Chapter 4
Energy Demand Forecasting Using Deep Learning

Bahrudin Hrnjica and Ali Danandeh Mehr

4.1 Introduction to Machine Learning

Systems of intelligent behavior have been the subject of interest for scientists over the last few decades. They have tried to integrate intelligence through adaption, learning, autonomy, and solving complex problems. Such research led to the emergence of a new scientific field that is now called artificial intelligence (AI) [28]. AI can be described as an action performed by a machine that can be characterized as intelligent, since if a human had to apply the same action, intelligence must be used to achieve the same goal. When AI is used, it makes possible for machines to use the experience for learning. Once it collects enough experiences, it is capable of producing the output for the new set of inputs in the similar way a human does. AI is a wide scientific field as the intelligence can be applied in various ways. Two related scientific fields which are closely related to AI are statistics and computer science. This is obvious, since one needs statistics in order to define and describe associated algorithms, and computer science is needed to translate the algorithms into a machine language in order to perform actions. Figure 4.1 shows the position of AI in relation to computer science and statistics with related applications.

ML is one of the main components of AI. It is a set of learning computer algorithms by which machines or computers can learn without explicitly being programmed. Moreover, ML can be defined as the field of AI which provides the algorithms for machines to automatically learn and improve their actions from a

B. Hrnjica (✉)
University of Bihac, Bihac, Bosnia and Herzegovina
e-mail: bahrudin.hrnjica@unbi.ba

A. Danandeh Mehr
Antalya Bilim University, Antalya, Turkey
e-mail: ali.danandeh@antalya.edu.tr

F. Al-Turjman (ed.), *Smart Cities Performability, Cognition, & Security*,
EAI/Springer Innovations in Communication and Computing,
https://doi.org/10.1007/978-3-030-14718-1_4

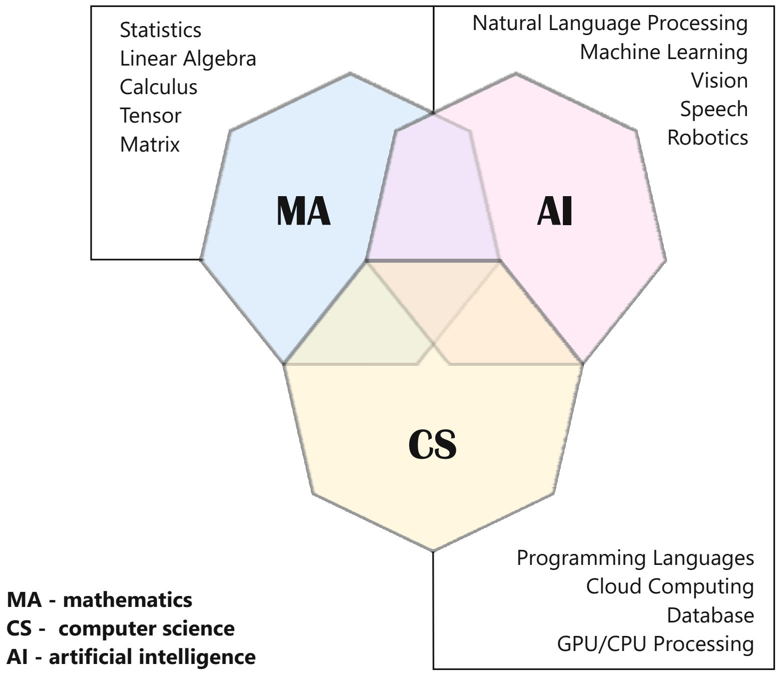


Fig. 4.1 AI as scientific filed in relation with statistics (mathematics) and computer science

given experience. While AI is defined as the ability to acquire and apply knowledge, ML is defined as the acquisition of knowledge or skill. On the other hand, using AI one tries to increase the chance of success but not accuracy. However, in ML one tries to increase the accuracy of the action regardless of the success. Last but not least, AI can be defined as a smart computer program, while ML is the concept of how machines use data to learn. As stated previously, ML is a set of computer algorithms, particularly designed for machines to help them in the learning process. Usually, an ML process consists of searching the data to recognize hidden patterns in the data. Once the patterns are recognized, the computer can make predictions for new or unseen data based on persisted knowledge. Supervised, unsupervised, and reinforcement learning are basically the three main types of the ML (Fig. 4.2):

In supervised ML, the learning process consists of finding the rule that maps inputs (features) to outputs (labels). During the learning process, available data can be divided into two sets. The first set is the training set which is responsible for the training process. The second set is called validation or testing set and is used by the learning algorithm to verify the training process.

