IT UNIVERSITY OF COPENHAGEN

Evaluating the scalability of graph neural networks on regional forecasting of danish wind power

Frederik Bechmann Faarup

Master Thesis
Computer Science
Frederik Bechmann Faarup

December 1, 2024
Supervisor: Maria Sinziiana Astefanoaei

CONTENTS

1 INTRODUCTION

Wind power is swiftly becoming a pivotal element in the global shift toward sustainable energy sources. In countries like Denmark, wind energy already accounts for over 40% of total energy production, and this percentage continues to climb. However, the reliance of wind power on weather conditions introduces significant variability in energy production and reliability. Achieving accurate wind power forecasts is essential for smoothing grid operations, planning maintenance schedules, and minimizing the need for additional balancing energy sources.

Advancements in deep learning, such as Recurrent Neural Networks, Transformers and recently, Graph Neural Networks (GNNs), have unlocked new possibilities for wind power forecasting. The latter effectively utilizes the spatial relationships among wind turbines. While GNNs have demonstrated promising results in literature on a wind farm-level, their scalability and computational demands raise important questions as the number of turbines—and thus the complexity of the network—increases.

This project investigates how GNNs scale when applied to a growing number of wind turbines - will the spatial information make forecasting more accurate? What is the scalibility of the GNNs?

To answer, this thesis[1] compares a set of GNN models against simple baseline models and a baseline Gated Recurrent Unit (GRU) model to evaluate performance differences. The data utilized is based on wind turbines in North Jutland, Denmark, and it includes the historical power output of each turbine, historically measured weather data at each turbine's location, and lastly, pre-forecasted weather data for those same locations.

To systematically assess scalability, we select three subsets of data representing an increasing number of turbines by expanding the geographical area from a single starting point. This approach allows us to observe the models' performance and computational requirements as the size of the networks grow.

Additionally, we estimate the carbon emissions produced during hyperparameter tuning, training, and evaluation of the models, using an existing tool from literature. By doing so, we can evaluate how the environmental impact scales alongside computational complexity, providing a more holistic understanding of the trade-offs involved.

In summary, we aim to answer the following questions: How do GNN models scale in terms of forecasting performance, computational complexity and carbon emissions when applied to an expanding network of wind turbines in the real world? What is the trade-off between model complexity and forecasting accuracy in the context of wind power prediction?

[1] For replication of study, the code is made available on Github.

2 BACKGROUND

2.1 SHORT INTRODUCTION TO WIND POWER FORECASTING

This project revolves around wind power forecasting (WPF), a regression task with the goal of estimating the future electricity produced by wind turbines. Various techniques have been used to produce the forecasts, like physical models, time series regression, deep learning and hybrids. Here, we focus on deep learning models, particularly, those of the type graph neural networks, but also recurrent neural networks.

WPF can be classified into different tasks based on the time scale, forecasting object, types of forecasts and the aforementioned forecasting techniques (Wang et al. (2021)). The time scale can be divided into very short-term (up to 30 minutes), short-term (30 minutes to 6 hours), medium-term (6 hours to 1 day) and long-term (above 1 day). The forecasting object is typically a single wind turbine or a wind farm, and the types of forecasts are deterministic or probabilistic models.

This project focuses on medium-term forecasts (24 hours). To get a sense of scalability, we focus on area-bound, turbine-wise forecasting of several individual turbines within different area sizes. Lastly, the models are deterministic, as they produce raw forecasting numbers rather than probability distributions.

Beside the historical power outputs of the turbines, it is typical for models to utilize historical weather conditions and, if available, forecasts of future weather conditions. At each time step, this gives two two-dimensional inputs for each turbine - or two three-dimensional inputs for a model with multiple turbines. The model F converts the inputs to a two-dimensional forecast, i.e.:

$$Y^{T_{out} \times N} = F(X_1^{T_{in} \times N \times M_1}, X_2^{T_{out} \times N \times M_2}) \tag{1}$$

, where Y is the forecasted output of the model, X_1 is the historical input and X_2 is the forecasted weather input. T_{in} and T_{out} are the time steps of the historical input and forecasted output, respectively, and N is the number of turbines. M_1 and M_2 are the number of features of the historical input and forecasted weather input, respectively. This means that at a given time step, the model attempts to forecast the next T_{out} time steps for each of the N turbines.

2.2 WIND DATA PREPROCESSING

Typically, the raw input features are processed before being fed through a WPF model. Other than the typical methods like standardization, the processing can involve **signal processing** and **outlier detection**. Outlier detection focuses on dealing with abnormal samples in the original wind data, while signal processing processes all data indiscriminately. Signal processing methods are usually used for two purposes: decomposing the data into subseries and denoising the data. Usually, the predictability on the decomposed subseries is higher than the original data, resulting in more accurate forecasting. There is a risk, though, that outliers still exist in the denoised data. With outlier detection, we are not certain if all outliers are detected completely, and we also risk removing data points that make up the structure of the data (Wang et al. (2021)).

Local Outlier Factor (LOF) is an example of an outlier detection method, which is used later in this project to visualize outliers of a single turbine in the train data, and to, conservatively, select outliers in the train set to impute. For the full details of LOF, see Breunig et al. (2000). Briefly, the algorithm assigns how much the density of each data point deviates from its k neighbors. If a data point is very isolated, then it will have a significantly lower density than its neighborhood, and hence, it the data point will be be considered an outlier.

2.3 FEATURE EXTRACTION

Many features can help improve the accuracy of WPF. However, similar to other domains of machine learning, redundant information in features can have adverse effects. Feature extraction can prevent this problem. It can be categorized into two groups, *classic* and *DNN-based*.

Classic feature extraction can be split further into two categories. The first one helps select useful features from a set of candidate features, for example Phase space reconstruction (PSR), Granger causality testing (GCT), ACF, PACF and mutual information (MI). The other converts candidate features into new ones, for example principal component analysis (PCA) and variants thereof. Since different feature extraction methods have different utilities, e.g. representing linear or nonlinear relationships, some researchers propose using a hybrid (Wang et al. (2021)).

While they may capture an immediate relationship, neither single nor hybrid classical feature extraction methods can deal with the deep features of complex wind power data with highly nonlinear characteristics. Deep networks such as Autoencoders (AE) or Convolutional Neural Network (CNN)-based feature extraction methods can solve this and are widely used in WPF.

Autoencoders are trained unsupervised to reconstruct the features. The encoder generates a hidden layer code, i.e. a representation of features, which the decoder uses as input to reconstruct the actual features. After unsupervised training, the encoder can be used as a step in a supervised setup that forecasts wind power.

Similarly, convolutional neural networks can be used to extract feature representations of the inputs. A simple CNN consists of four layers: input, convolutional, pooling and fully connected layers. The convolutional kernel can be of different dimensions, e.g. 1-D, 2-D or 3-D. For example, in a 2-D kernel, the input may consist of historical wind power data, historical weather data and the historical wind power data of adjacent turbines. This way, the convolutional operation extracts features from the input space both in the temporal dimensional, by convoluting over the historical features, and the spatial dimension, by convoluting over adjacent sites. The limitation, however, is that the spatial dimension must be represented in a grid-like structure, which is rarely the case of wind turbine layouts. That is why GNNs have seen relevance in the field, as we shall see shortly.

2.4 DEEP LEARNING IN WIND POWER FORECASTING

Wind power data contains complex relationship, which traditional linear regression models (e.g. ARIMA) or simple non-linear regression models (e.g. SVM) may fail to characterize well. Many researchers therefore turn to deep networks. Particularly, since WPF is a regression problem, the type recurrent neural networks (RNNs) are popular, like the ones described in the following.

2.4.1 RNNs

Long short-term memory (LSTM) a type of RNN that has proved effective in the literature (for example Sun et al. (2024). It is effective at maintaining historical information from a sequence of inputs, which is why it is well-suited for a regression task like WPF.

Each unit in an LSTM consists of a *cell*, an *input* gate, an *output* gate and a *forget* gate. The cell remembers information over a period of time, and the three gates regulate how information flows into and out the cell.

Similarly, Gated Recurrent Unit (GRU) is a simpler variant of LSTM that is still capable of learning long-short dependency without falling into the vanishing and exploding gradient problem - the problem when gradients shrink or grow uncontrollably, leading to unstable training. As seen in figure 1, GRU does not have the separate memory cell as in the LSTM, as each unit only contains two gates: a *reset* gate and an *update* gate. The former helps decide how much of the previous hidden state is considered, while the latter determines the degree of the previous hidden state that will be used to update the current state. GRU is faster to train than LSTM as it contains fewer parameters, and it is similarly a promising method in the WPF literature (e.g. Huang et al. (2023)).

Mathematically, a GRU unit produces the hidden state at time step t as follows:

$$z_t = \sigma(W_z * [h_{t-1}, x_t]) \tag{2}$$

$$r_t = \sigma(W_r * [h_{t-1}, x_t]) \tag{3}$$

$$\tilde{h_t} = \tanh(W_h * [r_t * h_{t-1}, x_t]) \tag{4}$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h_t} \tag{5}$$

, where z_t and r_t are the update and reset gates. W are the different weight matrices, h_{t-1} is the hidden state of the previous time step, and x_t is the input at the time step t. $\tilde{h_t}$ is then the candidate activation vector and h_t is the final hidden state at time step t.

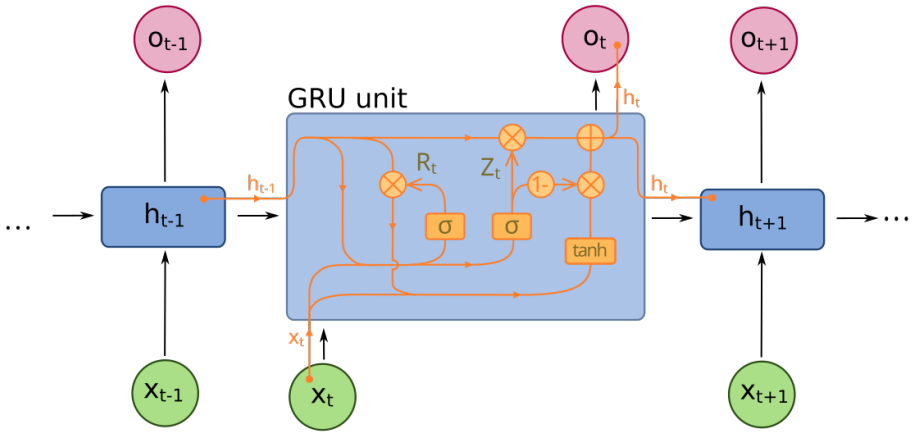

Figure 1: Overview of a single Gated Recurrent Unit (GRU).

Both LSTMs and GRUs can be stacked in multiple layers such that each unit in the layers after the first layer takes in the hidden state of the same time step at the previous layer. In other words, x_t of equation 2-4 is replaced with h_t^{l-1}. Further, stacking two layers with reverse order is a common way to create a bidirectional LSTM or GRU, which can capture temporal dependencies in both directions.

2.5 GRAPH NEURAL NETWORKS (GNNS)

Up until now, this report has covered methods that regard wind turbines as unrelated entities. Those methods produce forecasts based on the historical data of the target turbine only, ignoring nearby turbines. While groundbreaking at the time of their inventions, the methods only utilize the temporal information of the data and not the spatial information. In contrast, newer methods called Graph Neural Networks (GNNs) have shown promising results in the field by utilizing the spatio-temporal patterns in the data across wind turbines using graph-structured data together with the time series data.

