Figure 12: ANN-RNN

5.4 Task 4: Specifics of propagation

This question addresses the details of a neural network. More specifically, the calculations of forward and back propagation for a specific neural net.
Consider the following neural network consisting of two inputs, one hidden layer with two hidden neurons and one output neuron:

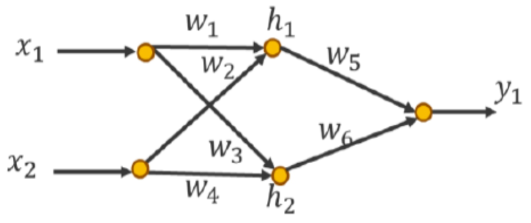


Figure 13: Neural net with two inputs, one hidden layer and one output

Note that biases are not included in this neural network. It is assumed that the activation function is the sigmoid function for all nodes. The first step is to initialize the $w_i (i = 1, 2, \ldots, 6)$, and the result using NumPy and seed = 23 gives

Weights
w1: 0.5172978838465893
w2: 0.9469626038148141
w3: 0.7654597593969069
w4: 0.2823958439671127
w5: 0.2210453632616525
w6: 0.6862220852374669

Table 8: Initial weights for Task 4

Now taking input values $x_1 = 0.04$ and $x_2 = 0.2$, gives:

$$sum_{h_1} = w_1 x_1 + w_3 x_2 = 0.21008443611682642$$
$$sum_{h_2} = w_2 x_1 + w_4 x_2 = 0.08709755916929882$$

which gives

$$out_{h_1} = \text{sigmoid}(sum_{h_1}) = 0.5523287874841797$$
$$out_{h_2} = \text{sigmoid}(sum_{h_2}) = 0.5217606352105785$$

so that

$$sum_{y_1} = w_5 \cdot out_{h_1} + w_6 \cdot out_{h_2} = 0.4801333885583371$$
$$out_{y_1} = \text{sigmoid}(sum_{y_1}) = 0.6177793720401447$$

The true value of y_1 is 0.5, which is not equal to the neural network's output. To see the error we use the error function. The error function is defined generally as $E_{total} = \frac{1}{2}\sum(y_i - out_{y_i})^2$ In this case with only one output it can be expressed as

$$E = \frac{1}{2}(y_1 - out_{y_1})^2$$

So the error before making any adjustments is 0.006935990239085405.
Now, we want to make this model better, which is done using the *backpropagation* method. The goal of this method is to optimize the weights of the neural network so that it learns the relationship between input and output, and is able to take arbitrary inputs and correctly map it to the outputs. What is done is to update the weights in the network so that the output of the network and the true value is closer, or in other words, minimize the error.
The error function is used to see the partial dependence for each weight. In other words it tells something about how much a change in w_i affects the total error. If the partial derivative is positive, this means that an increase in the weight would result in an increase in error (and the weight is updated to be smaller to minimize the error). The opposite is true for a partial derivative which is negative.

For the following calculations recall that for $f(x) = \text{sigmoid}(x)$, we have $f'(x) = f(x)(1 - f(x))$. Using the chain rule we get that