Unsupervised learning is the process of discovering patterns in data without defined output (unlabeled dataset). With unsupervised learning, the correct result cannot be determined because no output variable is defined. Algorithms are left to their capability to discover as much knowledge as possible from the data. Inasmuch

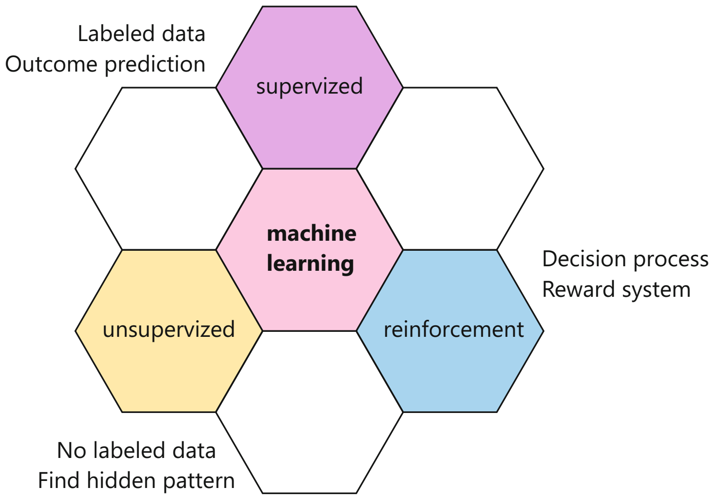


Fig. 4.2 Machine learning types

as there is no output variable, there is no need for splitting the data into training and testing sets. Thus, all the samples are used for training process. This kind of learning can be applied in image and signal processing, computer vision, etc.

Reinforcement learning is an ML process where a computer interacts with a dynamic system in which it must achieve a goal (like driving a vehicle or playing a game). Reinforcement learning provides feedback that consists of information showing how the last action was treated, was it successful or unsuccessful. Based on the feedback, the computer can learn and make further decisions.

Supervised ML can be classified by the type of the output as regression or classification. In regression the output is represented as a continuous number, while in classification, the output variable is discrete rather than continuous, and consists of two or more classes.

Regression ML considers finding relations between one or more variables which are called features, and then compared with a dependent variable called the label. Figure 4.3 shows how ML can be classified depending on its learning types.

ML-originated regression models are used in different disciplines such as finance, production, the stock market, and maintenance. The models can later be used for predicting or forecasting sales, weather temperature for the next day, stock market prices in the next few hours, energy consumption for a given time period, etc.

In most cases, a dataset is determined by time so that each dataset value has a defined timestamp. In this way the history of a dataset value can be monitored, which can be valuable for future business decisions. This kind of dataset is called time series data. In one example, a stock price is a set of observed values recorded in time, so for each time value (minutes, hours, or days) a stock price is calculated. In a second example daily energy consumption is collected so that at the end of each

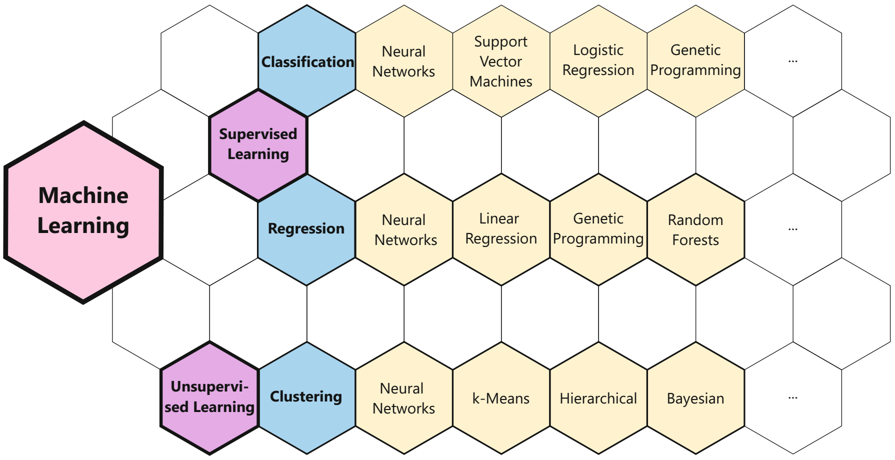


Fig. 4.3 Different type of ML algorithms

day it can be recorded. Regression type of ML covers broad applications. One of the most complex types of ML are time series events.

