Originalni naučni rad

UDK: 161.111:004.032.26:551.509.313.6
BIBLID:0350-8528(2024),24.p.15-36
doi:10.5937/zeint34-51258

A Comparative Analysis of Deep Neural Networks and Gradient Boosting Algorithms in Long-Term Wind Power Forecasting

Luka Ivanović[1], Saša D. Milić[1], Živko Sokolović[1], Aleksandar Rakić[2]

[1]University of Belgrade, Electrical Engineering Institute Nikola Tesla, Koste Glavinića 8a, 11000 Belgrade, Serbia

[2]University of Belgrade, School of Electrical Engineering, Bulevar kralja Aleksandra 73, 11000 Belgrade, Serbia

luka.ivanovic@ieent.org, s-milic@ieent.org, zivko.sokolovic@ieent.org, rakic@etf.bg.ac.rs

Abstract: A vital step toward a sustainable future is the power grid's incorporation of renewable energy sources. Wind energy is significant because of its broad availability and minimal environmental impact. The paper presents a comparative analysis of recurrent neural network algorithms and gradient boosting machines applied to time series data for the regression issue of estimating the active power generated by a wind farm. Gradient boosting algorithms combine the advantages of a few machine learning models (decision trees, random forests, etc.) to produce a powerful prediction model. In addition to conventional recurrent neural networks, the article deals with long short-term memory and gated recurrent unit as cutting-edge models for time series analysis and predictions. A comprehensive analysis was carried out on a large wind power generation data set.

Keywords: Machine Learning, Recurrent Neural Network, Long Short−Term Memory, Gated Recurrent Unit, Gradient Boosting Machines, XGBoost, wind farm, power generation

1. Introduction

Power systems are undergoing a revolution thanks to machine learning, which makes it possible to accurately estimate load, allocate resources optimally, and improve grid stability. Machine learning models predict

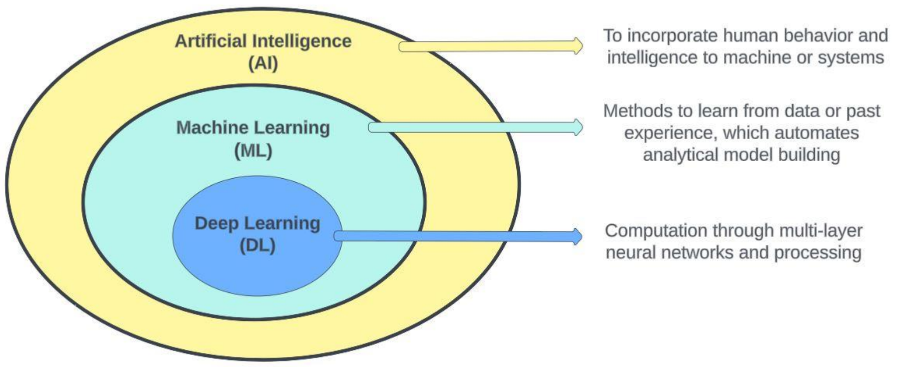
generation from wind energy sources, making it easier to integrate renewable energy sources into the grid and increasing integration efficiency.
The wind power forecast comprises four time scales: ultra-short-term, short-term, medium-term and long-term [1]. An ultra-short-term forecasting is used to predict power production in the next hours, which is useful for managing the daily operations of wind farm (WF) units. Short-term forecasting is useful for making predictions several hours to several days ahead of time, which is beneficial for wind turbine maintenance and the economy as a whole. The medium-term forecast is useful for developing quarterly plans for power generation for grid and wind farm construction, as it provides predictions several days to several months in advance.
In order to support strategic planning and investment decisions for renewable energy projects, long-term wind power forecasting involves prediction wind power generation patterns one to several years into the future. This approach offers insights into the possible expansion and development of wind power infrastructure by accounting for long-term climate trends.
The hands-on machine learning models have replaced and improved existing analyzing and forecasting methods in the power industry in the last few years. The adequate ML model selection is still a demanding research and engineering effort. One of the contributions of this paper is the selection method and practical demonstration of the application of several ML models. In this research, the authors focus on long-term forecasting with two Recurrent Neural Network (RNN) models compared to the Extreme Gradient Boosting (XGBoost) model, which represents an ensemble of the Gradient Boosting Machines (GBM) models, to assess which is more suitable and reliable for long-term wind power forecasting. The results will serve as a foundation for future research and further expansion of knowledge in wind power forecasting.

The paper is organized as follows: Section 2 presents the theoretical framework for deep neural network algorithms, and Section 3 offers insights into ensemble algorithms including GBM and XGBoost. Section 4 provides the comparative results. The discussion, conclusion, and future research are presented in Section 5.

2. Deep Neural Networks (DNN)

Artificial Intelligence (AI), Machine Learning (ML) and Deep Learning (DL) are often used terms to describe intelligent systems or software. Figure 1 shows the current state of DL in relation to ML and AI. According to Figure 1, DL is both of subset of ML and a component of the larger field of AI. In general, AI involves imbuing machines or systems with human behavior and intelligence, whereas ML involves learning from data or experiences and automating the development of analytical models. DL involves extracting information from data using multi-layer neural networks, with “deep” referring to the several levels through which data is processed to create a data-driven model [2].

Thus, DL can be considered as a core technique within AI, representing an edge in artificial intelligence that enables the development of intelligent systems and automation. Importantly, DL improves AI to a new level, often known as “Smarter AI”, and DL approaches play an important role in advanced analytics and intelligent decision-making. Overall, DL algorithms have the ability to significantly change the modern world, particularly in terms of providing a strong computational engine and contributing to technology-driven automation, smart systems, and intelligent decision-making in line with the goals.



Artificial Intelligence (AI)
To incorporate human behavior and intelligence to machine or systems
Machine Learning (ML)
Methods to learn from data or past experience, which automates analytical model building
Deep Learning (DL)
Computation through multi-layer neural networks and processing

Figure 1. A comparison of Deep Learning with Machine Learning and Artificial Intelligence

A typical DNN contains multiple hidden layers including input and output layers. Figure 2 shows a general structure of a DNN (hidden layer = N, N ≥ 2) comparing with a shallow network (hidden layer = 1).