But what are graphs exactly? Graphs can be defined as $G = (V, E)$, where V is the set of N nodes and E is the set of edges connecting these nodes. An adjacency matrix $A \in \mathbb{R}^{N \times N}$ can be used to represent the (weighted) edges between the nodes, where A_{ij} is non-zero if there is an edge between node v_i and v_j. A GNN then typically also consists of node features $X \in \mathbb{R}^{N \times D}$, where D_i is a D-dimensional feature vector of node v_i. The set of $G = (A, X)$ is sometimes referred to as an attributed graph. A spatial-temporal graph is then one that consists of multiple, dynamic attributed graphs for discrete times, i.e. $G = \{G_1, G_2, \cdots, G_T\}$, where $G_t = (A_t, X_t)$ is the attributed graph at time t. The relationship between the nodes, i.e. A_t can either be fixed or change over time. In other words, at each time step, we have the graph itself, as relationships between nodes - or turbines - and the features of each node at the given time.

2.5.1 Constructing A Graph

Unlike CNNs, for which the input is typically a grid-like structure, GNNs do not have a natural order of nodes. Therefore, one crucial step of a GNN is to construct a graph that can be used in the network. One question to ask is: How do we think turbines in the data are related? Which turbines should be connected to which, and by how strong of a connection? Should the connection between turbines be bidirectional or unidirectional (undirected or directed graph)? Some of these questions are answered in the following literature. Examples are also given where the graph is not heuristically define but rather learned as part of end-to-end frameworks.

Yu et al. (2020) construct a graph by measuring the euclidean distance between wind turbines and then use a minimum threshold cut-off to create a binary adjacency graph. If the resulting graph is a disconnected graph, then an edge is iteratively drawn between the shortest points between different clusters until the graph is no longer disconnected.

Park and Park (2019) introduce a physics-induced GNN (PGNN) to forecast the wind power of wind farms. They do so by considering the wake effect - a phenomena where upstream wind turbines affect downstream wind turbines by reducing the wind speed and increasing the turbulence. Figure 2 illustrates how the wake effect of wind turbine *j*s is immediately strongest right after the wind passes the turbine and then spreads out and dilutes as the distance to the downstream wind turbine i increases. Wind turbines are represented as nodes and an edge between two nodes is constructed at a given wind direction if the Euclidean distance between the turbines is below a set threshold and if the downstream turbine is within a set angle of the upstream turbine relative to the wind direction. In other words, their graph construction is dynamic as it changes over timesteps depending on the direction of the wind.

Extending further, Wu et al. (2020) use a graph learning layer in their Multivariate Time Series Graph Neural Networks (MTGNN) model to dynamically learn the hidden relationships between the nodes of a spatio-temporal graph. The MTGNN model did well in a wind farm multi-step forecasting competition (Jiang et al. (2023))

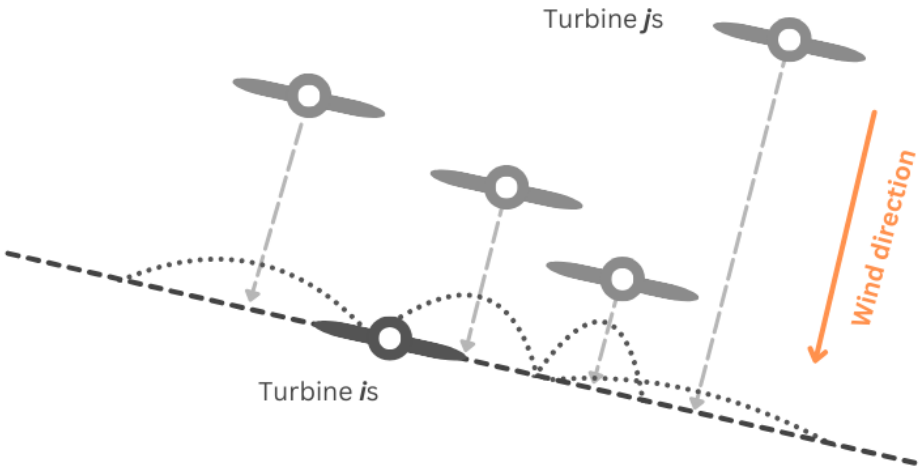

Figure 2: Sketch of the wake effect of wind turbine *j*s on wind turbine *i*.

with 134 wind turbines in a single wind farm. The following equations describe the graph learning layer:

$$M_1 = \tanh(\alpha E_1 \Theta_1) \tag{6}$$

$$M_2 = \tanh(\alpha E_2 \Theta_2) \tag{7}$$

$$A = \mathrm{ReLU}\left(\tanh\left(\alpha\left(M_1 M_2^T - M_2 M_1^T\right)\right)\right) \tag{8}$$

$$\text{for } i = 1, 2, \ldots, N \tag{9}$$

$$\mathrm{idx} = \mathrm{argtopk}(A[i, :]) \tag{10}$$

$$A[i, -\mathrm{idx}] = 0 \tag{11}$$

, where E_1 and E_2 are randomly initialized node embeddings that are learned during training. Θ_1 and Θ_2 are model parameters, α is a hyperparameter and $\mathrm{argtopk}(\cdot)$ returns the index of the top-k largest values of a vector. Equation 8 ensures that adjacency matrix A is asymmetric, meaning relationships between turbines are uni-directional. Equation 9-11 makes A sparse by only choosing the k closest neighbors of each turbine and setting the edges with the remaining nodes to zero, which reduces the computational cost of the following graph convolution.

2.5.2 Graph Operations

A natural next step after defining the graph is to define how we would like a node to aggregate information from its neighborhood. In other words, how do we utilize the information from the neighbors of a turbines? This operation is commonly called a graph operation or graph convolution. It is common in the literature to divide the graph operations of GNNs into two; *Spectral* GNNs and *Spatial* GNNs.

Spectral GNNs is a type of method based on **spectral theory**, eigenvectors and eigenvalues. In essence, the methods transform the input graph signal into the graph spectral domain using a graph Fourier transformation, filters it (with a function with learnable parameters), and then transforms it back. The input graph signal refers to the previously defined matrix $X^{N \times D}$ with a D-dimensional input feature vector for each of N nodes.

To apply the spectral transformation to the graph signal, a modification of the adjacency matrix called Laplacian is used. The Laplacian L encodes the graph connectivity and is defined as:

$$L = D - A \tag{12}$$

, where D is a diagonal degree matrix (containing node degrees on the diagonal and zero values for the rest), and A is the previously mentioned adjacency matrix. The Laplacian includes information about the local structure of the graph with node connections, like the adjacency matrix, but also the global structure of the graph with node degrees.

Often, a normalized version of the Laplacian is used, as it dampens the influence of high degree nodes:

$$L = I - D^{-\frac{1}{2}} A D^{-\frac{1}{2}} \tag{13}$$

, where I is the identity matrix. $D^{-\frac{1}{2}}$ is the inverse of the square root of the degree matrix, where each diagonal element is $D_{ii}^{-\frac{1}{2}} = \frac{1}{\sqrt{D_{ii}}}$.

The Laplacian can then be eigendecomposed as:

$$L = U \Lambda U^T \tag{14}$$

, where U is the matrix of eigenvectors (sometimes called the graph Fourier basis) and Λ is the diagonal matrix of eigenvalues. The eigenvalues of the Laplacian have the property that they represent graph frequencies. Small eigenvalues ($\lambda \approx 0$ correspond to low-frequency components in the graph, representing global, smooth patterns, while large eigenvalues correspond to high-frequency components in the graph, representing local, abrupt changes.

The graph Fourier transformation on the graph input signal x is defined using the matrix of eigenvectors as:

$$\hat{x} = U^T x \tag{15}$$

A filter g_θ can then be applied as a convolution in the spectral domain before the graph signal is transformed back to the spatial domain using U. The whole expression can be written as:

$$g_\theta *_G x = U g_\theta(\Lambda) U^T x \tag{16}$$

Here, $g_\theta(\Lambda)$ is a function of eigenvalues Λ. Often $g_\theta(\Lambda)$ is a diagonal matrix with some learnable parameters θ. An example of a filter is a simple linear polynomial filter:

$$g_\theta(\Lambda) = \theta_0 + \theta_1 \Lambda \tag{17}$$

, where θ_0 and θ_1 are the learnable parameters. In this example, θ_0 is a constant term and θ_1 scales each eigenvalue λ_i in Λ. The diagonal matrix form is then:

$$g_\theta(\Lambda) = \begin{bmatrix} \theta_0 + \theta_1\lambda_1 & 0 & \cdots & 0 \\ 0 & \theta_0 + \theta_1\lambda_2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \theta_0 + \theta_1\lambda_N \end{bmatrix} \tag{18}$$

A problem with the filter above is that $g_\theta(\Lambda)$ can be computationally expensive to compute as it requires eigendecomposition of the Laplacian. This has $O(N^3)$ time complexity - i.e. it grows cubically with the number of nodes in the graph!

Work has been done to overcome the computational complexity. Particularly, early works of Defferrard et al. (2016) approximate the filter $g_\theta(\Lambda)$ using Chebyshev polynomial approximation and thereby creating a Chebyshev convolution used in a ChebNet. $g_\theta(\Lambda)$ is approximated as a polynomial of order K:

$$g_\theta(\Lambda) \approx \sum_{k=0}^{K} \theta_k T_k(\hat{\Lambda}) \tag{19}$$

, where T_k are Chebyshev polynomials and $\hat{\Lambda}$ is a rescaled version of Λ. The convolution is then:

$$g_\theta *_G x \approx \sum_{k=0}^{K} \theta_k T_k(\hat{L}) x \tag{20}$$

Here, $T_k(\hat{L})$ can be computed recursively, without the eigendecomposition, using the recusive definition of Chebyshev polynomials:

$$T_0 = 1 \tag{21}$$

$$T_1 = x \tag{22}$$

$$T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x) \tag{23}$$

The above gives an efficient approximation of spectral graph convolutions on singular graphs. However, for a WPF problem, we need a solution that can work with spatio-temporal graphs, i.e. one - as previously defined - that contains a graph signal at every time step. To solve such, Seo et al. (2016) build upon ChebNet by using the operation in RNNs, like LSTMs and GRUs, to model temporal graph data. They define a Chebyshev Graph Convolutional GRU as:

$$z = \sigma(W_{xz} *_G x_t + W_{hz} *_G h_{t-1}) \tag{24}$$

$$r = \sigma(W_{xr} *_G x_t + W_{hr} *_G h_{t-1}) \tag{25}$$

$$\tilde{h} = \tanh(W_{xh} *_G x_t + W_{hh} *_G (r \odot h_{t-1})) \tag{26}$$

$$h_t = z \odot h_{t-1} + (1 - z) \odot \tilde{h} \tag{27}$$

Here, z and r correspond to the update and reset gates respectively from a traditional GRU and $\tilde{h}$ corresponds to the candidate activation vector. At each step, the input and the previous hidden states are filtered by the ChebNet convolutional operation $*_G$ as previously defined.

Spatial GNNs is another type of method that involves **message passing**. These methods can be viewed as simplified versions of spectral GNNs with filters that are designed directly to the neighborhood of each node. The convolutions are applied in the spatial domain, rather than the spectral domain. The neighborhood of each node is aggregated with a function, which is typically summing, averaging or a learned function.