The forecasting of time series events is an important area of ML, since there are so many forecasting tasks that involve a time component. The time component adds additional information to the time series data, but it also makes time series problems more difficult to handle in comparison to many other regression tasks. Predictions of the time series events are still one of the most difficult tasks, and are active research subjects of many engineers and scientists. Time series events can be detected around us, and the prediction of its future states is of tremendous importance. For example, it is of crucial importance for the world economy to forecast the price of energy [25], sales [13], or stock prices [17]. Furthermore, time series events can be used to predict the weather and environmental, hydrological, and geological events [4, 6, 18]. Smart cities need better and smarter surveillance cameras [22], digital surveillance systems with different frameworks [23, 32], and 5G-inspired IIoT paradigm in health care [1].

In this chapter, the application of deep learning is used in order to present an approach of forecasting energy demand that can be part of a smart cities cloud solution. Since our cities face non-stop growth in population and infrastructures, handling its resources in an intelligent way may result in multiple cost savings. One of the resources which is very important for smart cities is electricity, it is of crucial importance that it is handled in an efficient and intelligent way. The basic role of the smart energy concept is to optimize its consumption and demand resulting in decreased energy costs and increased efficiency. Among the variety of benefits, the smart energy concept mainly enhances the quality of life of the inhabitants of the cities as well as making the environment cleaner. One of the approaches for the smart energy concept is to develop prediction models using ML algorithms in order to forecast energy demand, especially for daily and weekly periods.

The upcoming chapter describes thoroughly what is behind the deep learning concept as a subset of ML and how neural networks can be applied for developing energy prediction models. A specialized version of the RNN, e.g., LSTM, and unsupervised neural network type called autoencoders are described in detail. With an autoencoder unsupervised neural network, features are transformed so that important information is not lost due to high inter-correlation between them. With LSTM the historical influence of the data has been captured. The LSTM can capture the long-time dependency with constant error propagation while using backpropagation through time BPTT, which outperforms the standard implementation of the RNN. To build a deep learning model, the computer program ANNdotNET is introduced. The ANNdotNET is an open source project hosted at https://github.com/bhrnjica/anndotnet on GitHub, the largest open source repository platform. The ANNdotNET provides a user-friendly ML framework with the capability of importing data from the smart grids of a smart city. By design, the ANNdotNET is a cloud solution program that can be connected with other IoT devices for data collecting, feeding, and providing efficient models to energy managers for a bigger smart city cloud solution. As an example, the chapter provides the evolution of daily and weekly energy demand models for Nicosia, the capital of Northern Cyprus. Currently, energy demand predictions for the city are not as efficient as expected. Therefore, the results of this chapter can be used as efficient alternatives for IoT-based energy prediction models for the city.

Using prediction models based on deep neural network, one might not be able to answer questions about behavior, seasonality, and trends of a given time series. To cope with this problem, two state-of-the-art time series decomposition algorithms are used here in order to analyze and determine the trend and seasonality of energy demand. Moreover, the prediction model based on time series decomposition called TSD has been developed and compared with the deep learning model.

4.2 Artificial Neural Network

A number of ML algorithms have been developed over the past few decades that try to discover knowledge from the data. Which ML algorithm is the best, for a given problem? Is the algorithm satisfactory? These are the two main questions that were discussed in the thousands of ML studies. It seems that the former attracted less attention than the latter. Artificial neural network, ANN, undoubtedly is one of the most popular ML algorithms that every data scientist has heard about it. It is a part of supervised ML algorithms that is based on the concept of the biological neural network. Similarly, as a genetic algorithm GA tries to mimic biological evolution [15], ANN attempts to simulate the decision process as human neurons do. The concept of ANN is based on the neuron that can be described as the basic cell of the human brain. Each neuron consists of a cell, a tubular axon, and dendrites. The cell processes signals coming from dendrites and sends it to the axon. The axon forms synaptic connections with other neighboring neurons. The axon consists of branched ends which are used as the input for the next neuron cell. Neurons are linked via a synapse where signals are exchanged from one neuron to another.

Akin to the biological neuron, the artificial neuron is defined as a set of input parameters x_i $(i = 1, .. n)$, which represents the input signals, set of weight factors w_i $(i = 1, \ldots, n)$, which represents the synapses, the dot product $\sum \mathbf{w} \cdot \mathbf{x}$ of input and weighted vectors, representing the neuron cell, and activation function a (.), representing the axon of biological neurons [20]. Figure 4.4 shows the similarities between the biological and artificial neuron.