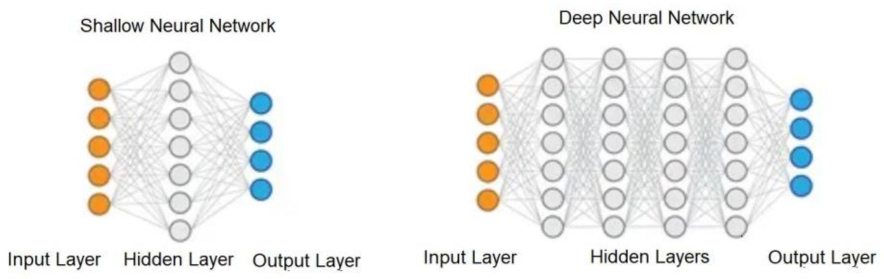


Figure 2. A general architecture of a shallow network with one hidden layer and a deep neural network with multiple hidden layers

2.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are deep learning models that capture sequential dynamics through feedback connections, similar to cycles within a network of nodes. While this approach may appear contradictory at first, the precise description of recurrent edges eliminates ambiguity, unlike in Feedforward Neural Networks (FNNs), where computation order is unambiguous. RNNs are unrolled over time steps, with consistent parameters applied at each step, synchronously propagating standard connections to succeeding layers and dynamically transferring information across adjacent time steps by recurrent connections. RNNs, shown as an unfolded perspective in Fig. 3, can be thought of as feedforward neural networks in which the parameters of each layer (both standard and recurrent) are shared over time steps [3].

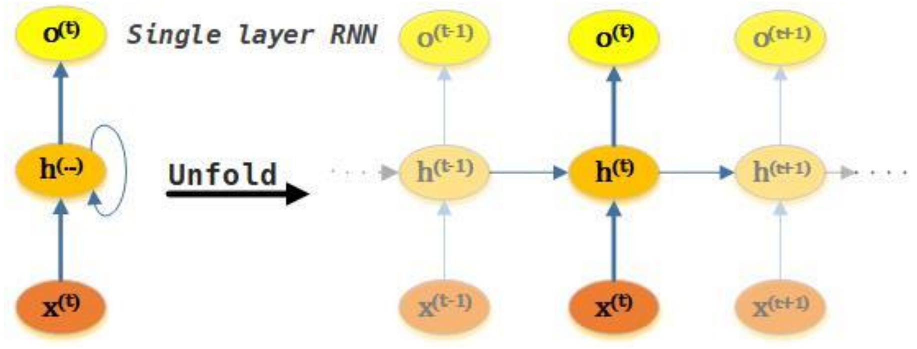


Figure 3. Unfolded RNN

Each hidden unit in a FNNs receives a single input, which is the input layer's net preactivation. In contrast, each hidden unit in a RNN receives two separate inputs: preactivation from the input layer and activation of the same hidden layer from the previous time step, *t-1*. Initially, at time step *t=0*, the hidden units are initialized to zeros or small random numbers. At time steps where *t>0*, the hidden units receive input from the current data point *x(t)*, as well as the previous hidden unit values at *t-1*, denoted as *h(t-1)*.

2.1.1 Computing activations in an RNN

The different weight matrices in a single-layer RNN are as follows (Fig.4):

- $\mathbf{W}_{xh}$: The weight matrix between the input $\mathbf{x}^{(t)}$ and the hidden layer ***h***
- $\mathbf{W}_{hh}$: The weight matrix associated with the recurrent edge
- $\mathbf{W}_{ho}$: The weight matrix between the hidden layer and output layer

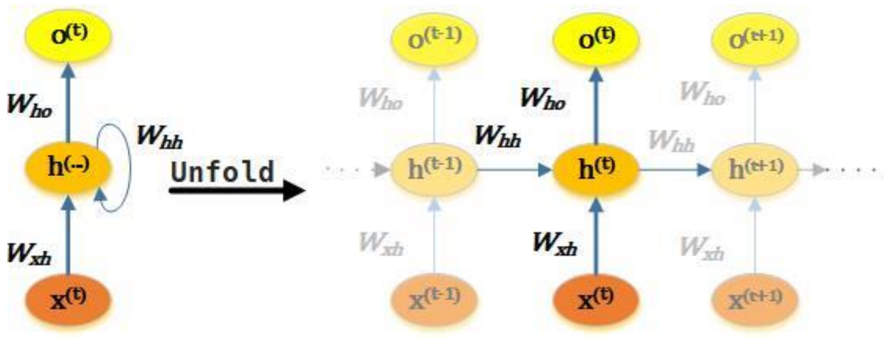


Figure 4. Unfolded RNN with associated weight matrices

Activations are computed in a manner similar to typical multilayer perceptrons and other types of FNNs. For the hidden layer, the net input, $\mathbf{z}_\mathrm{h}$ (preactivation), is computed by a linear combination, that is, simply compute the sum of the multiplications of the weight matrices with the respective vectors and add the bias unit:

$$\boldsymbol{z}_h^{(t)} = \boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h \tag{1}$$

Then, the activations of the hidden units at the time step, t, are calculated as follows:

$$\boldsymbol{h}^{(t)} = \phi_h\left(\boldsymbol{z}_h^{(t)}\right) = \phi_h\left(\boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_h\right) \tag{2}$$

Once the activations of the hidden units at the current time step are computed, then the activations of the output units will be computed, as follows:

$$\boldsymbol{o}^{(t)} = \phi_o\left(\boldsymbol{W}_{ho}\boldsymbol{h}^{(t)} + \boldsymbol{b}_o\right) \tag{3}$$

2.1.2 Training RNNs using backpropagation through time (BPTT)

Backpropagation in RNNs is known as BPTT [4]. This approach needs us to expand or unroll an RNN’s computational graph one step at a time. The unrolled RNN is essentially a FNN with a unique property: the same parameters appear at each time step. Then, just as in any FNN, the chain rule to backpropagate gradients across the unrolled network can be used. The gradient with respect to each parameter must be added across all places where the parameter appears in the unrolled net. If the parameters were time-variant, the model could adapt to changing data patterns but would increase in complexity, making training more difficult and computationally expensive. This could lead to improved performance in non-stationary processes, though algorithms like transformers or attention mechanisms may be better suited for such tasks.

In the following section, we will demonstrate how to compute the gradients of the objective function with respect to all decomposed model parameters. To simplify, we consider an RNN without bias parameters. The activation function in the hidden layer employs the identity mapping ($\phi(x) = x$). For time step t, define the single input and output as $\boldsymbol{x}^{(t)} \in \mathbb{R}^d$ and $\boldsymbol{y}^{(t)}$, respectively. The hidden state $\boldsymbol{h}^{(t)} \in \mathbb{R}^h$ and the output $\boldsymbol{o}^{(t)} \in \mathbb{R}^q$ are computed as follows:

$$\boldsymbol{h}^{(t)} = \boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\boldsymbol{h}^{(t-1)} \tag{4}$$

$$\boldsymbol{o}^{(t)} = \boldsymbol{W}_{ho}\boldsymbol{h}^{(t)} \tag{5}$$

where $\boldsymbol{W}_{xh} \in \mathbb{R}^{h\times d}$, $\boldsymbol{W}_{hh} \in \mathbb{R}^{h\times d}$ and $\boldsymbol{W}_{ho} \in \mathbb{R}^{h\times d}$ are the weight parameters.