$$\begin{aligned}
\frac{\partial E}{\partial w_1} &= \frac{\partial E}{\partial out_{y_1}} \cdot \frac{\partial out_{y_1}}{\partial sum_{y_1}} \cdot \frac{\partial sum_{y_1}}{\partial out_{h_1}} \cdot \frac{\partial out_{h_1}}{\partial sum_{h_1}} \cdot \frac{\partial sum_{h_1}}{\partial w_1} \\
&= (y_1 - out_{y_1})(-1) \cdot out_{y_1}(1 - out_{y_1}) \cdot w_5 \cdot out_{h_1}(1 - out_{h_1}) \cdot x_1 \\
&= 6.08015998829407e - 05 \\
\frac{\partial E}{\partial w_2} &= \frac{\partial E}{\partial out_{y_1}} \cdot \frac{\partial out_{y_1}}{\partial sum_{y_1}} \cdot \frac{\partial sum_{y_1}}{\partial out_{h_2}} \cdot \frac{\partial out_{h_2}}{\partial sum_{h_2}} \cdot \frac{\partial sum_{h_2}}{\partial w_2} \\
&= (y_1 - out_{y_1})(-1) \cdot out_{y_1}(1 - out_{y_1}) \cdot w_6 \cdot out_{h_2}(1 - out_{h_2}) \cdot x_1 \\
&= 0.0003040079994147035 \\
\frac{\partial E}{\partial w_3} &= \frac{\partial E}{\partial out_{y_1}} \cdot \frac{\partial out_{y_1}}{\partial sum_{y_1}} \cdot \frac{\partial sum_{y_1}}{\partial out_{h_1}} \cdot \frac{\partial out_{h_1}}{\partial sum_{h_1}} \cdot \frac{\partial sum_{h_1}}{\partial w_3} \\
&= (y_1 - out_{y_1})(-1) \cdot out_{y_1}(1 - out_{y_1}) \cdot w_5 \cdot out_{h_1}(1 - out_{h_1}) \cdot x_2 \\
&= 0.0001904838115353583 \\
\frac{\partial E}{\partial w_4} &= \frac{\partial E}{\partial out_{y_1}} \cdot \frac{\partial out_{y_1}}{\partial sum_{y_1}} \cdot \frac{\partial sum_{y_1}}{\partial out_{h_2}} \cdot \frac{\partial out_{h_2}}{\partial sum_{h_2}} \cdot \frac{\partial sum_{h_2}}{\partial w_4} \\
&= (y_1 - out_{y_1})(-1) \cdot out_{y_1}(1 - out_{y_1}) \cdot w_6 \cdot out_{h_2}(1 - out_{h_2}) \cdot x_2 \\
&= 0.0009524190576767915 \\
\frac{\partial E}{\partial w_5} &= \frac{\partial E}{\partial out_{y_1}} \cdot \frac{\partial out_{y_1}}{\partial sum_{y_1}} \cdot \frac{\partial sum_{y_1}}{\partial w_5} \\
&= (y_1 - out_{y_1})(-1) \cdot out_{y_1}(1 - out_{y_1}) \cdot out_{h_1} \\
&= 0.015360821354885446 \\
\frac{\partial E}{\partial w_6} &= \frac{\partial E}{\partial out_{y_1}} \cdot \frac{\partial out_{y_1}}{\partial sum_{y_1}} \cdot \frac{\partial sum_{y_1}}{\partial w_6} \\
&= (y_1 - out_{y_1})(-1) \cdot out_{y_1}(1 - out_{y_1}) \cdot out_{h_2} \\
&= 0.014510690170591213
\end{aligned}$$

Updating the weights are done according to the following formula

$$w_i^+ = w_i - \alpha \frac{\partial E}{\partial w_i}$$

Where w_i^+ are the new (updated) weights, w_i is the current weight, $\frac{\partial E}{\partial w_i}$ is the partial derivative and α is the learning rate. The learning rate says something about the intervals or step length that is used in each updating steps. This is not avoid taking too big or too small steps, which can result in jumping over a minimum or taking too long to find a minimum, respectively. α is set to 0.4 in this task. As the formula shows, weights are updated in the opposite direction of their partial derivative, as expressed earlier. The following table sums up the updated weights after the first step.

Updated weights
w1: 0.5172735632066361
w2: 0.9468410006150483
w3: 0.7653835658722927
w4: 0.2820148763440420
w5: 0.2149010347196984
w6: 0.6804178091692303

Table 9: Updated weights after first round of backpropagation

Performing the same calculations with the updated weights results in a lower error of 0.006757994362320645. Note that, for the calculations, the intermediary gradients are not recomputed for every weight, i.e $\frac{\partial E}{\partial out_{y_1}}$ is only computed once, saved, and re-used for all partial weight calculations in that round. Now we want to run the training process. As seen in the first iteration, the error decreases in the iteration. The setup of the error function and the updates of the weights will give decreasing error for each iteration. But as seen, the error decreases only by a small amount, which can result in propagation which last for a long time. Specifically, the last steps to get to the optimal weights will result in small difference in the error. Therefore, we want stop the training when the difference between the error in a round and the error in the previous round is lower than threshold. In this assignment it was set to 1e-8. This gave the following error for the first 10 rounds and the last round before training stopped:

Iteration	Error
1	0.006935990239085405
2	0.006757994362320645
3	0.006584065883545012
4	0.006414136694760013
5	0.00624813859377414
6	0.006086003356807131
7	0.005927662807821905
8	0.005773048884604081
9	0.005622093701617335
10	0.0054747296096708436
⋮	⋮
347	3.365267594710107e-07

Table 10: Error for the first 10 iterations, and the last round before training stopped

5.5 Conclusions and reflections

In Task 1: One feature and different methods, we saw that the methods performed similar, despite their varying complexity. This shows that the relationship between wind speed and wind power generation, is not a very complex one. The linear regression model is the simplest and still did well, which could be caused by a near-linear relationship between wind speed and wind power generation.