For each node i, the neighborhood representation at the lth layer can be described as:

$$a_i^l = \text{AGGREGATE}^{(l)}(\{h_j^{(l-1)}, v_j \in N(v_i)\}) \tag{28}$$

, where $h_j^{(l-1)}$ is the feature vector of each node j that is in the neighborhood $N(v_i)$ of node i. AGGREGATE is a function that aggregates the features vectors from the neighbors, and a_i^l denotes the aggregated representation of the neighborhood of node i at layer l.

Then, the features of node i itself at the previous layer $l-1$ is combined with the aggregated message from its neighbors to produce the transformed node embedding of node i at the lth layer:

$$h_i^l = \text{COMBINE}^{(l)}(h_i^{(h-1)}, a_i^{(k)}) \tag{29}$$

Yu et al. (2020) exemplify the use of spatial GNN in WPF by proposing a superposition graph convolution layer (SGNN) for offshore wind power prediction. Here, each node, for each channel, is convoluted with its adjacent neighbours using a weight matrix for that layer. This local computation results in an updated graph for each channel, which they superimpose with the original input in the layer as to encourage feature reuse and alleviate vanishing gradients. Because of the superimposition at each layer, the network grows exponentially per layer, which is why the authors opt for fewer network layers. Their network further consists of a fully connected layer which converts the node specific features from the graph layers to the final values of the output layer.

The results of SGNN are compared with kNN, SVR and LSTM, all with extracted information from the surrounding turbines using the same neighborhood definition as that of the SGNN graph, to allow for a fair comparison. The performance comparison between the models is split into different feature windows, i.e ranges of historic time series used for prediction. Their proposed SGNN model generally performs best, and the authors argue that it is due to the feature retaining capabilities of the superpositioning in the model. They further show that the proposed SGNN model performs better in the central region of the wind farm compared to the edge regions, indicating the effect of the graph convolution operation.

Further, in the aforementioned physics-induced GNN (PGNN) (Park and Park (2019)), the network takes in a wind farm graph as a directed graph with *sender* and *receiver* nodes and learns the edge features and edge weights, as follows:

$$\begin{aligned} e'_{ij} &\leftarrow f_e(e_{ij}, n_j, n_i; \theta_1) \\ w_{ij} &\leftarrow f_w(d_{ij}, r_{ij}; \theta_2) \\ e'_{ij} &\leftarrow w_{ij} \times e'_{ij} \end{aligned} \tag{30}$$

Here, $f_e(\cdot;\theta_1)$ is an edge update function that updates all the edges of the graph by utilizing the node features n_j and n_i of the sender and receiver nodes, respectively, and edge feature e_{ij} between the nodes. The function $f_w(\cdot;\theta_2)$ utilizes the relative positions of the wind turbines, in the form of the downstream wake distance d_{ij} and the radial-wake distance r_{ij} between the wind turbines to compute the edge weights. In other words, the network utilizes the spatial relationship between the turbines in relation to the direction of the wind.

PGNN further updates the features of each node n_i with the current input node features, the aggregated, updated edge features of all the upstream wind turbines of the current node and the global feature, g, which describes the graph. The global graph feature is also updated in the PGN layers. After multiple subsequent PGN layers with residual connections, a graph dense layer follows, which takes as input the updated graph G' from the PGN layers and outputs estimated power values of each wind turbine node.

All the update functions in the PGNN are based on multi-layered perceptrons (MLPs), except for $f_e(\cdot;\theta_1)$ which is based on a physics-induced basis function. As such, the authors argue that the architecture allows the network to learn the wake interactions between the wind turbines that affect the power output.

3 TOOLS

3.1 PYTORCH GEOMETRIC & GEOMETRIC TEMPORAL

Besides using common libraries like NumPy, Pandas, GeoPandas etc. for data wrangling and processing and PyTorch as the main force in the deep learning, this project has used the library `PyTorch Geometric Temporal`[1] (PyGT) (Rozemberczki et al. (2021)) for time series forecasting with graph-like data. PyGT builds upon `PyTorch Geometric`[2] (PyG), which builds upon PyTorch. While PyG allows for training of GNNs, PyGT takes it a steps further and implements a range of acknowledged *temporal* GNN modules to allow for training of temporal GNNs.

One feature of PyGT is its data iterators that allow for iterating over graph data of various types. In this project, we have written a custom class that returns a `StaticGraphTemporalSignal` object. This is a graph type where the edges are static and the temporal features are dynamic. Thus during training, the graph must only be loaded to GPU once, i.e:

```
1 device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
2 for snapshot in train_dataset:
3     static_edge_index = snapshot.edge_index.to(device)
4     static_edge_weight = snapshot.edge_attr.to(device)
5     break
```

Python code 3.1: Loading graph to GPU once

, where `snapshot` is a snapshot of a single timestep, including the node features and graph definitions of that timestep.

For some of the architectures, we rely solely on `PyTorch Geometric` for the graph convolutional layer, and then `PyTorch` for the RNN layer.

3.2 CARBONTRACKER

`Carbontracker`[3] (Anthony et al. (2020)) is a Python library made for users to estimate the carbon emission of training deep learning models.

In practice, the tool measures the total power of components such as GPU, CPU and DRAM during training. To account for the power consumption of the supporting infrastructure, Carbontracker multiplies the measured power by Power Usage Effectiveness (PUE), which is a measurement of the energy efficiency of the data center that handles the computation and is given by:

$$\text{PUE} = \frac{\text{Total Facility Energy}}{\text{IT Equipment Energy}} \tag{31}$$

Since the exact PUE is not always known for any data center, Carbontracker uses a global average, for example 1.55 for 2022.

Carbontracker further fetches city-level carbon intensities every 900 seconds which are multiplied by the power consumption during the interval. This results in the total carbon footprint of the training framework:

$$\text{Carbon Footprint} = \text{Power Consumption} \cdot \text{Carbon Intensity} \tag{32}$$

[1] https://pytorch-geometric-temporal.readthedocs.io/en/latest/index.html
[2] https://pytorch-geometric.readthedocs.io/en/latest/
[3] https://github.com/lfwa/carbontracker

The carbon intensities are based on the IP Adress of the compute, which is found by the Python libary `geocoder`. Due to lack of a publicly available, globally accurate and free real-time carbon intensity database, only Great Britain and Denmark have available carbon intensities. The remaining use static data, like average values of EU-28 countries. The API used in Denmark, where this project is conducted, is from Energi Data Service[4].

[4] https://www.energidataservice.dk/

4 DATA AND MATERIAL

4.1 DATA DESCRIPTION

This project works mainly with three data sets. First, a large-scale data set of most wind turbines in Denmark with quarterly or hourly information about the kWh (output) produced per turbine, coupled with meta data like coordinates, rotor diameter and navhub height. Table 1 and 2 are lists of all the variables in the historic turbine data (settlements) and the meta data respectively. A few redundant variables are omitted, like `Turbine short name`.

Table 1: Settlement data of wind turbines

Column	Explanation
GSRN	The unique identifier of wind turbines
VAERDI	The reading from a metering point in kWh
TIME CET	Timestamp in UTC+1. Some locations (Jutland and Fyn) have quarterly timestamps, while the rest (Zealand) has hourly

Table 2: Meta data for wind turbines

Column	Explanation
GSRN	The unique identifier of wind turbines
Turbine type	H: Household turbine, W: Single turbine, P: Turbine park, M: Turbine in a park
Parent GSRN	For turbines of type "M" this value refers to the parent matering point, since turbines in a park do not have individual settlements but rea aggregated to the parent
In service	Start of production
Out service	End of production
BBR municipal	BBR code of the municipal where the turbine is located
Placement	Land: Onshore, Hav: Offshore
UTM x and UTM y	UTM 32 coordinates
UTM precision	UTM precision
Capacity kw	Maximum production capacity in kW
Model	An 8-digit INT
Manufacturer	An 8-digit INT
Rotor diameter	Turbine rotor diameter in meters
Navhub height	Height in meters of the turbine navhub
Valid to and Valid from	The timeframe in which the master data is valid

The second data set is from DMI (The Danish Meteorological Institute) and contains hourly data about a broad range of observed, **historical** weather conditions. The data can be fetched in different dimensions, most immediate option being the

observable weather conditions at weather station coordinates. The goal of including weather data is to use it as predictors for the power output of the wind turbines. Since the wind turbines and weather stations do not share coordinates, some sort of interpolation is necessary to map the two.

DMI Open Data[1] provide the observed weather data in interpolated spatial resolutions (*10kmGridValue, 20kmGridValue, municipalityValue* and *countryValue*). This project opts for the 10 km grid values as those are the most granular. The interpolation from station values to the grid is done using a modified inverse-distance algorithm into a 1x1 grid covering whole Denmark. Each station is weighed by the distance and climatological comparability to the grid cell. The 1x1 cells are aggregated to other resolutions, such as the 10x10 grids used here. Table 3 gives an overview of the available parameters on the grid-level, where some of the parameters are sorted out due to immediate irrelevance, like grass height temperature.

Table 3: Parameters in the 10x10 grid of observed, historical weather data

Parameter	Explanation
from	Start time of weather measurement
to	End time of weather measurement
timeResolution	Time resolution of weather data, e.g. hourly or daily
coordinates	Coordinates of 10x10 grid
mean wind speed	Mean wind speed in m/s
mean wind dir	Mean wind direction (in degrees)
mean temp	Mean temperature in Celcius
min temp	Minimum temperature in Celcius
mean relative hum	Mean relative humidity in percentage
max wind speed 10min	Maximum wind speed over 10 minutes averages
max wind speed 3sec	Maximum wind speed over 3 seconds averages
mean pressure	Mean pressure measured in hPa
mean cloud cover	Fraction of the sky covered by clouds, measured in percentage

The third data set consists of **forecasted** weather data from DMI in a grid format similar, to the DMI histocial Open Data above. The data from 2008-2018 is based on the "HIR" model, while the data from 22th February 2018 is based on the more precise "ENetNea" model. The forecasted weather data contains the following variables: temperatur at 2 meter height, wind direction at 10 and 100 meter height, wind speed at 10 and 100 meter height. The forecasts are in an hourly format with up to 55 hours ahead, and the 55-hour long forecasts start in inconsistent intervals but around every three hours.

A time period of two years (2018 and 2019) was chosen for all data sets to scope down the working data but still allow for seasonal trends in the data.

4.2 DATA ANALYSIS

To get a better grasp of the data, let us first look into the raw data. Figure 3 shows all the turbines in the dataset, their respective types and their XY-coordinates. One thing to note is that the household turbines and turbine parks do not have coordinates. They represent 18.6% (15817) and 8.1% (6937), respectively, while turbines in

[1] https://opendatadocs.dmi.govcloud.dk/DMIOpenData

a park and single turbines represent 5.9% (4996) and 67.4% (57400), respectively, of the total dataset of 85150 turbines.