Denote by $l(\boldsymbol{o}^{(t)}, \boldsymbol{y}^{(t)})$ the loss at time step t. Our objective function, the loss over T time steps from the beginning of the sequence is thus:

$$L = \frac{1}{T}\sum_{t=1}^{T} l(\boldsymbol{o}^{(t)}, \boldsymbol{y}^{(t)}) \tag{6}$$

To visualize the dependencies between model variables and parameters during RNN computation, we can create a computational graph for the model as shown in Figure 5:

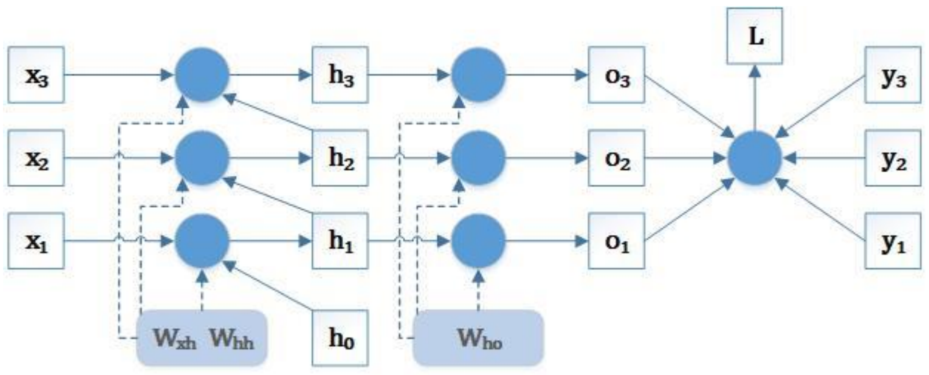


Figure 5. Computational graph showing dependencies for an RNN model with three time steps. Boxes represent variables (not shaded) and circles represent operators

The model depicted in Fig.4 involves parameters denoted as $\mathbf{W}_{\mathbf{xh}}$, $\mathbf{W}_{\mathbf{hh}}$ and $\mathbf{W}_{\mathbf{ho}}$. Usually, gradient corresponding to these parameters: $\boldsymbol{\partial L/\partial W_{xh}}$, $\boldsymbol{\partial L/\partial W_{hh}}$ and $\boldsymbol{\partial L/\partial W_{ho}}$ must be computed in order to train this model.

We can compute and store the gradients progressively by navigating in the opposite direction of the arrows, based on the relationships shown in Fig 3. First, it is quite simple to differentiate the objective function at any time step t, with respect to the model output:

$$\frac{\partial L}{\partial \boldsymbol{o}^{(t)}} = \frac{\partial l(\boldsymbol{o}^{(t)}, \boldsymbol{y}^{(t)})}{T \cdot \partial \boldsymbol{o}^{(t)}} \in \mathrm{R}^{O} \tag{7}$$

We can now compute the gradient of the objective function with respect to the parameter $\mathbf{W}_{\mathbf{ho}}$ in the output layer. According to Fig. 3, the objective function *L* depends $\mathbf{W}_{\mathbf{ho}}$ through ***o*^(1)^, ..., *o*^(T)^**. Employing the chain rule gives us:

$$\frac{\partial L}{\partial \boldsymbol{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{o}^{(t)}} \times \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{W}_{ho}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{o}^{(t)}} {\boldsymbol{h}^{(t)}}^{T} \tag{8}$$

where $\mathbf{\partial L/\partial o^{(t)}}$ is given by (7).

Next, at the final time step *T*, the objective function *L* depends on the hidden state $\boldsymbol{h}^{(T)}$ only via $\boldsymbol{o}^{(T)}$. Therefore, the gradient $\mathbf{\partial L/\partial h^{(t)}} \in \mathbb{R}^{h}$ is:

$$\frac{\partial \mathrm{L}}{\partial \boldsymbol{h}^{(T)}} = \frac{\partial L}{\partial \boldsymbol{o}^{(T)}} \times \frac{\partial \boldsymbol{o}^{(T)}}{\partial \boldsymbol{h}^{(T)}} = {\boldsymbol{W}_{ho}}^{T} \frac{\partial L}{\partial \boldsymbol{o}^{(T)}} \tag{9}$$

The complexity increases for any time step $t < T$ where the objective function *L* relies on $\boldsymbol{h}^{(t)}$ through $\boldsymbol{h}^{(t+1)}$ and $\boldsymbol{o}^{(t)}$**.** By applying the chain rule, the gradient of the hidden state $\mathbf{\partial L/\partial h^{(t)}} \in \mathbb{R}^{h}$ at any time step $t < T$ can be recursively computed as:

$$\frac{\partial \mathrm{L}}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial L}{\partial \boldsymbol{h}^{(t+1)}} \times \frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(T)}} + \frac{\partial L}{\partial \boldsymbol{o}^{(t)}} \times \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} = {\boldsymbol{W}_{hh}}^{T} \frac{\partial L}{\partial \boldsymbol{h}^{(t+1)}} + {\boldsymbol{W}_{ho}}^{T} \frac{\partial L}{\partial \boldsymbol{o}^{(t)}} \tag{10}$$

For analysis, expanding the recurrent computation for any time step $1 \leq t \leq T$ gives:

$$\frac{\partial \mathrm{L}}{\partial \boldsymbol{h}^{(t)}} = \sum_{i=t}^{T} \left({\boldsymbol{W}_{hh}}^{T}\right)^{T-i} {\boldsymbol{W}_{ho}}^{T} \frac{\partial L}{\partial \boldsymbol{o}^{T+t-i}} \tag{11}$$

As can be seen from (11) this straightforward linear example already demonstrates a number of important issues with long sequence models, including the possibility of very large values of ${\boldsymbol{W}_{hh}}^{T}$. Eigenvalues greater than one diverge and those smaller than one disappear in it. This is numerically unstable, which manifests itself in the form of vanishing and exploding gradients.

It is shown that the objective function L relies on the model parameters $\boldsymbol{W}_{xh}$ and $\boldsymbol{W}_{hh}$ in the hidden layer through hidden states $\boldsymbol{h}^{(1)}, \ldots, \boldsymbol{h}^{(T)}$. To compute gradients with respect to these parameters $\partial L/\partial \boldsymbol{W}_{xh} \in \mathbb{R}^{h\times d}$ and $\partial L/\partial \boldsymbol{W}_{hh} \in \mathbb{R}^{h\times h}$, we employ the chain rule, yielding:

$$\frac{\partial L}{\partial \boldsymbol{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}^{(t)}} \times \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{W}_{xh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}^{(t)}} {\boldsymbol{x}^{(t)}}^{T} \tag{12}$$

$$\frac{\partial L}{\partial \boldsymbol{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}^{(t)}} \times \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{W}_{hh}} = \sum_{t=1}^{T} \frac{\partial L}{\partial \boldsymbol{h}^{(t)}} {\boldsymbol{h}^{(t-1)}}^{T} \tag{13}$$

where $\partial L/\partial \boldsymbol{h}^{(t)}$ which is recurrently computed by (9) and (10) is the key quantity that affects numerical stability.