The first concept of the artificial neuron is called perceptron which was introduced by Rosenblatt [26]. Let the x_n represent the input vector with n components, the associated weight w_n, and bias value b_0 and activation function $sign$. The output y of the perceptron can be expressed as:

$$y = f\,(net) = f\,(\mathbf{w} \cdot \mathbf{x}) = sign\,\left(\sum_{i=1}^{n} x_i w_i + b_0\right), \tag{4.1}$$

where sign represents the activation function defined as:

$$sign\;(net) = \begin{cases} +1, if\ net \geq 0, & \mathbf{w} \cdot \mathbf{x} \geq 0 \\ -1, if\ net < 0, & \mathbf{w} \cdot \mathbf{x} < 0 \end{cases} \tag{4.2}$$

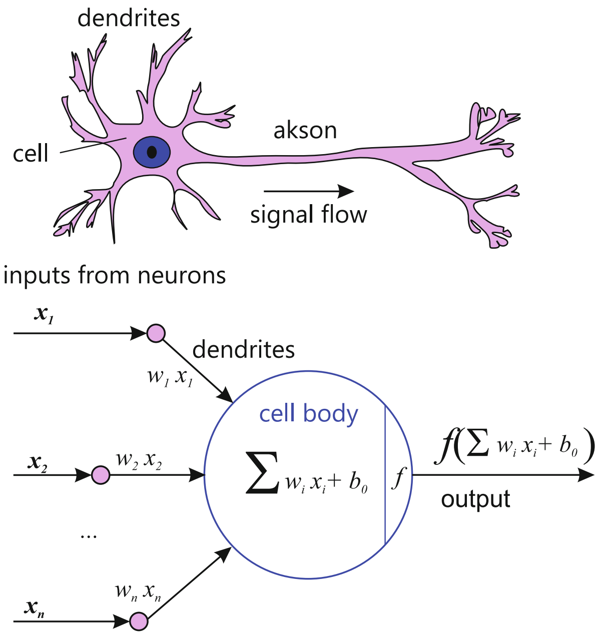


Fig. 4.4 Graphic interpretations of biological and artificial neurons and their similarities

As can be seen the activation function is the last operation in the expression (4.2), which obviously shows the perceptron produces an output as binary value of 1 and -1.

Besides $sign$ there are many other activation functions which can produce different kind of outputs, e.g., $Tanh$, $ReLU$, $Sofmax$, etc. The expression (4.1) and (4.2) with any combination of the activation functions represents the forward pass of one neuron. The forward pass calculates the output for a given input and weights values. In context of a whole ANN, forward pass calculates the output of each neuron in the network, where the last neuron's output represents the output of the network.

Just as millions of biological neurons can be connected, the artificial neurons can form an ANN which can solve very complex problems. Neurons in ANN are grouped in layers. Each layer can consist of one or more neurons. Usually, ANN layers are classified as input, output, and hidden layers. The input layer represents the layer constructed from the input variables (called features). The number of neurons in the input layer is always related by the number of features. Similarly, the output layer is based on the output variable. The number of output variables (called labels) must be the same as the number of neurons in output layer.

The simplest ANN can be formed from at least one input, one hidden, and one output layer. This simple network configuration is called the feed forward network, FFN. The number of neurons of each layer may vary depending on the complexity of the problem. In the input layer, each neuron corresponds to the input parameters, while the output layer is related to the output result. In the middle of the input and the output layer there can be one or more hidden layers with arbitrary numbers of neurons. Figure 4.5 shows the FFN with four layers, one input, one output, and two hidden layers.

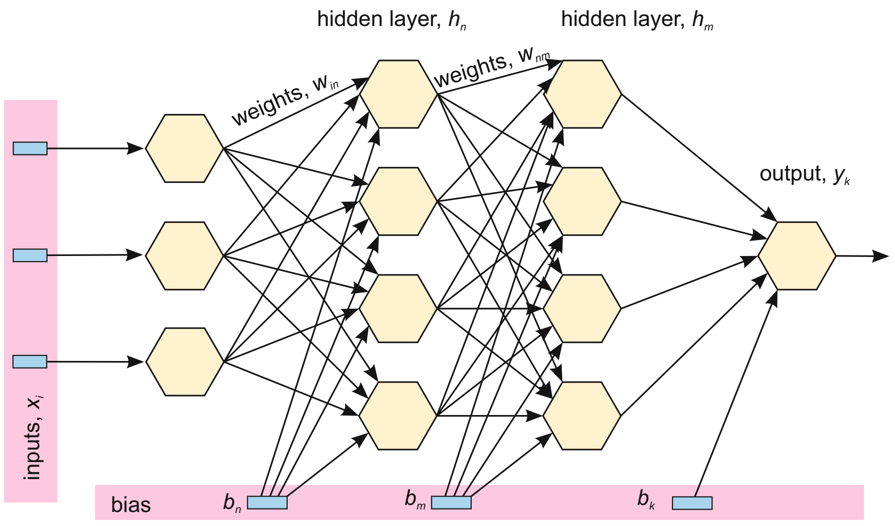


Fig. 4.5 FNN with four interconnected layers

Since each neuron makes some kind of decisions, one can conclude that one perceptron cannot do much. In case of the ANN previously described, the ANN can produce valuable decisions, which can lead to a solution to complex problems. In all cases the process of producing a better solution depends on the weighted factor values of each neuron.