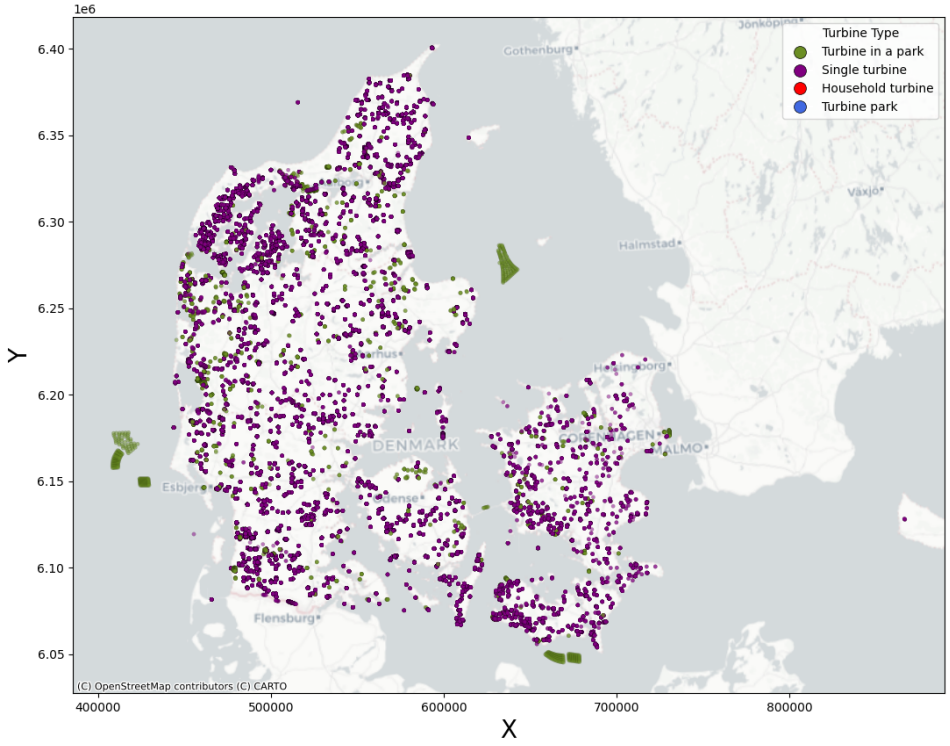

Figure 3: Map of all the turbines in the dataset. Household turbines and turbine parks are not shown as they do not have coordinates in the data.

The next section (4.3) describes how the raw data is preprocessed, which narrows down the amount of turbines. Figure 4 shows the resulting turbines after the initial preprocessing together with their capacity (maximum kWh production). The succeeding figure 5 zooms in on a subset area (Northern Jutland) which will be the focus for the remaining of the paper. The area was chosen arbitrarily because it had a cluster of turbines with varying capacity, and it is still a relatively big scale (bigger than a single wind farm) with 348 turbines.

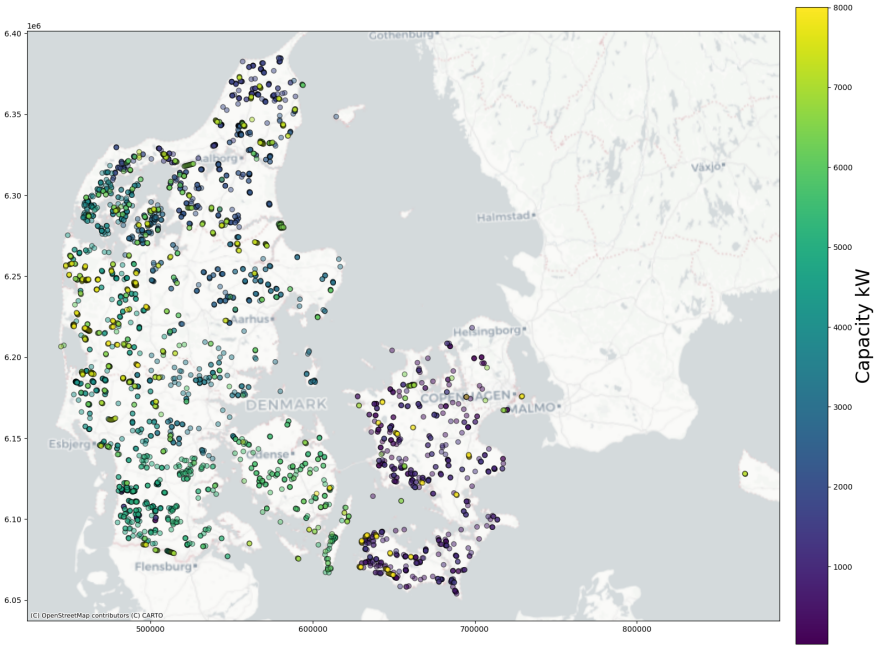

Figure 4: Map of all turbines in the data after initial data cleaning, including the maximum production capacity in kW of each turbine.

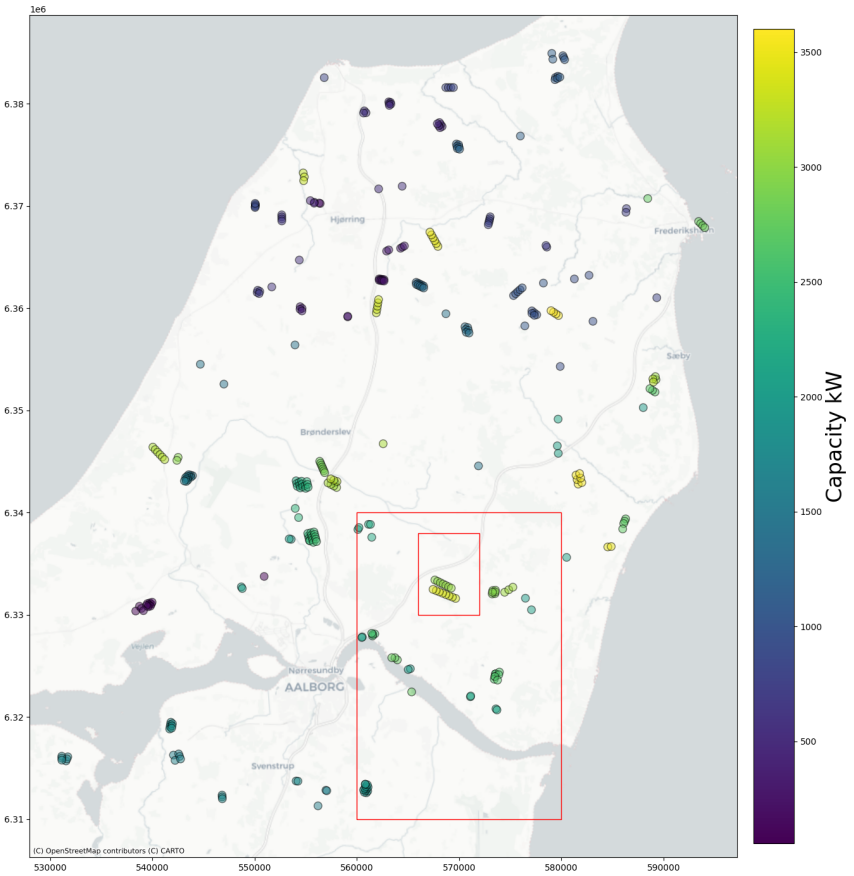

Figure 5: Zoomed map on the northern part of Jutland with wind turbines and their capacity. This is the data used in the project with different scales, where the smallest red square is the boundary subset 1, the largest square is the boundary of subset 2, and the whole map is subset 3.

As mentioned in the background, it is very typical for wind turbines to express a power curve in terms of wind power with different regions, like cut-in and cut-off. This seems to be empirically true when looking at the scatter plot figure 6 of a single turbine in the data. The plot shows how the wind power peaks around the denoted capacity of the turbine, and that most of the samples with low wind speed equate zero wind power (cut-in region). The plot includes outliers that were detected using Local Outlier Factor (`LocalOutlierFactor` from the library `sklearn`) with 20 neighbours. There seems to be a broad range of outlier types, and even though the outlier detection does not classify them, we can still infer outliers of certain types. For example, some samples have a near-optimal wind speed, but the turbine still produces zero kWh. This can be due to a multitude of factors, such as turbine malfunction, maintenance, etc., which are not present in the data.

We can also observe a natural variation in kWh across samples with identical wind speeds. It is suspected that this variation is caused by other direct variables in the data, such as wind direction, air temperature etc, or because of latent factors such as wake effect from neighbouring turbines.

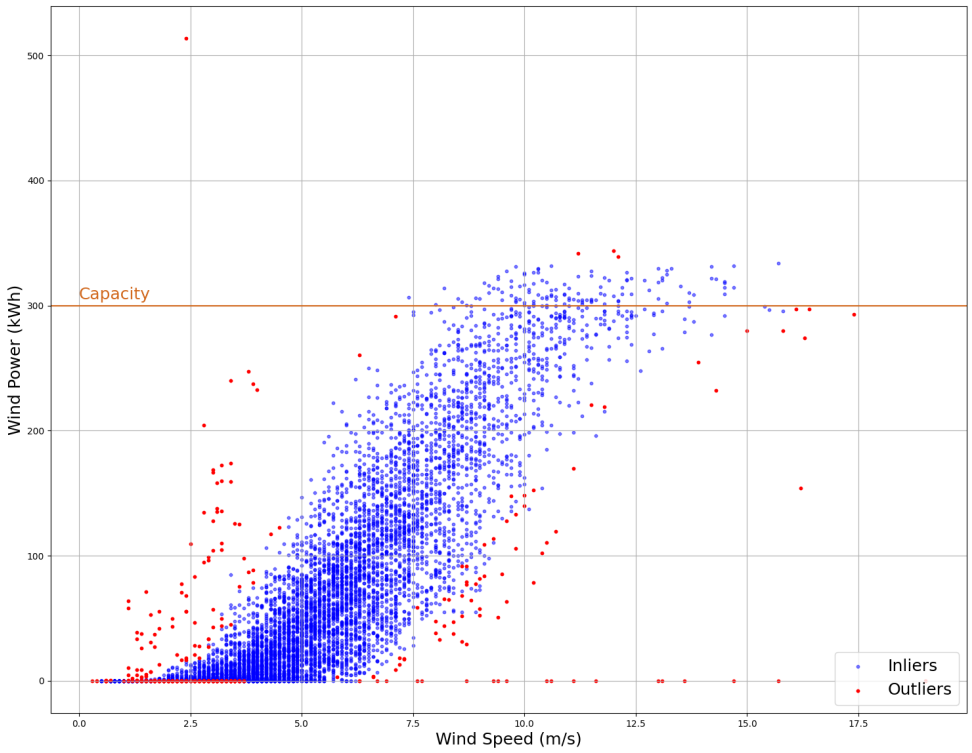

Figure 6: Scatter plot of wind power as a function of wind speed for the train data (one year) of one turbine. Outliers detected with Local Outlier Factor are marked.

Figure 7 shows a snapshot of the wind conditions in Denmark over one hour, with the mean wind speeds and mean wind directions of each tile in the 10x10 grid weather data. The maps indicates what you would expect from wind conditions; that nearby tiles have similar wind conditions at the same timestamp.