2.2 BPTT gradient challenges

2.2.1 Vanishing gradient problem

The vanishing gradient problem in RNNs is caused by the propagation of gradients across time steps and recurrent connections. Now, let's examine a basic RNN cell that processes a series of inputs across time steps denoted as *t*=1, *t*=2, ..., *t*=*T*.

- The RNN cell computes a new hidden state and produces an output at each time step t by utilizing the previous time step's hidden state in addition to the current input.
- The hidden state at time *t-1* influences the hidden state at time step *t*. This dependency on previous time steps causes the gradient propagation through a chain of recurrent connections.
- The chain rule is used iteratively across the different time stages to calculate gradients during the backpropagation through time process. Starting at the last time step *T*, this iterative process moves backwards.

The vanishing gradient problem in RNNs comes from the fact that the repeated multiplication of gradients at each time step can cause the gradients to become very small as the propagate backwards through time. The gradients may become almost zero if the RNN cell is initialized with minimal weights or the sequence is long. Consequently, during training, the sequence's early time steps receive incredibly weak gradient signals.

Additionally, there is a weak or missing update to their respective weights. This results in the slow convergence of learning and the loss of essential information.

2.2.2 Exploding gradient problem

The exploding gradient problem is a challenge encountered during training DNNs. It happens when the network's loss function's gradients become unnecessarily huge in relation to the weights (parameters).

Exploding gradients are a problem that occurs when derivatives of the layers of the neural network get larger as we go backward during backpropagation. In essence, this is the opposite of the vanishing gradient problem.

The root cause of this problem lies in the weights of the network, rather than the choice of the activation function. High weight values lead to correspondingly high derivatives, causing significant deviations in new weight values from the previous ones. As a result, the gradient fails to converge and can lead to the network oscillating around local minimum, making it challenging to reach the global minimum point.

In summary, exploding gradients occur when weight values lead to excessively large derivatives, making convergence difficult and potentially preventing the neural network from effectively learning and optimizing its parameters.

2.3 Modern RNNs

Modern RNN is a paradigm that symbolizes the ongoing search for better sequential data processing in the field of neural network topologies. In this section, it will be shown some of the well-known RNN architectures, such as *Long Short-Term Memory* (LSTM) and *Gated Recurrent Unit* (GRU).

Conventional RNN, LSTM, and GRU are all architectures designed for processing sequential data and handling data sequences by hidden state that memories temporal information.

2.3.1 Long Short-Term Memory (LSTM)

LSTM is an improved RNN model, explicitly designed to address the challenge of capturing long-term dependencies in sequential data. LSTM introduces a gating mechanism comprising input, forget, and output gates, enabling the network to retain or discard information over extended sequences more effectively. LSTM has found extensive applications in natural language processing tasks, including language modeling, machine translation, and sentiment analysis, where understanding context and temporal dependencies is crucial [6].

In short, the LSTM architecture consists of a set of recurrently connected sub-networks, known as memory blocks. The idea behind the memory block is to maintain its state over time and regulate the information flow through non-linear gating units. Fig. 6 shows the architecture of a LSTM block, which involves the gates, the input signal $\boldsymbol{x}^{(t)}$, the output $\boldsymbol{h}^{(t)}$, the activation functions and connections. The output of the block is recurrently connected back to the block input and all of the gates [7].

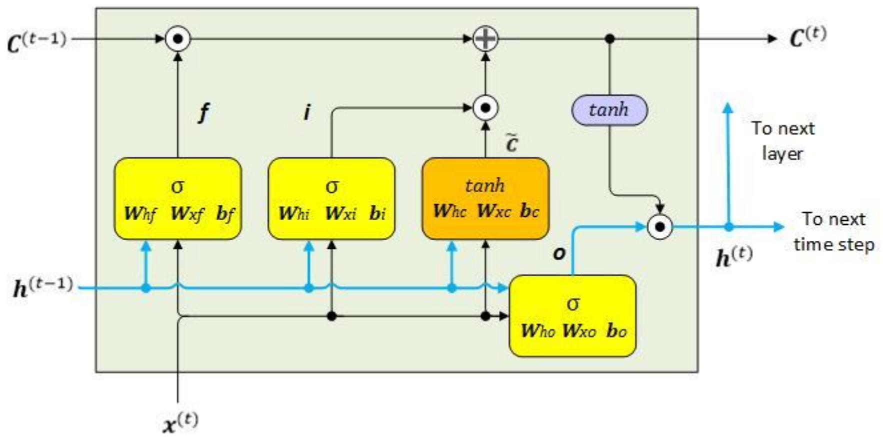


Figure 6. Architecture of a typical LSTM block

Cell state from the previous time step, $\boldsymbol{C}^{(t-1)}$, is modified to get the cell state at the current time step, $\boldsymbol{C}^{(t)}$, without being multiplied directly with any weight factor. The flow of information in this memory cell is controlled by several computation units. In the previous Fig, $\odot$ refers to the element-wise product and $\oplus$ means element-wise summation. Four boxes are indicated with an activation function, either the sigmoid function (σ) or tanh, and a set of weights. These boxes apply a linear combination by performing matrix-vector multiplications on their inputs. These units of computation with sigmoid activation functions, whose output units are passed through $\odot$, are called gates.

In an LSTM cell, there are three different types of gates, which are known as the forget gate, the input gate, and the output gate:

- The **forget gate** $(\boldsymbol{f}_t)$ allows the memory cell to reset the cell state without growing indefinitely. In fact, the forget gate decides which information is allowed to go through and which information to suppress. Now, $(\boldsymbol{f}_t)$ is computed as follows:

$$\boldsymbol{f}_t = \sigma\left(\boldsymbol{W}_{xf}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hf}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_f\right) \tag{14}$$

- The **input gate** $(\boldsymbol{i}_t)$ and **candidate value** $\widetilde{\boldsymbol{C}}_t$ are responsible for updating the cell state. They are computed as follows:

$$\boldsymbol{i}_t = \sigma\left(\boldsymbol{W}_{xi}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hi}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_i\right) \tag{15}$$

$$\widetilde{\boldsymbol{C}}_t = \tanh\left(\boldsymbol{W}_{xc}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hc}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_c\right) \tag{16}$$

The cell state at time *t* is computed as follows:

$$\boldsymbol{C}^{(t)} = \left(\boldsymbol{C}^{(t-1)} \odot \boldsymbol{f}_t\right) \oplus \left(\boldsymbol{i}_t \odot \widetilde{\boldsymbol{C}}_t\right) \tag{17}$$

- The **output gate** $(\boldsymbol{o}_t)$ decides how to update the values of hidden units:

$$\boldsymbol{o}_t = \sigma\left(\boldsymbol{W}_{xo}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{ho}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_o\right) \tag{18}$$

Given this, the hidden units at the current time step are computed as follows:

$$\boldsymbol{h}^{(t)} = \boldsymbol{o}_t \odot \tanh\left(\boldsymbol{C}^{(t)}\right) \tag{19}$$

2.3.2 Gate Recurrent Unit (GRU)

Because back-propagation in LSTM involves a high number of parameters, computation times are long. In [13], Cho et al. proposed the GRU, which has fewer gates than LSTM, to shorten computation time. Although the design of the GRU has been modified, its value is comparable to that of the LSTM. Figure 7 shows the graphical representation for GRU. Similar to LSTM, GRU uses gating units to capture long-term dependencies in order to overcome the vanishing and exploding gradient problem.