In Task 2: Two features and MLR, we introduced the wind direction in addition to the wind speed, but MLR did not perform much better than LR where only wind speed was used. We noticed that the choice of feature space can have a significant impact on model performance - The periodic angle representation did not perform as well with MLR as the continuous wind speed component features did. Maybe a different, more expressive model would not be as sensitive to the chosen feature space, but it is expected that a linear model has difficulties with a periodic angle representation.

Task 3: No features and different methods illustrated that the prediction of power output by looking at several previous power outputs yielded the best results. In this task all methods had an RMSE ≈ 0.125, which is about 0.1 points better than the methods used in Task 1 and Task 2. This shows the importance of small details to get the correct power output. The wind can be turbulent when hitting the wind turbines or the turbine can have problems at times. These types of problems were likely present the time slot just before, and thus using previous close-in-time power output gives relevant information for the next time slot. The window size that performed best were about 100, which corresponds to a little over 4 days.

Futhermore, there are also other techniques that could have been used. First, the wind turbines are usually high above ground and here the wind speeds are more stable. If we had used the values for 100m above ground, the results might have been more accurate. Also, the formula for a wind turbine's power output is

$$P = 0.5 C_p \rho \pi r^2 v^3$$

where C_p is the coefficient of performance (efficiency factor, in percent), ρ is air density (in kg/m3), r is the blade length (in meters) and v is the wind speed (in meters per second). This shows a strong relationship with wind speed, but also that air density plays a role. Air density is dependent on several factors, including temperature, altitude and humidity.

Further work includes investigating the results for wind speed 100m above ground, introducing the other variables present in the data. Also, it would be interesting to combine the sliding window power input and the wind speed, to see if it is possible to get the best of both methods. This can be done by having two different ML methods, and combining their answer in a sophisticated way, resulting in a small Mixture of Experts model. An additional piece of information we did not utilize are the timestamps: They may enable improved predictions by accounting for seasonality in the data, e.g. higher wind speeds and low temperatures in October / November.

References

[1] Elise Dupont, Rembrandt Koppelaar, and Hervé Jeanmart. “Global available wind energy with physical and energy return on investment constraints”. In: *Applied Energy* 209 (Oct. 2017). DOI: `10.1016/j.apenergy.2017.09.085`.

[2] Dalwinder Singh and Birmohan Singh. “Investigating the impact of data normalization on classification performance”. In: *Applied Soft Computing* 97 (Dec. 2020), p. 105524. ISSN: 1568-4946. DOI: `10.1016/j.asoc.2019.105524`. URL: `https://www.sciencedirect.com/science/article/pii/S1568494619302947` (visited on 04/25/2024).

[3] *Calculate wind speed and direction from u,v in wind data*. en-US. URL: `https://sgichuki.github.io/Atmo/` (visited on 05/09/2024).

[4] *numpy.arctan2 — NumPy v1.26 Manual*. URL: `https://numpy.org/doc/stable/reference/generated/numpy.arctan2.html#` (visited on 05/09/2024).

[5] *7 Types of Artificial Neural Networks for Natural Language Processing*. en-US. URL: `https://www.kdnuggets.com/7-types-of-artificial-neural-networks-for-natural-language-processing` (visited on 05/09/2024).

[6] akshay Tondak. *Recurrent Neural Networks — RNN Complete Overview 2023*. en-US. July 2022. URL: `https://k21academy.com/datascience-blog/machine-learning/recurrent-neural-networks/` (visited on 05/09/2024).

[7] *What is Sklearn? — Domino Data Lab*. URL: `https://domino.ai/data-science-dictionary/sklearn` (visited on 05/09/2024).

[8] *PyTorch*. en. URL: `https://pytorch.org/` (visited on 05/09/2024).