4.2.1 Learning Process in ANN

The process of finding suitable weight values is called learning of ANN. Finding the weight values starts with output calculations of each neuron in the network. The process begins with output calculations of neurons in the input layer, then the output of each input neuron becomes the input for the neurons in the first hidden layer, and so forth. Once the result of the last neuron is calculated in the output layer, the result becomes the output of the network. The result is calculated for each sample (row) in the training dataset.

Assume the training dataset is defined with two features and one label. Let Table 4.1 represent the training dataset with 3 samples (rows). As illustrated in Fig. 4.6, FFN with a 2-4-1 structure was used and indicates that the input layer

Table 4.1 Sample training dataset consists of two features (X_1 and X_2) and one label (Y), with three data samples (rows)

X_1	X_2	Y
1	1	2
1	2	3
3	1	4

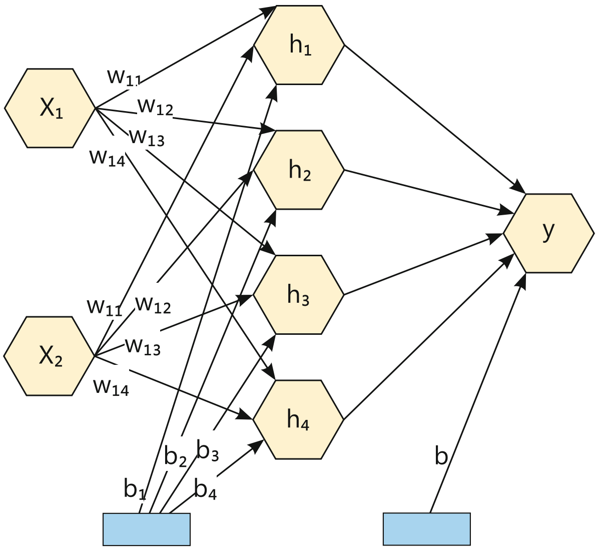


Fig. 4.6 FNN with a 2-4-1 structure, FNN(2, 3, 1)

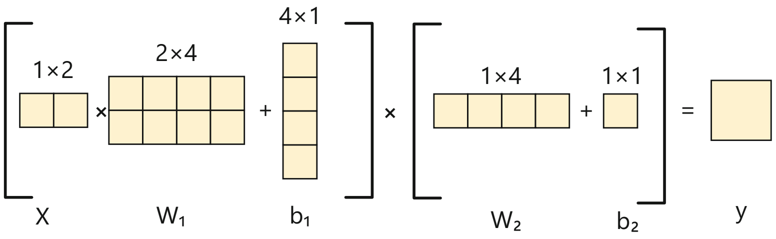


Fig. 4.7 Matrix multiplication generated from the previously defined FNN (2, 3, 1)

has two neurons, the hidden layer has four neurons, and the output layer has one neuron. Identity activation function ($x = f(x)$) at both hidden and output layers was applied for the sake of simplicity.

Once the training dataset and network configuration are defined, the network output can be calculated. The ANN output calculation is based on the matrix calculation. Figure 4.7 shows a matrix representation of the network given in Fig. 4.6.

Based on Fig. 4.7, and the training dataset given in Table 4.1, the output is calculated for each row of the datasets, but first, the initial values of the weights and biases must be defined. This is usually a random process. Assume that the following values are assigned to the matrices W_1 and W_2, and biases b_1 and b_2.

$$W_1 = \begin{bmatrix} 0.5\ 0.5\ 0.5\ 0.5 \\ 0.6\ 0.6\ 0.6\ 0.6 \end{bmatrix}; \quad W_2 = \begin{bmatrix} 1.8\ 0.4\ 0.4\ 0.4 \end{bmatrix} \tag{4.3}$$

$$b_1 = \begin{bmatrix} 0.1 \\ 0.12 \\ 0.10 \\ 0.11 \end{bmatrix}; \quad b_2 = \begin{bmatrix} 0.1 \end{bmatrix} \tag{4.4}$$

Now, the network output can be calculated as:

$$[X \cdot W_1 + b_1] \cdot [W_2 + b_2] = \begin{bmatrix} \hat{y} \end{bmatrix}, \tag{4.5}$$

Using the data from Table 4.1, the outputs are calculated for each row:

- Row 1:

$$\hat{y}_1 = \left[[1\ 1] \begin{bmatrix} 0.5\ 0.5\ 0.5\ 0.5 \\ 0.6\ 0.6\ 0.6\ 0.6 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \right] \cdot \left[[1.8\ 0.4\ 0.4\ 0.4] + [0.4] \right] = 2.26. \tag{4.6}$$

- Row 2:

$$\hat{y}_2 = \left[[1\ 2] \begin{bmatrix} 0.5\ 0.5\ 0.5\ 0.5 \\ 0.6\ 0.6\ 0.6\ 0.6 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \right] \cdot \left[[1.8\ 0.4\ 0.4\ 0.4] + [0.4] \right] = 3.34. \tag{4.7}$$

- Row 3:

$$\hat{y}_3 = \left[[3\ 1] \begin{bmatrix} 0.5\ 0.5\ 0.5\ 0.5 \\ 0.6\ 0.6\ 0.6\ 0.6 \end{bmatrix} + \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \right] \cdot \left[[1.8\ 0.4\ 0.4\ 0.4] + [0.4] \right] = 4.06. \tag{4.8}$$

Based on the expressions (4.6), (4.7), and (4.8) the predicted value represents the column vector of three elements:

$$\hat{y} = \begin{bmatrix} 2.26 \\ 3.34 \\ 4.06 \end{bmatrix}. \tag{4.9}$$

On the other hand, the actual outputs from Table 4.1 can be represented by the column vector as:

$$y = \begin{bmatrix} 2 \\ 3 \\ 4 \end{bmatrix}. \tag{4.10}$$

The residual vector e, which is the deference between the actual and predicted values, is given as:

$$e = \begin{bmatrix} 2 - 2.26 \\ 3 - 3.34 \\ 4 - 4.06 \end{bmatrix} = \begin{bmatrix} -0.26 \\ -0.34 \\ -0.06 \end{bmatrix}. \tag{4.11}$$

The learning process is based on defining the cost function C which can be of various types depending on the problem at hand. In most cases, it is the squared error between the actual and predicted results [27]:

$$C = \frac{1}{2} \sum_{i=1}^{n} e_i^2, \tag{4.12}$$

where:

- e is the residual values,
- n is the number of data samples for the training,
- y is the actual values,
- $\hat{y}$ - is the calculated values.

In case of the previous example, the cost function produces the following error value:

$$C = \frac{1}{2} \sum_{i=1}^{3} (y_i - \widehat{y_i})^2 = \frac{1}{2} \left((-0.26)^2 + (-0.34)^2 + (-0.06)^2\right) = 0.0934. \tag{4.13}$$

Minimizing the error value is the central part of training process with an ANN and might result in satisfactory predictions. One common way is to apply the backpropagation algorithm that is the iterative process of correcting weights and biases based on the partial derivative of the cost function.

For instance, at each neuron k, the weight value ${w_k}^i$ at the iteration i will have the new value ${w_k}^{i+1}$ in the next iteration $i + 1$, for the gradient of the cost function with respect to the w_k multiplied with the learning rate factor η:

$$\Delta w_k = \eta \frac{\partial C}{\partial w_k}. \tag{4.14}$$

The new value of the weight, w_k, in the $i+1$ iteration is expressed as:

$${w_k}^{i+1} = {w_k}^i + \Delta w_k. \tag{4.15}$$

For the calculation of the cost function, the gradient starts from the last (output) layer and is propagated backwards to the input layer using the chain derivative rule.

The entire learning process can be described in two stages for each iteration. Once the iteration starts, the output is calculated by starting from the input layer, and for each weight and each input variable, the output is calculated for each neuron. Once the output of the network is calculated, the second process is started by calculating the gradient from the output layer to the input layer in the backwards order. For each weight value, the gradient is calculated and added to the previous values as shown in Eq. (4.15). Training of the FFN can be a complex task, since not all ANN models can solve the problem accurately. Since an ANN model can be built with one or more hidden layers, and each hidden layer can contain an arbitrary number of neurons, the learning process may provide unexpected results. In the case of small number of neurons in the hidden layer, the model may be too rigid, and the learning process very slow, which leads to the fact that the number of neurons in the ANN is not sufficient to adapt to the data. On the other hand, a large number of neurons in the hidden layer may lead the ANN model to fit the data perfectly, but due to the complex nature of the problem the model is trying to predict, the prediction for the unseen data may give unsatisfactory results [3].