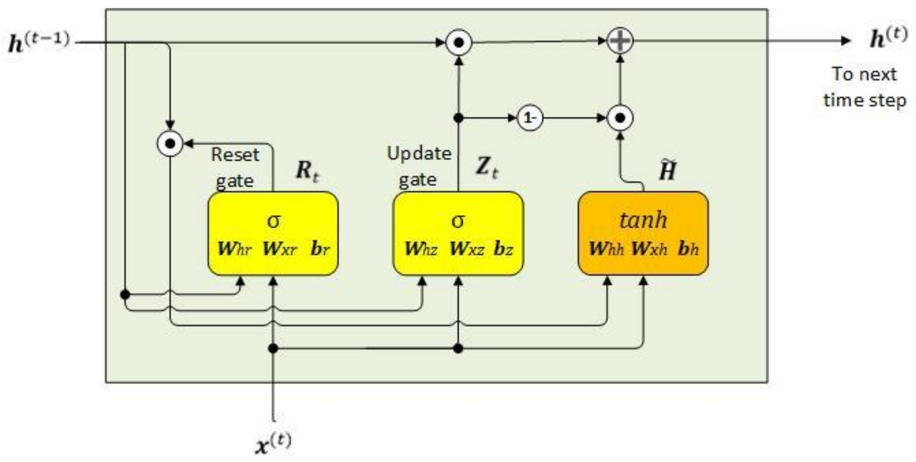


Figure 7. GRU block architecture

The GRU unit is defined by the below set of equations. In them $\widetilde{\boldsymbol{H}}_t$ stands for the hidden state candidate.

$$\boldsymbol{R}_t = \sigma\left(\boldsymbol{W}_{xr}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hr}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_r\right) \tag{20}$$

$$\boldsymbol{Z}_t = \sigma\left(\boldsymbol{W}_{xz}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hz}\boldsymbol{h}^{(t-1)} + \boldsymbol{b}_z\right) \tag{21}$$

$$\widetilde{\boldsymbol{H}}_t = \tanh\left(\boldsymbol{W}_{xh}\boldsymbol{x}^{(t)} + \boldsymbol{W}_{hh}\left(\boldsymbol{R}_t \odot \boldsymbol{h}^{(t-1)}\right) + \boldsymbol{b}_h\right) \tag{22}$$

$$\boldsymbol{h}^{(t)} = \left(\boldsymbol{h}^{(t-1)} \odot \boldsymbol{Z}_t\right) \oplus (\boldsymbol{1} - \boldsymbol{Z}_t) \odot \widetilde{\boldsymbol{H}}_t \tag{23}$$

3. Ensemble Learning

Ensemble algorithms are becoming more and more popular in the field of machine learning because of their capacity to improve predictive performance and robustness. The ensemble methods that are most frequently used are: Random Forests, GBM, Adaptive Boosting (AdaBoost), XGBoost and Bootstrap Aggregating (Bagging). This paper will focus on XGBoost's performance.

Decision trees are machine learning algorithms frequently used for both classification and regression problems. They are hierarchical structures that present a sequence of decisions based on the features of the input data. Every leaf node in the tree indicates the expected result, and every internal node represents a decision made in response to a specific feature. Recursively dividing the feature space according to the feature that provides the optimal split is the process of creating a decision tree. This method generates data subsets that are progressively uniform in relation to the target variable. Until a stopping condition is satisfied, like reaching a maximum depth or getting minimum number of samples in each leaf node, this splitting process keeps going. Decision trees are useful for understanding feature importance and decision making processes since they are simple to read and show. Decision trees also have the ability to independently choose features and find feature connections, and they can handle both numerical and categorical data.

But decision trees can overfit, particularly if they develop too deeply or if there are noise in the data. Techniques like pruning, limiting the maximum depth of the tree or using ensemble methods like GBM can be used to solve this problem.

3.1 Gradient Boosting

Boosting algorithms use an iterative process to combine weak learners, learners who are slightly better than random, into strong learners [7]. Gradient boosting is a boosting-like algorithm for regression problems. Given a training dataset *D* = $\{x_i, y_i\}_1^N$, the goal of gradient boosting is to find an approximation, $\hat{F}(x)$, of the function $F^*(x)$, which maps instances *x* to their output values *y*, by minimizing the expected value of a given loss function, *L(y,F(x))* [8]. Gradient boosting builds an additive approximation of $F^*(x)$ as a weighted sum of functions:

$$F_m(x) = F_{m-1}(x) + \rho_m h_m(x) \quad (24)$$

where ρ_m is the weight of the m^{th} function, $h_m(x)$. These functions are the models of the ensemble (e.g. decision trees). The approximation is constructed iteratively. First, a constant approximation of $F^*(x)$ is obtained as:

$$F_0(x) = \underset{\alpha}{argmin} \sum_{i=1}^{N} L(y_i, \alpha) \quad (25)$$

Subsequent models are expected to minimize:

$$(\rho_m, h_m) = \underset{\rho,h}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, F_{m-1}(x_i) + \rho h(x_i)) \quad (26)$$

However, instead of solving the optimization problem directly, each h_m can be seen as a greedy step in a gradient descent optimization for F^*. For that, each model h_m is trained on a new dataset $D = \{x_i, r_{mi}\}_{i=1}^{N}$ where the pseudo residuals, r_{mi}, are calculated by:

$$r_{mi} = \left[\frac{\partial L(y_i, F(x))}{\partial F(x)}\right]_{F(x)= F_{m-1}(x)} \quad (27)$$

The value of ρ_m is subsequently computed by solving a line search optimization problem.

This algorithm may encounter overfitting if the iterative process lack proper regularization. In some cases, if the model h_m fits the pseudo-residuals perfectly, the next iteration may generate zero pseudo-reisudals, prematurely terminating the process. To regulate the additive nature of gradient boosting, various regularization parameters are used. The intuitive way to regularize gradient boosting is to apply shrinkage to reduce each gradient decent step:

$$F_m(x) = F_{m-1}(x) + v\rho_m h_m(x) \quad (28)$$

with $v = (0, 1]$. The value of v is usually set to 0.1.