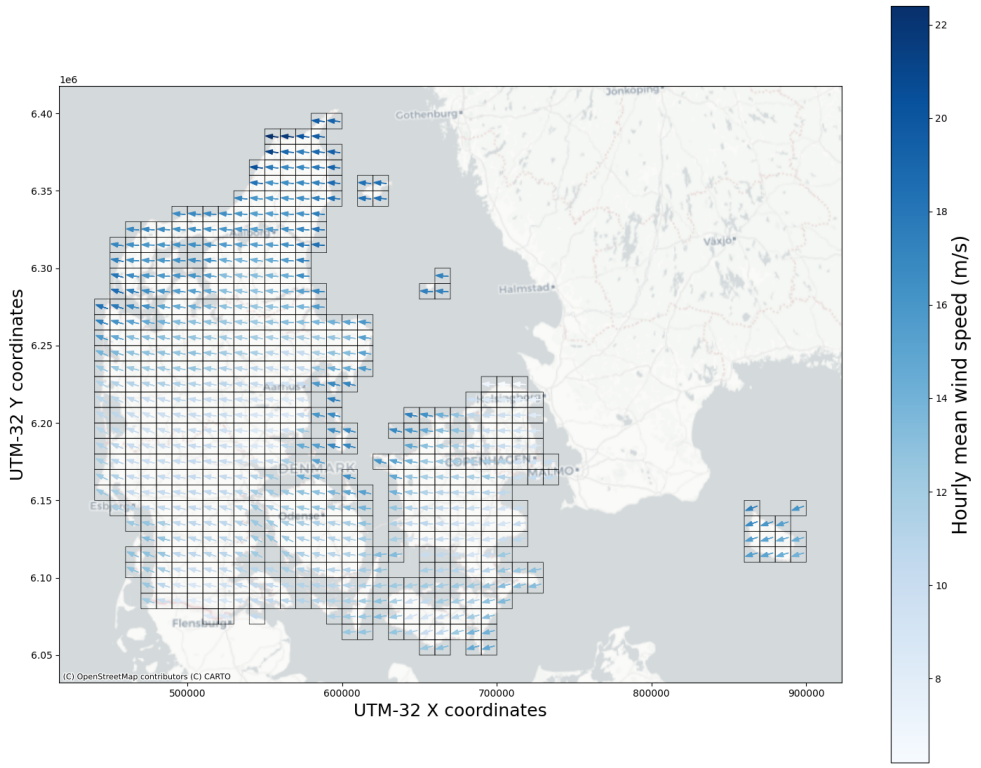

Figure 7: Snapshot of 1-hourly wind speeds and wind directions in 10x10 km grid over Denmark.

4.3 DATA PROCESSING

The main data processing can be split into two categories, represented by two python files, `initial_wrangling.py` and `feature_processing.py` in the Github repository. The former consists of heavy, one-time processing of the data, such that it matches a format that can be used for the study.

The main purpose of this one-time processing is to load the daily weather JSON files and join them with the turbine data by locating the turbine coordinate in the weather grids. The timestamps of the turbine data are in UTC+1 format and are therefore subtracted by one hour to convert them into UTC, which matches the format of the weather data. Then, the two data sets are inner joined on timestamps and the cellId that each turbine was within. This eliminates turbines that did not have coordinates in the data, reducing the number of turbines. Some turbines had a significantly low number of timestamps in the whole period, which would make them little useful in the experiment, so they were removed. All of this results in a single data set split into two files, one for 2018 and one for 2019, representing a train set and a test set, with turbine power outputs and historical weather conditions at their locations.

The one-time processing also involved curating the forecasted weather data such that for each starting time step in the wind power data, there would be a sequence of hourly forecasted weather data corresponding to the time steps ahead that we want our model to forecast. In our case, we want our models to forecast 24 hours ahead, so we need the future 24 hours of weather forecasts at any given time step. As mentioned, the forecasted weather sequences did not begin at every hour, so at each starting time, the sequence with the most recent starting time was chosen. For example, the time "2018-01-01 01:00" is a part of nine different forecasting sequences that begin at one of the following amount of hours before: 50, 44, 38, 32, 26, 20, 14, 8

or 2. To get the most accurate forecast, the lowest value "2" is chosen together with the consecutive sequence of *2 to (forecast_length + 2)* hourly forecasts.

For the recurrent processing, a `Pytorch` Custom DataLoader[2] was created. This seperates data loading and network training, and it provides an iterable with mini-batches over the dataset. Class arguments decide whether to load the train set or test set, which weather features to include, which X and Y coordinate intervals of turbine data to include (i.e. the subsets of data), and which feature preprocessing pipeline to apply on the loaded data.

The feature preprocessing includes removing `NaN`-values by replacing them with the mean of the same feature at the same turbine, standardizing/normalizing, handling outliers and handling degrees. Most features are standardized by subtracting the mean and dividing by the standard deviation for the particular feature at each individual turbine. This is except for the turbine power outputs, relative air humidity and cloud cover, which are normalized, as they have clear minimum and maximum values. Outliers are handled at a turbine level for the two features wind speed and wind power by; 1) detecting the outliers using Local Outlier Factor with 20 neighbors and; 2) interpolating them with the mean of the 2 nearest neighbors for the particular turbine. Lastly, the wind direction feature, which is originally in degrees from 0 to 360, is encoded into two new features $\theta \mapsto (sin(\theta \cdot \frac{\pi}{180}), cos(\theta \cdot \frac{\pi}{180}))$. The euclidean distance in this transformation reflects better the true angular distance of the wind direction, and it eliminates the problem that an angle of $1 \approx 360$.

For the feature processing of the test set that requires statistics such as the means and standard deviations, then those are taken from the train set to avoid data leaking. The outlier interpolation is only applied to the train set such that the test set is kept unchanged when used as true label in the evaluation, as a means to make the experiments come as close to real world scenarios as possible, where outliers may be present. For training and as inputs, interpolation of outliers prevents the models from learning from outliers that they had no chance of forecasting correctly - especially since the data set lacked information about turbine downtime.

[2] https://pytorch.org/tutorials/beginner/data_loading_tutorial.html

5 METHOD

5.1 EXPERIMENTAL SETUP

The experiments are conducted by training various models on the train set of 2018 data and comparing their performances on the test set of 2019. To get a grasp of how the GNNs scale, experiments were conducted on different subset sizes of the train and test set with 15, 67 and 348 wind turbines out of the whole Northern Jutland, as previously mentioned.

5.2 MODEL ARCHITECTURES

5.2.1 Simple Baselines

A couple of simple persistence baseline models were evaluated. One of them (naive) always forecasts ahead the values of the most recent value of the history. The other (mean) forecasts the mean of the values in the history window.

5.2.2 GRU Baseline

A more advanced baseline was also evaluated, consisting of a GRU-based model, inspired by a seq2seq architecture. The model consists of two layers of bidirectional GRU: First, an encoder layer that takes in the historical weather and wind power data to produce a hidden state. Second, a decoder that is initialized with the hidden state of the encoder, takes in the forecasted weather data and produces an output. The output of the last GRU unit in the decoder is fed through a linear layer to produce the final outputs.

Training a unique RNN for each turbine quickly became unfeasible as the number of turbines grew, so another approach had to be taken. Instead, a single model with shared layers for each turbine was trained, where turbine identification was added as extra features to the inputs of both the GRU layers, so they could differentiate between turbines. Since one-hot encoding would produce very sparse data, an embedding layer was instead used. The embedding layer is a trainable lookup table of vectors that takes in the ID for a turbine and outputs a vector of a predefined length as a hyperparameter (typically much smaller than the number of turbines). The output vector of the embedding layer was concatenated to the inputs of both the encoder and decoder.

The training data has a lot of redundancy in the history, so layer normalization and a dropout layer was added between the decoder GRU and the linear layer in the hopes of reducing overfitting and stabilizing training.

5.2.3 GNNs

Initially a model with a Chebyshev Graph Convolutional Gated Recurrent Unit Cell (ChebGRU) from `PyTorch Geometric Temporal` was trained as it allowed for a simple model based on the previously mentioned temporal version of Chebyshev graph convolutions (Seo et al. (2016)). However, due to a lack of native batching, the model training was slow and hyperparameter tuning took more than 2 days on the HPC. Instead, the models described below were used in the experiments as they

all allowed for batch training. For all the GNNs, unless explicitly stated, the graphs were initialized with a euclidean distance threshold of maximum 500 meters for two turbines i and j to be connected and edge weights based on exponential decay, i.e.:

$$\text{edge\_weight}_{ij} = e^{-0.001 \cdot \text{distance}_{ij}} \tag{33}$$

Equation 33 (also shown in figure 8) gives exponentially higher weight to turbines closer. For example, turbines with 50 meters apart have an edge weight around 0.95 while turbines 450 meters apart an edge weight around 0.64. For all GNNs, the edge definitions remain static unless explicitly stated. The different GNN architectures are described below.

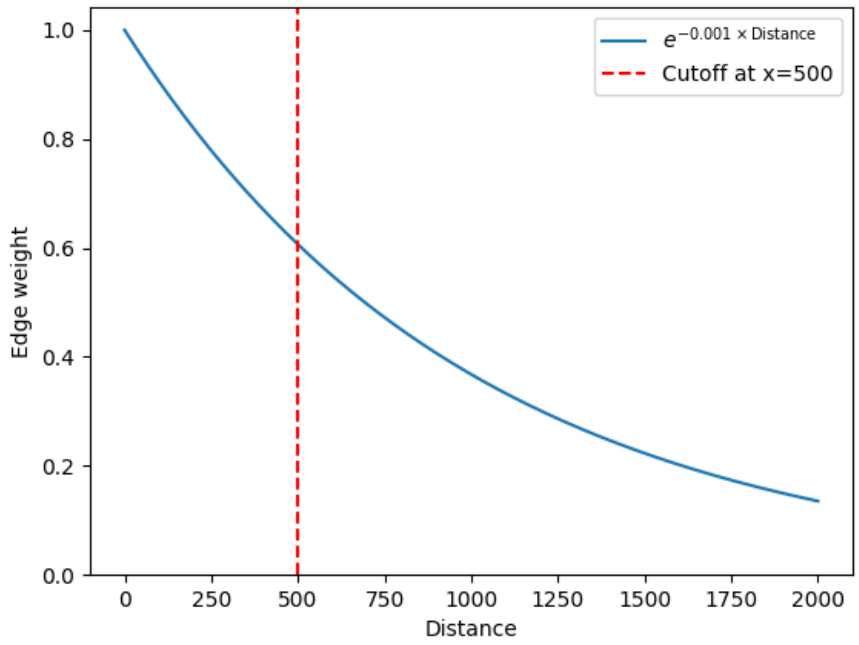

Figure 8: Edge weights as a function of distance between turbines

ChebGRU: A modification of the GRU baseline with a Chebyshev convolutional layer from the `PyTorch Geometric` library added to the forecasted weather data. The GNN layer converts the inputs from $T \times N \times M$ to $T \times N \times H$, where T is the amount of timesteps, N is the number of turbines, M is the number of input features and H is the hidden channels size after the graph convolution.

The GNN layer does not natively support a temporal dimension. Therefore, at each forward pass, the batch dimension and time dimension of the input is combined into a bigger batch of size $batch\_size \times time$ before feeding it through the graph convolution. Afterwards, the batch and time dimensions are separated again in the updated, forecasted data, which together with the historical data is fed through the baseline GRU.

In other words, the GNN is exactly similar to the baseline GRU, except is adds a graph convolutional operation to the forecasted weather data first.