4.2.2 Deep ANN

Previous studies indicate that FFNs are not powerful enough for most of today's problems. For instance, FFNs were not found suitable for the natural language processing, social network filtering, speech and audio recognition, machine translation, medical image analysis, bioinformatics, drug design, stochastic time series forecasting, etc. In order to solve such problems, the network configuration must be extended and made more robust. To achieve a more robust network, one may increase the number of hidden layers. In this situation, the multiple hidden layers with a nonlinear activation function can produce nonlinear processing which is more efficient for solving complex problems. Simply by increasing the number of hidden layers produces a very complex system of network configurations that need to be learned. In most cases more than one hidden layer in the network cannot be learned in the same way described previously, due to the vanishing and exploding gradients phenomenon [2]. This changes the approach of looking at ANNs and revolutionized the learning process for multiple layer networks.

As stated previously the keyword "deep" in "deep ANN" refers to the number of hidden layers. One can define the credit assignment path (CAP) depth as the transformation chain from the input to output of the neural network [30]. In the case of FNN, the depth of the CAP indicates the number of hidden layers plus one for the output layer, since it is parametrized in the same way as the hidden layer. For RNN where the input features can be propagated through a layer more than once per iteration, the CAP depth is potentially undetermined. So, to make a measurable difference between deep learning and learning requires a CAP depth to be greater than 2. The process of learning complex networks configuration, such as deep neural networks, deep belief networks, and RNN, is called deep learning, DL, a subset of the wider ML field. How DL is specific to the ML field can be depicted graphically as in Fig. 4.8. The learning process of a DL specific networks is a very complex task, it is still based on the backpropagation error concept with a specific way of error propagation and optimization techniques.

4.2.3 Recurrent Neural Network

FFN models are usually built on the fact that data do not have any order when entering into the network. So, the output of ANN depends only on the input features. In case of specific data when the order is important, usually when data is recorded in time or when dealing with sequences of data, simple FFN cannot manage it as one can expect because the previous state cannot be incorporated [24]. When the output

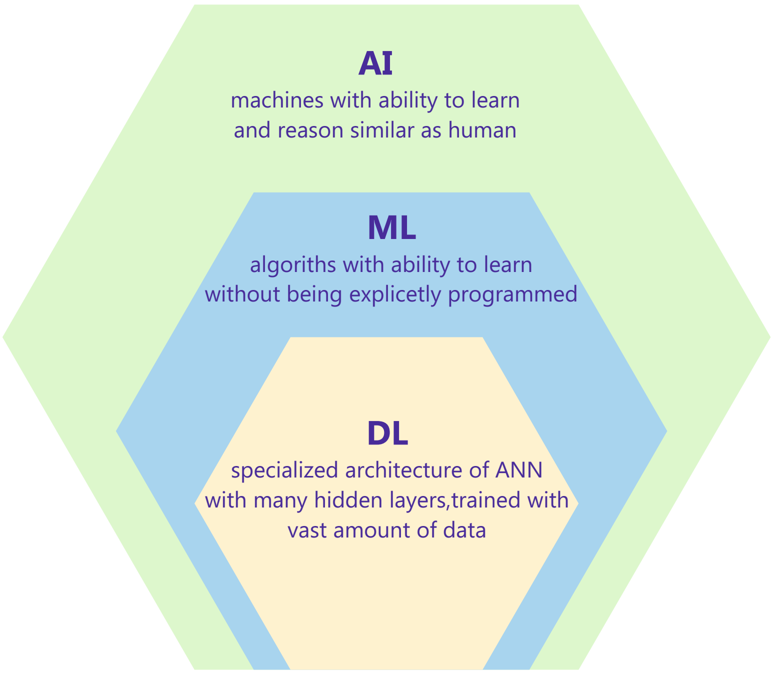

AI
machines with ability to learn
and reason similar as human
ML
algoriths with ability to learn
without being explicetly programmed
DL
specialized architecture of ANN
with many hidden layers,trained with
vast amount of data

Fig. 4.8 DL as a subset of ML and AI

is determined by both the inputs and the previous states, the FNN must be extended to support the previous states. A well-known solution for this kind of problem is to develop the RNN, which was first introduced by Hopfield [12], and later popularized when the backpropagation algorithm was improved [24]. The concept of the RNN is depicted in Fig. 4.9. As seen, the RNN contains cycles showing that the current state of the network relies on current data, but also on the data produced by the previous outputs of the network. So, in the case of the RNN, two kinds of inputs are provided: the output of the previous time, $\mathbf{h_{i-1}}$, and the current input $\mathbf{x_i}$. Due to its nature, the RNN has a special kind of internal memory which can hold long-term information history [29].