Furthermore, additional regularization can be attained by constraining the complexity of trained models. In the case of decision trees, this may involve setting limitations on the trees' depth or the bare minimum of instances needed to divide a node. Additionally, another set of parameters that are integrated into different versions of gradient boosting include those that provide randomness to the base learners, which improves ensemble generalization even more, such as random subsampling without replacement.

3.2 XGBoost

XGBoost is a highly efficient and scalable implementation of gradient boosting. It is widely regarded as one of the most powerful and effective machine learning algorithms. One of the key characteristics of XGBoost is its ability to handle missing data effectively through robust algorithms. It introduces several optimizations, including parallel computing, tree pruning, and regularization techniques, to enhance training speed and model performance [9]. The loss function used to control the complexity of the trees is:

$$L = \sum_{i=1}^{N} L(y_i, F(x_i)) + \sum_{m=1}^{M} \Omega(h_m) \quad (29)$$

$$\Omega(h) = \gamma T + \frac{1}{2}\lambda\|\omega\|^2 \quad (30)$$

where *T* is the number of leaves of the tree and ω are the output scores of the leaves. Pre-pruning can be achieved by integrating this loss function with the split criterion of decision trees. Trees with higher values of γ are simpler. The minimal loss reduction gain required to separate an internal node is determined by the value of γ. Also, shrinkage is an extra regularization parameter in

XGBoost that reduces the additive expansion's step size. Lastly, other strategies like the depth of the trees, etc., can also be used to limit the complexity of the trees. Tree complexity reduction also leads to shorter training times and reduced storage space needed for the models.

3.2.1 XGBoost Parameters Tuning

The overall parameters have been divided into 3 categories:

1) General Parameters – define the overall functionality of XGBoost

- booster – select the type of the model, tree-based or linear model
- nthread – used for parallel processing, number of available threads

2) Booster Parameters – tuning the individual booster/ tree at each step
3) Learning Task Parameters – define the optimization objective and the metric to be calculated at each step

- objective – define the loss function to be minimized (for regression it is used squarederror, squaredlogerror,logistic, absoluteerror)
- eval_metric – used for validation data (typical values: root mean square error, mean absolute error, negative log-likelihood, multiclass classification error rate, area under the curve)

Booster parameters are shown in Table 1, and algorithm used for fine tuning parameters is shown in Fig. 8.

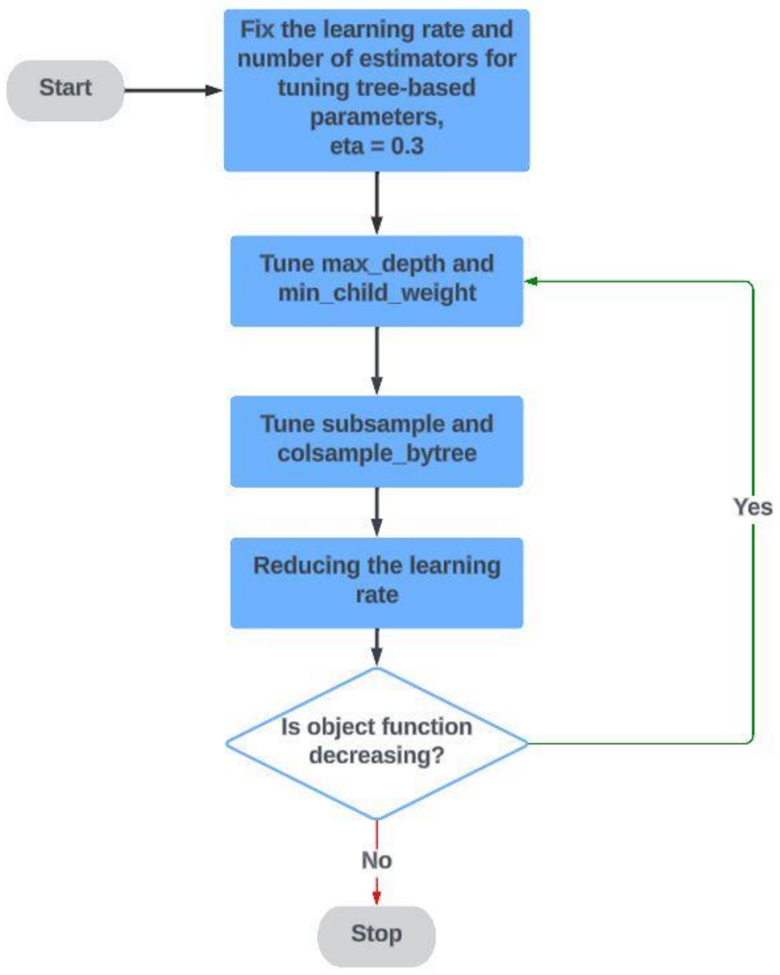

Start
Fix the learning rate and number of estimators for tuning tree-based parameters, eta = 0.3
Tune max_depth and min_child_weight
Tune subsample and colsample_bytree
Reducing the learning rate
Is object function decreasing?
Yes
No
Stop

Figure 8. An algorithm used for XGBoost Parameters Tuning

Table 1: Booster Parameters for XGBoost Parameters Tuning

Booster Parameter	**Description**	**Typical Values**
eta	Analogous to GBM learning rate.	0.01-0.2
min_child_weight	The minimum sum of weights of observations required in a child.	Tuned used cross-validation
max_depth	The maximum depth of a tree. Used to control over-fitting.	3-10
max_leaf_nodes	The maximum number of terminal nodes or leaves in a tree.	
gamma	Specifies the minimum loss reduction required to make a split.	Tuned depending on loss function
subsample	Denotes the fraction of observations to be random samples for each tree.	0.5-1
colsample_by_tree	Denotes the fraction of columns to be random samples for each tree.	0.5-1
lambda	L2 regularization term on weights (analogous to Ridge regression).	For reducing overfitting
alpha	L1 regularization term on weight (analogous to Lasso regression).	For high dimensionality

4. Results and Discussion

This paper uses time series dataset to examine how well ensemble algorithm XGBoost performs with RNN models [10,11]. The dataset used in this research was obtained from the „Wind Power Generation Data – Forecasting“ Kaggle dataset [12].

The input variables utilized for our models include temperature (in degrees Fahrenheit) at a height of 2 meters above the surface, relative humidity (expressed as a percentage) at the same height, wind speed (in meters per second) at both 10 and 100 meters above the surface, wind direction (in degrees, ranging from 0 to 360) at both heights, and wind gusts (in meters per second) at 100 meters above the surface. Our target output is the turbine output, which has been normalized to a range between 0 and 1.

The five years included in the wind power forecasting dataset are 2017 through 2021. The training phase used data from the first four years, while the fifth year served for testing the models. The implementation used the Tensorflow and Keras libraries to build RNN models, as well as the XGBoost library to create XGBoost models.