GCNConv: Similar to above, but using a simple graph convolutional operation based on the adjacency matrix (Kipf and Welling (2016)) instead, before feeding the output to the turbine GRUs. The graph convolution is defined as:

$$X' = \hat{D}^{-1/2}\hat{A}\hat{D}^{-1/2}X\Theta \tag{34}$$

, where $\hat{A} = A + I$ is the adjecency matrix with self-loops and $\hat{D}_{ii} = \sum_{j=0} \hat{A}_{ij}$ is its diagonal degree matrix. The adjacency matrix can contain other values than ones and zeros, indicating the weight between turbines. X is the matrix of node input features and Θ is a matrix of trainable parameters.

GCNConv with EdgeConv: Similar to the GCNConv above, but with an added edge convolutional layer before the node convolutional layer. The edge convolutional layer (based on Wang et al. (2018)) converts the input from n input features to e hidden features before the node convolutional layer. The edge convolution for each node x_i is defined as:

$$x'_i = \sum_{j \in N(i)} h_\Theta(x_i \| x_j - x_i) \tag{35}$$

, where h_Θ denotes a neural network and in this case, we use a simple MLP. For each source node, the edge convolution feeds the MLP with a concatenation of the target node features and the features of the source node to produce updated node features. The edge convolution does not consider edge weights as all connected nodes are used equally.

A3TGCN: This network is built natively in `PyTorch Geometric Temporal`, and it is an implementation of the Attention Temporal Graph Convolutional Cell (Zhu et al. (2020)). The module combines graph convolution with GRU to produce hidden states, which are then inputted to an attention model to get a context vector that can produce predictions. The hidden states for each time step t are produced as follows:

$$u_t = \sigma(W_u * [GC(A, X_t), h_{t-1}] + b_u) \tag{36}$$

$$r_t = \sigma(W_r * [GC(A, X_t), h_{t-1}] + b_r) \tag{37}$$

$$c_t = \tanh(W_c * [GC(A, X_t), (r_t * h_{t-1})] + b_c) \tag{38}$$

$$h_t = u_t * h_{t-1} + (1 - u_t) * c_t \tag{39}$$

, where $GC(A, X_t)$ is the same graph convolutional operation as in GCNConv. Then, the weight of each hidden state h is computed in an attention model by Softmax in an MLP. The weighted sum of each h produces the final context vector, which is fed through a fully connected layer to produce the forecasts for each turbine. Since both the feature and time dimensions of the historical data and the forecasted weather data is different, the architecture consists of a separate A3TGCN layer for each of the inputs. The outputs of the two layers are concatenated before being fed through a linear layer.

5.3 HYPERPARAMETERS

To find the optimal hyperparameters, each deep learning model was trained to forecast 24 hourly timesteps ahead in time series cross validation splits of several folds, and then the best epoch RMSE loss of up to 30 epochs with early stopping was averaged across each fold. Doing it this way ensures that the validation data always comes temporally *after* the training data such that the model is never trained on future data to be evaluated on past data. The cross validation split further ensures that multiple yearly seasons of the training data of one year are used, rather than only evaluating the hyperparameters on the latter part of a year. This can be important as wind turbines may have a seasonal variation.

A split of 3 folds was chosen because too small subsets (with 5 or more folds) led to train sets with statistics that did not match the validation set when standardizing/normalizing.

Figure 9 shows the data splits of each fold. As seen in the figure, the first indices of the train data are used for all the splits, meaning they are weighed higher in the evaluation of hyperparameters. This could be fixed by removing the repeated train data at each split, such that each fold contains unique train and validation data. However, this was not done as it would would drastically reduce the quantity of train data for the latter splits.

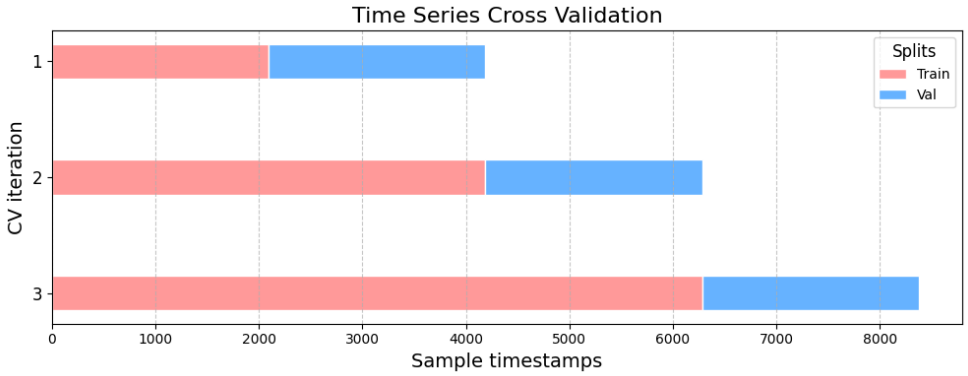

Figure 9: Time series cross validation data splits of training data

Table 4 gives an overview of the hyperparameter space that was searched for, for each model. To limit the search space, all the models were trained with some frozen hyperparameters, like input sequence length. All models were trained on decreasing batch sizes of 20, 7 and 2 for subset 1-3, respectively, except for A3TGCN, which was always trained with a batch size of 100. This was done because the graph layers and the GRU layers of the models combine the original batch dimension with the time dimension and turbine dimension, respectively, leading to memory issues if the original batch size was too large. This should not be a problem for performance, as *"batch size governs the training speed and shouldn't be used to directly tune the validation set performance. Often, the ideal batch size will be the largest batch size supported by the available hardware."*[1]. Instead, other parameters should be tuned to optimize performance.

All the training was done using ADAM-optimizer (Kingma and Ba (2014)) with a tuned learning rate of 0.001 or 0.0005, a fixed weight decay of 0.00001 and the aforementioned average RMSE as loss function. All training was with early stopping if the validation error did not decrease for 5 epochs in a row after the first 10 epochs. Each hyperparameter combination was used to train the model on all the 3 time series folds and evaluated on the validation set of the given fold. The performance was then averaged across the folds to find the approximately optimal selected combination. This means that the number of combinations can be multiplied by 3 to get the total number of training runs. Number of epochs was recorded as a rounded average between the folds with the best parameter set. This is done such that the values can be reused during the final training without leaking information from the test set to do early stopping.

[1] https://github.com/google-research/tuning_playbook#choosing-the-batch-size

Table 4: Hyperparameter search space and selected values for each deep learning model, on each of the three subsets of the data with 15, 67 and 348 turbines, respectively.

Model	Hyperparameters	Search Space	# of Combinations	Selected Values		
				Subset 1	Subset 2	Subset 3
Turbine-level GRU	Embedding dim.	[4, 8]	12	*8*	8	8
	GRU hidden sizes	[16, 32, 64]		32	64	64
	Learning rate	[0.0005, 0.001]		0.0005	0.0005	0.0005
	Epochs	1 to 30		23	20	21
GCNConv-GRU	Embedding dim.	[4, 8]	24	*8*	8	8
	GCN channels	[8, 16]		*8*	8	16
	GRU hidden sizes	[16, 32, 64]		32	16	16
	Learning rate	[0.0005, 0.001]		0.0005	0.0005	0.001
	Epochs	1 to 30		21	17	4
ChebConv-GRU	Embedding dim.	[4, 8]	48	*8*	4	4
	GCN channels	[8, 16]		*8*	8	16
	K	[1, 2]		2	2	1
	GRU hidden sizes	[16, 32, 64]		*16*	16	64
	Learning rate	[0.0005, 0.001]		0.0005	0.0005	0.001
	Epochs	1 to 30		27	17	6
EdgeConv-GCNConv-GRU	Embedding dim.	[4, 8]	48	*8*	8	4
	Edge conv. channels	[8, 16]		*8*	8	8
	GCN channels	[8, 16]		*16*	16	16
	GRU hidden sizes	[16, 32, 64]		*16*	16	16
	Learning rate	[0.0005, 0.001]		0.001	0.0005	0.0005
	Epochs	1 to 30		18	19	10
A3TGCN	GRU hidden sizes	[16, 32, 64]	6	32	64	64
	Learning rate	[0.0005, 0.001]		0.001	0.001	0.001
	Epochs	1 to 30		22	27	27

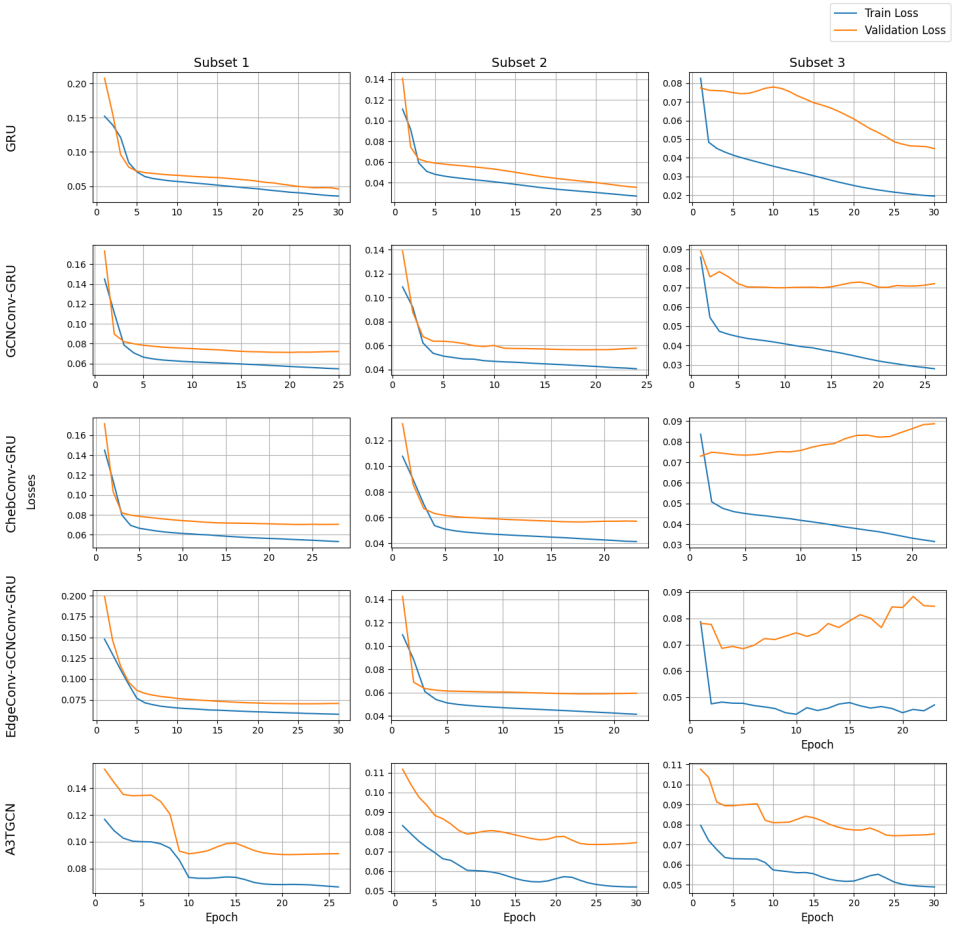

Figure 10: Examples of training and validation losses of the best hyperparameters of each model on each subsets of data. The losses are from the last split in the time series cross validation.

5.4 EVALUATION

Before evaluating, we need to calculate the effective size of the data, i.e. the number of effective, individual samples that can be used for training and testing. Generally, the effective size is given by equation 40:

$$S = \text{total time steps} - I - O + 1 \tag{40}$$

, where S is the effective size, I is the input sequence length and O is the output sequence length. For example, with an input size of 48 (2 days), and output size of 24 (1 day), a total train time steps of 8735, we have an effective size of 8664 for train. In other words, we have 8664 samples to train over for each turbine. Similar process is applied to the test set for evaluation.

The loss function and validation metric during training is the average Root Mean Squared Error (RMSE) for each individual turbine and time step forecast, as such:

$$\text{Average RMSE} = \frac{1}{S \cdot N \cdot T} \sum_{i=1}^{S} \sum_{j=1}^{N} \sum_{k=1}^{T} \sqrt{(y_{i+j}^{k} - \hat{y}_{i+j}^{k})^2} \tag{41}$$

, where S is the effective size of the data set, N is forecast-ahead window (for example 24 for one day), T is the number of turbines, $y_{i \cdot N+j}^{k}$ is the true wind power for turbine K at time step j of the ith sample and $\hat{y}_{i \cdot N+j}^{k}$ is the forecasted value. The equation looks complex, but in reality we simply take the RMSE of each forecasted value and average it over all the turbines, forecast window size and amount of samples.

Taking inspiration from Zhou et al. (2022), we also evaluate the average Mean Absolute Error (MAE) - similar to equation 41 but without the squaring and square root of the inner error term. This metrics weighs errors more equally compared to RMSE which penalizes higher errors more. We use an average of the two metrics at each sample as given by 43 as a balanced **score** metric.

$$\text{Average MAE} = \frac{1}{S \cdot N \cdot T} \sum_{i=1}^{S} \sum_{j=1}^{N} \sum_{k=1}^{T} (y_{i+j}^{k} - \hat{y}_{i+j}^{k}) \tag{42}$$

$$\text{Score} = \frac{1}{S \cdot N \cdot T \cdot 2} \sum_{i=1}^{S} \sum_{j=1}^{N} \sum_{k=1}^{T} \sqrt{(y_{i+j}^{k} - \hat{y}_{i+j}^{k})^2} + (y_{i+j}^{k} - \hat{y}_{i+j}^{k}) \tag{43}$$

Beside the average turbine-level RMSE, RMSE of the whole test area was also evaluated, as this gives an indication of area-wide performance of the model, rather than average performance on individual turbines. The metric is given by:

$$\text{Area RMSE} = \sqrt{\frac{1}{S \cdot N} \sum_{i=1}^{S} \sum_{j=1}^{N} \left(\sum_{k=1}^{T} \left(y_{i+j}^{k} - \hat{y}_{i+j}^{k} \right) \right)^2} \tag{44}$$

The difference here is that equation 44 sums the forecasts over all the turbines T, i.e. the whole area, before computing the RMSE.

6 RESULTS & DISCUSSION

In the final experiment, each model was trained on the whole train set (2018) and evaluated on the whole test set (2019), for each of the subsets of data, with the optimal hyperparameters found in tuning. The results can be seen in table 5, with a sub-table for each subset of data. The outputs of the models are rescaled before the evaluation, meaning the metrics have a meaningful interpretation. For example, as the `GCNConv-GRU` model has an RMSE of 303 on the subset 2, it means that the forecasts of the model on average deviate 303 from the true kWh output of the turbine.

Table 5: Performance of each model on each of the three subsets of increasing turbine quantity. Avg. RMSE and Avg. MAE are calculated for every single forecast and true target of all turbines and then averaged over the total amount of points. Score is an average between the two metrics. Area RMSE is calculated over the whole area after summing the forecasts and the true targets, separately, of all turbines.

Subset **1**

Model	Avg. RMSE	Avg. MAE	Score	Area RMSE
Naive Persistence	826	574.1	700	11195
Mean Persistence	755.2	599.2	677.2	10418.7
Turbine-level GRUs	**658.7**	**513.1**	**585.9**	**8861**
GCNConv-GRU	688	531.8	609.9	9339.7
ChebConv-GRU	685.5	532.9	609.2	9283.7
EdgeConv-GCNConv-GRU	682.3	529.6	606	9260.4
A3TGCN	704.2	550.8	627.5	9554.9

Subset **2**

Model	Avg. RMSE	Avg. MAE	Score	Area RMSE
Naive Persistence	364.3	250.4	307.3	20718.9
Mean Persistence	335.9	264.8	300.4	19801.4
Turbine-level GRUs	**256.1**	**182.1**	**219.1**	**13592.3**
GCNConv-GRU	303	233.2	268.3	17419.1
ChebConv-GRU	304.2	233.2	268.7	17441
EdgeConv-GCNConv-GRU	302.2	232.8	267.5	17349.5
A3TGCN	303.9	229	266.5	17202.6

Subset **3**

Model	Avg. RMSE	Avg. MAE	Score	Area RMSE
Naive Persistence	300.5	204.9	252.7	84446.6
Mean Persistence	282.3	224.5	253.4	83827.2
Turbine-level GRUs	**217.8**	**148.2**	**183**	**58011.5**
GCNConv-GRU	291.8	210.2	251	86216
ChebConv-GRU	278.4	197.5	238	81923.1
EdgeConv-GCNConv-GRU	274.4	197.6	236	80193.5
A3TGCN	285.6	221.8	253.7	83038.6

The results show clearly that the baseline GRU has the best performance across the board. Adding graph convolution to the forecasted weather input (as done in `GCNConv-GRU`, `ChebConv-GRU` and `EdgeConv-GCNConv-GRU`) does not seem to improve the performance in any of the subsets. On the smaller scale, at subset 1 and 2,

the performance of the GNNs seem stable, however, at subset 3 they perform significantly worse than the baseline GRU. Zooming in on the research question, the GNNs do not seem to scale well in terms of performance, given this WPF experiment. But do the results generalize? Or are they only a function of the given setup?

There are many ways one can set up GNNs to work with time series, particularly in WPF. Recall from the background how one paper sought to model the wake effect of nearby turbines with a GNN. The choice of edges and edge weights in this thesis was inspired by that, using a relatively short distance cut-off of just 500 meters, such that only nearby turbines would have effect on the graph convolution. This was also done as a means to cut the computational cost of the GNN layer such that it would be more scalable.

However, as weather tends to be very similar, sometimes identical, within a 500 meter radius of a turbine, the GNN layer might not have enough information in our architectures to learn from the spatial relationships between the turbines. Given the scale of the data, future studies might instead consider trying out a longer distance cut-off. This could possibly allow the GNN layers to model how the weather at much further away locations will affect the wind power output of a given turbine in the future, as the weather moves location. This could improve performance - but it would also increase the computational cost of the GNN layer, as the number of edges would grow.

To assess whether a solution such as increasing the edge count between the turbines would be feasible, we can analyze the computational complexity of the graph convolutional operation in each model, together with how often the graph operation is repeated per sample. This is given in table 6.

The graph operation of the `A3TGCN` model is repeated more per sample, because it has a separate layer for each of the two inputs, and each of them has three graph operations, corresponding to the gates and activation of the GRU (see equations 36-38). The graph layers all grow linearly in time complexity with both turbines N and edges between them E. Typically, K is a small value, and so are F_{in} and F_{out} in our case (the input and output features of the convolution). Therefore N and E are the bottlenecks. In a fully-connected network of turbines, there would be $N(N-1)/2$ or $O(N^2)$ edges, meaning the graph convolutions would grow quadratically with the number of turbines. Clearly, we need to limit the number of edges in the graph convolutional layers in some way!

Table 6: Estimated time complexities of graph operations of each GNN model, including the repetitions of the operation per sample. N is the number of turbines, F_{in} and F_{in} are the input and output features, E is the number of edges, and K is the number of Chebyshev polynomials. T_1 is the sequence length of the historical data, and T_2 is the sequence length of the forecasted data.

Model	GC time complexity	Repitions
GCNConv-GRU	$O(N \times F_{in} \times F_{out} + E \times F_{out})$	T_2
ChebConv-GRU	$O(K \times$ `GCNConv-GRU`$)$	T_2
EdgeConv-GCNConv-GRU	$O(E \times 2 \times F_{in} \times F_{in}$ + `GCNConv-GRU`$)$	T_2
A3TGCN	Same as `GCNConv-GRU`	$3T_1 + 3T_2$

Beside forecasting performance, we also recorded the estimated carbon emissions of all the runs of hyperparameter tuning and of the final training, as seen in figure 11 and 12, respectively. Hyperparameter tuning clearly has the biggest cost of computation and carbon emission, as it accounts for many training runs. Before comparing the numbers between the models, it is worth noting that they have different sizes of hyperparameter search spaces, so some of the models were tuned for more iterations than others. We can, however, see that the GNN-based models have a clear upward trends in terms of carbon emission (and therefore computational cost) as the number of turbines grows. Generally, the longer the models were

trained in the High Performance Cluster (HPC), the more electricity they consumed, and thereby, the more carbon they were estimated to emit.

The `A3TGCN` model has the lowest carbon emission cost, even though it has the most graph convolutions per sample. We can attribute this to the lower amount of hyperparameters searched for when tuning this model. As seen in figure 12, the model has the highest carbon emission across the models when each of them are just trained once.

Interestingly, the hyperparameter tuning of the simple GRU baseline decreased in carbon emission as the number of turbines in the data grew. We hypothesize that this is due to the model being quicker to converge when trained on more turbines with similar temporal patterns and, therefore, requiring fewer epochs on average compared to the smaller subsets.

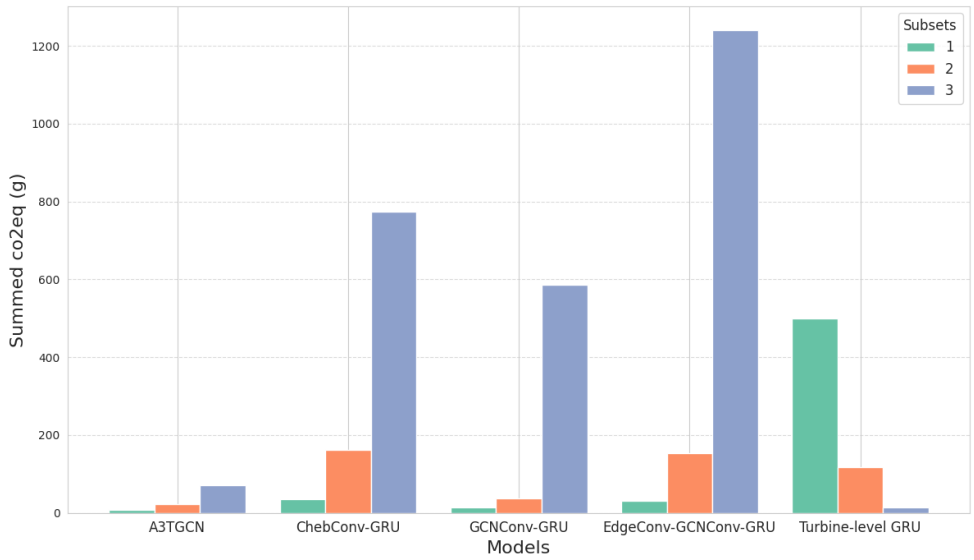

Figure 11: Estimated carbon emissions of hyperparameter tuning of different models, per subset of turbines.