Firstly, we identified the best model among various RNN architectures. The experiment was performed with Simple RNN, LSTM, and GRU models. Additionally, we also explored a few configurations with different numbers of layers and epochs. The models were trained with and without dropout layers using dropout probabilities (0.2 and 0.3) to evaluate the impact of regularization. The dropout layer is a paradigm for regularization technique for preventing neural network overfitting by randomly setting a percentage of input units to zero during training. This causes the network to learn more robust features that are not dependent on individual neurons. The training time for each model has been measured. The results for each RNN model are presented in Table 2.

Table 2: The results for the various RNN models

NN architecture + Number of Epochs	Model type	Without Regularization		With Regularization (Dropout = 0.2)		With Regularization (Dropout = 0.3)	
		MSE	Training Time [s]	MSE	Training Time [s]	MSE	Training Time [s]
1 hidden layer with 64 neurons, 10 Epochs	RNN	0.3491	20.89	0.1703	15.93	0.1951	14.98
	LSTM	0.1779	47.01	0.1716	47.78	0.1730	49.43
	GRU	0.1955	40.31	0.1676	29.09	0.1737	35.41
1 hidden layer with 64 neurons, 100 Epochs	RNN	0.1734	182.69	0.1664	186.14	0.1657	181.52
	LSTM	0.1716	260.17	0.1675	222.59	0.1641	223.85
	GRU	0.1774	222.05	0.1686	206.06	0.1701	212.32
2 hidden layers with 64 neurons, 10 Epochs	RNN	0.2013	26.28	0.1994	24.67	0.1762	30.77
	LSTM	0.1823	74.49	0.1693	55.51	0.1706	52.65
	GRU	0.1856	70.99	0.1673	49.49	0.1767	42.42
2 hidden layers with 64 neurons, 100 Epochs	RNN	0.1787	289.09	0.1738	250.47	0.1703	253.28
	LSTM	0.1801	431.623	0.1730	407.57	0.1698	375.02
	GRU	0.1750	391.01	0.1687	361.99	0.1712	337.39

As we can see, the best results in terms of mean squared error (MSE=0.1641) was achieved by the LSTM model with one hidden layer of 64 nodes, trained for 100 epochs. Table 2 also shows that the NN with a dropout layer performs better than the one without, as it helps prevent overfitting. Furthermore, the training time for GRU models is consistently shorter than that for LSTM models, which is understandable given the simpler structure of the GRU model compared to the LSTM.

In Fig. 9 we can observe the performance of the aforementioned LSTM model by comparing predicted with actual values of active power for the year 2021.

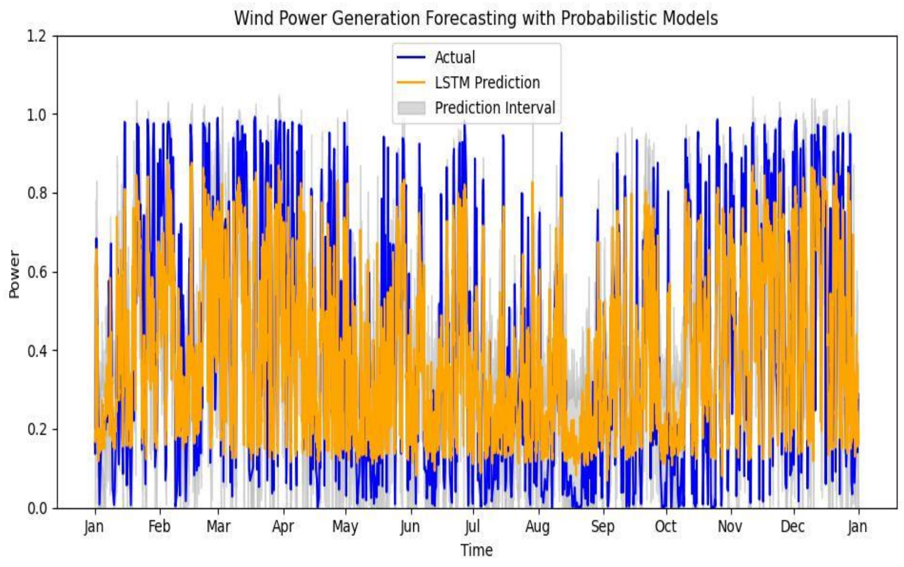


Figure 9. The performance of the best RNN (LSTM) model

After that, the authors attempted to find the best XGBoost model using the algorithm presented in Fig 8. Grid search was used as the hyperparameter tuning technique. In the first iteration, the optimal values for 'max_depth' and 'min_child_weight' were identified. Following this, the optimal values for 'subsample' and 'colsample_bytree' were determined, and the learning rate parameter 'eta' was subsequently reduced. In the second iteration, the entire process was repeated for the new learning rate to find the optimal parameters.

Additional parameters used in the XGBoost model include the number of boosting rounds, which was set to 10,000, 'verbose_eval' set to 50 (indicating that a message will be displayed every 50 rounds), and 'early_stopping_rounds' set to 50 (indicating that the training process will stop if the objective function does not change for 50 consecutive rounds). The entire parameter tuning process is represented in Table 3.

As we can see, parameter tuning resulted in an improvement of approximately 11%. In Fig. 10, the performance of the best XGBoost model is shown by comparing predicted and actual values of active power for the year 2021.

Table 3: The parameters tuning process for XGBoost

XGBoost	Mean Squared Error	
Without Parameters Tuning	0.1679	
With Parameters Tuning	**First Iteration**	MSE
	’max_depth’ in the range of values (2,12), ’min_child_weight’ in the range of values (2,8)	0.1538
	Best results: ’max_depth’ = 8, ’min_child_weight’ = 7	
	’subsample’ in the range of the values (0.7,1), ’colsample_bytree’ in the range of values (0.7,1)	0.1538
	Best results: ’subsample’ = 1, ’colsample_bytree’ = 1	
	’eta’ in the range of values (0.3,0.25,0.2,0.15,0.1,0.05,0.01,0.005)	0.1514
	Best results: ’eta’ = 0.05	
	Second Iteration	MSE
	’max_depth’ in range (2,12), ’min_child_weight’ in range (2,8)	0.1506
	Best results: ’max_depth’ = 11, ’min_child_weight’ = 7	
	’subsample’ in range (0.7,1), ’colsample_bytree’ in rangee (0.7,1)	**0.1496**
	Best results: ’subsample’ = 0.7, ’colsample_bytree’ = 1	