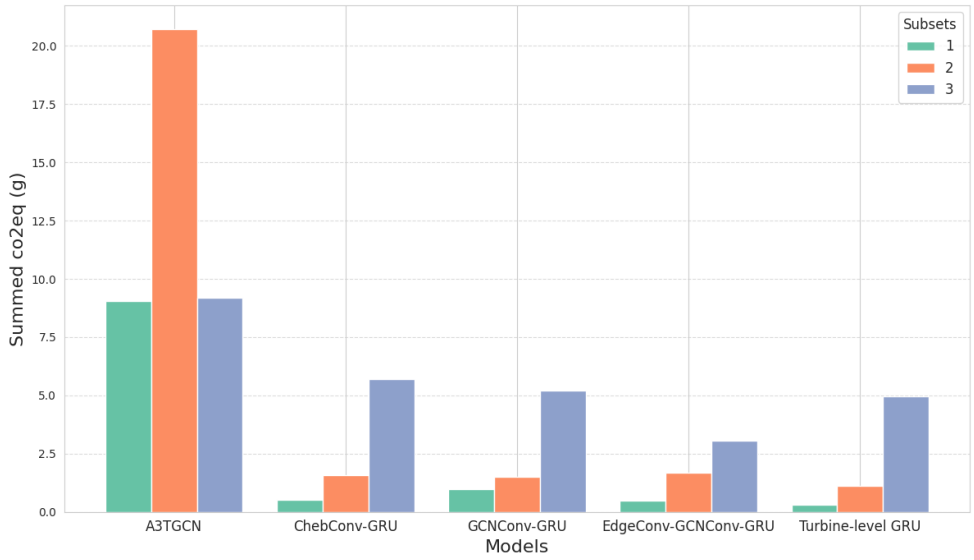

Figure 12: Estimated carbon emissions of training of different models, per subset of turbines.

Other factors in the experimental setup may also be attributed to the lack of performance gain from the GNN layers. For one, and maybe most importantly,

the task itself is hard. Forecasting the individual power outputs of 348 turbines, 24 hours ahead, is a hard task. The data set contains wind turbines from various providers with hidden outlier information, such as when the turbines are down for maintenance. A conservative outlier imputation was done on the training set, while leaving the test set as it was. Ideally, such external information should be included in the data set, so that evaluation can be done more fairly. Further, the hourly aggregation of data might be too sparse to capture short-term variations and patterns in the wind.

In figure 13, we can see how each model performs on each of the subsets, when the RMSE is averaged over each hour in the 24 hours ahead forecasting window. Clearly, all of the models struggle increasingly more as their predictions get further ahead in time. This is not surprising, but it highlights the challenge of the task.

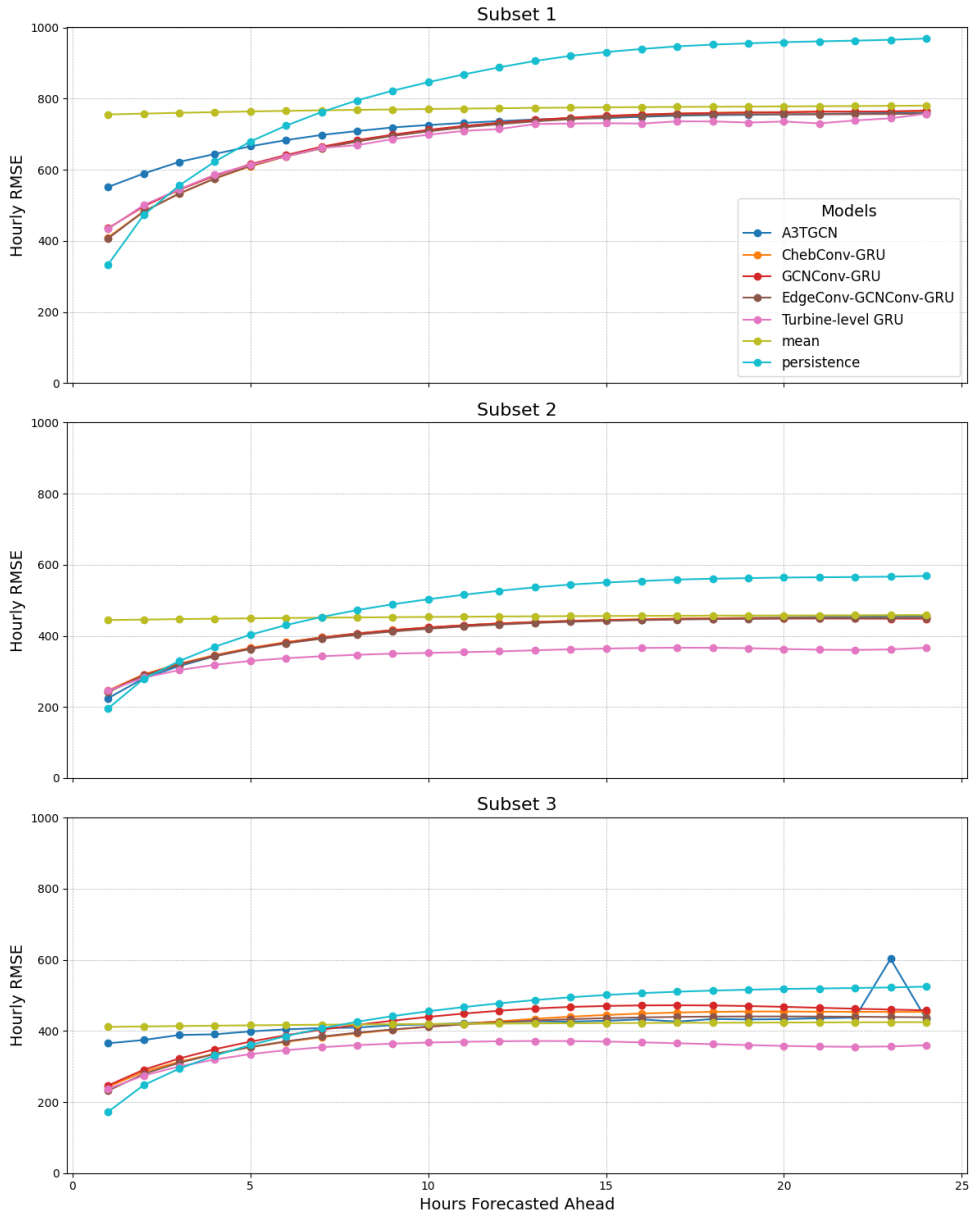

Figure 13: RMSE calculated at each hour for each of the models, in each subset of data.

7 CONCLUSION

This thesis explored the scalability of Graph Neural Networks (GNNs) in wind power forecasting (WPF), comparing their performance, computational complexity, and carbon emissions against simpler baseline models, including a Gated Recurrent Unit (GRU) model. The results demonstrate that while GNNs hold theoretical potential for leveraging spatial relationships between wind turbines, they fall short of outperforming the baseline GRU in terms of forecasting accuracy across all scales of data with our configurations.

The challenges faced by GNNs were particularly evident in the largest dataset, where their performance degraded significantly. This suggests that the approach to spatial modeling, such as the 500-meter distance cut-off for graph edges, may not be sufficient in capturing meaningful relationships between turbines. Adjusting the graph structure, such as increasing the edge cut-off distance, could improve the spatial modeling at the cost of greater computational complexity. Furthermore, the quadratic growth in complexity with respect to the number of turbines in a fully connected graph of turbines underscores the need for careful design to balance model scalability and computational feasibility.

Carbon emission analysis revealed that GNNs generally require more computational resources than simpler models, with emissions scaling alongside the number of turbines. This highlights the environmental trade-offs associated with advanced modeling techniques. However, differences in hyperparameter tuning strategies also played a role, suggesting that efficient tuning protocols could mitigate some of the computational and environmental costs.

Several factors in the experimental setup, including the sparsity of hourly aggregated data, the inherent difficulty of the forecasting task, and the lack of detailed turbine-specific information (e.g., maintenance schedules), further complicated model performance. These limitations highlight the importance of richer datasets and refined experimental designs in future work.

In summary, while GNNs have been promising in the literature and in closed experiments for their ability to incorporate spatial dependencies in WPF, this study finds that they currently do not scale well in performance when applied to larger turbine networks. Future research should focus on optimizing graph configurations, exploring alternative temporal-spatial modeling approaches, and leveraging more comprehensive, real-world datasets. Ultimately, the trade-offs between model complexity, accuracy, and environmental impact must be carefully balanced to advance the practical application of GNNs in wind power forecasting.

BIBLIOGRAPHY

Anthony, L. F. W., B. Kanding, and R. Selvan (2020, July). Carbontracker: Tracking and predicting the carbon footprint of training deep learning models. ICML Workshop on Challenges in Deploying and monitoring Machine Learning Systems. arXiv:2007.03051.

Breunig, M. M., H.-P. Kriegel, R. T. Ng, and J. Sander (2000). Lof: identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, SIGMOD '00, New York, NY, USA, pp. 93–104. Association for Computing Machinery.

Defferrard, M., X. Bresson, and P. Vandergheynst (2016). Convolutional neural networks on graphs with fast localized spectral filtering.

Huang, X., Y. Zhang, J. Liu, X. Zhang, and S. Liu (2023, September). A short-term wind power forecasting model based on 3d convolutional neural network–gated recurrent unit. *Sustainability 15*(19), 14171.

Jiang, J., C. Han, and J. Wang (2023). Buaabigscity: Spatial-temporal graph neural network for wind power forecasting in baidu kdd cup 2022.

Kingma, D. P. and J. Ba (2014). Adam: A method for stochastic optimization.

Kipf, T. N. and M. Welling (2016). Semi-supervised classification with graph convolutional networks.

Park, J. and J. Park (2019). Physics-induced graph neural network: An application to wind-farm power estimation. *Energy 187*, 115883.

Rozemberczki, B., P. Scherer, Y. He, G. Panagopoulos, A. Riedel, M. Astefanoaei, O. Kiss, F. Beres, G. Lopez, N. Collignon, and R. Sarkar (2021). PyTorch Geometric Temporal: Spatiotemporal Signal Processing with Neural Machine Learning Models. In *Proceedings of the 30th ACM International Conference on Information and Knowledge Management*, pp. 4564–4573.

Seo, Y., M. Defferrard, P. Vandergheynst, and X. Bresson (2016). Structured sequence modeling with graph convolutional recurrent networks.

Sun, Y., Q. Zhou, L. Sun, L. Sun, J. Kang, and H. Li (2024). Cnn–lstm–am: A power prediction model for offshore wind turbines. *Ocean Engineering 301*, 117598.

Wang, Y., Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon (2018). Dynamic graph cnn for learning on point clouds.

Wang, Y., R. Zou, F. Liu, L. Zhang, and Q. Liu (2021). A review of wind speed and wind power forecasting with deep neural networks. *Applied Energy 304*, 117766.

Wu, Z., S. Pan, G. Long, J. Jiang, X. Chang, and C. Zhang (2020). Connecting the dots: Multivariate time series forecasting with graph neural networks.

Yu, M., Z. Zhang, X. Li, J. Yu, J. Gao, Z. Liu, B. You, X. Zheng, and R. Yu (2020). Superposition graph neural network for offshore wind power prediction. *Future Generation Computer Systems 113*, 145–157.

Zhou, J., X. Lu, Y. Xiao, J. Su, J. Lyu, Y. Ma, and D. Dou (2022). Sdwpf: A dataset for spatial dynamic wind power forecasting challenge at kdd cup 2022.

Zhu, J., Y. Song, L. Zhao, and H. Li (2020). A3t-gcn: Attention temporal graph convolutional network for traffic forecasting.