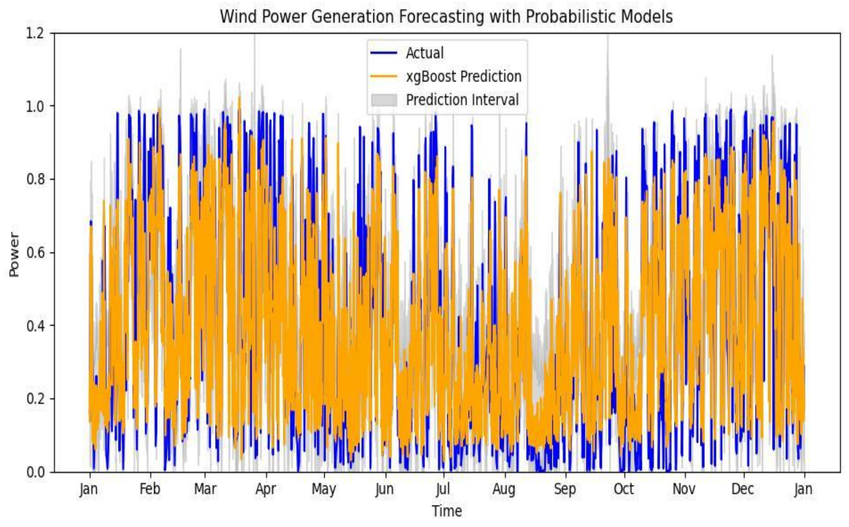


Figure 10. The performance of the best XGBoost model

Our analysis demonstrated that both models are capable of capturing the complex temporal dependencies inherent in wind power generation data. However, XGBoost consistently outperformed RNNs, providing slightly better predictive accuracy across various performance metrics, including MSE. The superior performance of XGBoost can be attributed to its ability to handle non-linearity, missing data and outliers more effectively, while requiring less hyperparameter tuning than RNNs. On the other hand, RNNs, despite their potential to model temporal dependencies through recurrent connections, showed limitations in convergence speed and sensitivity to training data size and sequence length. The findings suggest that for this specific multivariate time series forecasting problem, XGBoost is more efficient and reliable approach.

5. Conclusion

This paper underscores the necessity of analyzing a variety of ML models to address the needs of modern power systems expanded with wind farms as renewable sources. The applications of wind farms within power systems necessitate the comprehensive analysis of multiple types of ML models to harness their potential.

One of the key strategies in power system forecasting, control, and management is time series analysis. The choice of model for forecasting can significantly impact the accuracy and efficiency of predictions.

In this paper, the authors compared RNN algorithms with GBM algorithms. The results indicate that GBM algorithms with tuned parameters slightly outperform DNN models. This is expected given the limited dataset, which likely prevents the DNNs from fully utilizing their ability to learn complex features. Nonetheless, the DNN models still perform well, suggesting they could achieve better results with larger datasets.

While RNNs are foundational, LSTMs and GRUs offer more sophisticated mechanisms for handling time series data. In addition, XGBoost can be powerful when the data is suitably transformed and can capture complex patterns that recurrent networks might miss.

In future work, the authors plan to enhance the prediction models using state-of-the-art algorithms for multivariate time series, such as transformers. They also aim to apply these models to real-world systems and optimize their performance.

6. Acknowledgment

This work was supported in part by the Ministry of Science, Technological Development and Innovation of the Republic of Serbia under the Contract on the realization and financing of the scientific research work of Research and Innovation Organizations in 2024.

References

[1] Z. Tian, "A State-Of-The-Art Review on Wind Power Deterministic Prediction," *Wind Engineering*, vol. 45, pp. 1374–1392, 2021.

[2] I. H. Sarker, "Deep Learning: A Comprehensive Overview on Techniques, Taxonomy, Applications and Research Directions," *SN Computer Science*, vol. 2, 420, 2021.

[3] A. Zhang, Z. C. Lipton, M. Li, A. J. Smola, *Dive into Deep Learning*. 2023. [Online]. Available: https://d2l.ai/

[4] P. J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550-1560, 1990.

[5] I. Danish & Abbas, Asad. (2024). A Deep Dive into Neural Networks: Architectures, Training Techniques, and Practical Implementations. *Journal of Environmental Sciences and Technology*, vol. 02, pp. 61−71, 2023. https://doi.org/10.13140/RG.2.2.14866.84162

[6] G. Van Houdt, C. Mosquera, G. Nápoles, "A Review on the Long Short-Term Memory Model," *Artificial Intelligence Review*, vol. 53, pp. 5929−5955, 2020.

[7] Y. Freund, R. E. Schapire, "A Short Introduction to Boosting," *Journal of Japanese Society for Artificial Intelligence*, vol. 14, pp. 771−780, 1999.

[8] J. H. Friedman, "Greedy function approximation: a Gradient Boosting machine," *The Annals of Statistics*, vol. 29, pp. 1189–1232, 2001.

[9] T. Chen, C. Guestrin. "Xgboost: A scalable tree boosting system," in *Proc. 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '16*, (New York, NY, USA, 13 Aug. 2016), p.p. 785–794.

[10] A. Mobarak, A. Y. Owda, M. Owda, "Electrical Load Forecasting Using LSTM, GRU, and RNN Algorithms," *Energies*, vol. 16, 2283, 2023.

[11] M.Chen, et al., "XGBoost-Based Algorithm Interpretation and Application on Post-Fault Transient Stability Status Prediction of Power System", *IEEE Access*, vol. 7, pp. 13149−13158, 2019.

[12] Data available: https://www.kaggle.com/datasets/mubashirrahim/wind-power-generation-data-forecasting/data

[13] J. Chung, et al., "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling," 2015. [Online]. Available: https://doi.org/10.48550/arXiv.1412.3555

Kratak sadržaj: Važan korak ka održivoj budućnosti je uključivanje obnovljivih izvora energije u elektroenergetsku mrežu. Energija vetra je značajna zbog svoje široke dostupnosti i minimalnog uticaja na životnu sredinu. U radu je prikazana komparativna analiza rekurentnih algoritama neuronske mreže i mašina za povećanje gradijenta primenjenih na podatke vremenske serije za regresivne procene aktivne snage koju generiše vetropark. Algoritmi sa pojačanjem gradijenta kombinuju prednosti nekoliko modela mašinskog učenja (stabla odlučivanja, nasumične šume, itd.) da bi proizveli moćan model predviđanja. Pored konvencionalnih rekurentnih neuronskih mreža, članak se bavi dugotrajnom kratkoročnom memorijom i taktovanom rekurentnom neuralnom jedinicom kao najsavremenijim modelima za analizu vremenskih serija i predviđanja. Sveobuhvatna analiza je sprovedena na velikom skupu podataka o proizvodnji energije iz vetroelektrana.

Ključne reči: Mašinsko učenje, rekurentna neuronska mreža, dugotrajna kratkoročna memorija, zatvorena ponavljajuća jedinica, mašine za povećanje gradijenta, XGBoost, vetropark, proizvodnja energije.

Komparativna analiza dubokih neuronskih mreža i algoritama sa pojačanjem gradijenta u dugoročnom predviđanju snage vetra

Luka Ivanović, Saša D. Milić, Živko Sokolović, Aleksandar Rakić

Rad primljen u uredništvo: 28.05.2024. godine.

Rad prihvaćen: 02.10.2024. godine.