Figure 4.9 shows two kinds of representations of the RNN. On the left side, the RNN is presented in classic feed forward like mode, where the three layers are presented: input, hidden, and output layer. Around the hidden layer we can see cycling which indicates the recursion. The RNN can be shown in an unrolled state in time t. The RNN is presented with t interconnected FFN, where t indicates the past steps, thus far. The concept of the RNN is promising and very challenging, but there are problems with applications, mainly when dealing with complex time dependent models [2]. Most of the obstacles of the RNN can be summarized in two categories [2]: the vanishing and exploding gradient. The learning process of the RNN is mostly based on the backpropagation algorithm, the so-called backpropagation

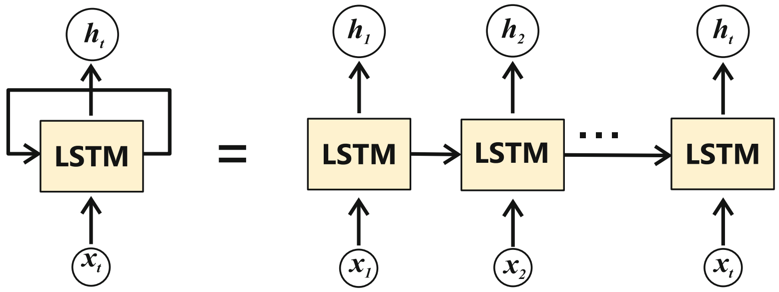


Fig. 4.9 Schematic representation of the RNN

through time or BPTT. The BPTT algorithm stores the activation of the units while going forward in time, while in the backwards phase takes those activations for the gradient calculation [27]. In the vanishing gradient problem of learning RNN, updates of weights are proportional to the gradient of the error calculated in the previously described manner. In most cases, the gradient value is negligibly small, which results in the fact that the corresponding weight is constant and stops the network from further training. The exploding gradient problem refers to the opposite behavior, where the updates of weights (gradient of the cost function) become larger in each backpropagation step. This problem is caused by the explosion of the long-term components in the RNN. In both cases the error propagation through the network is not constant, which causes one of the mentioned problems.

The solution to the abovementioned problems is found in the specific design of the RNN, called long short-term memory, LSTM [11]. LSTM is a special RNN which can provide a constant error flow. The constant error propagation through the network involves a special network design. LSTM consists of memory blocks with self-connection defined in the hidden layer, which has the ability to store the temporal state of the network. Besides memorization, an LSTM cell has special multiplicative units called gates, which control the information flow. Each memory block consists of the input gate that controls the flow of the input activations into the memory cell, and the output gate controls the output flow of the cell activation. In addition, an LSTM cell also contains the forget gate, which filters the information from the input and previous output and decides which one should be remembered or forgotten and dropped. With such selective information filtering, the forget gate scales an LSTM cell's internal state, which is self-recurrently connected by previous cell states [8]. Besides gating units, the LSTM cell consists of a self-connected linear unit called constant error carousel, CEC, whose activation is called the cell state. The cell state allows for constant error flow, previously mentioned as the problem of the vanishing or exploding gradient, of the backpropagation error in time. The gates of the LSTM are adaptive, since each time the content of the cell is out of date, the forget gate learns to reset the cell state, so the input and the output gates control the input and the output, respectively. Figure 4.10 shows an LSTM cell with activation layers: input, output, forget gates, and the cell. Each layer contains the activation function before passing through.

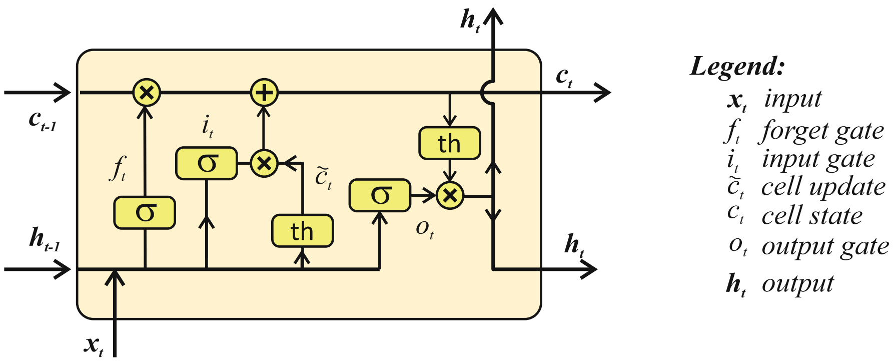


Fig. 4.10 LSTM cell with its internal structure

As can be seen, an LSTM network can be expressed as an ANN where the input vector $x = (x_1, x_2, x_3, \ldots x_t)$ in time t maps to the output vector $y = (y_1,\ y_2,\ \ldots,\ y_m)$, through the calculation of the following layers:

- the forget gate sigmoid layer for the time t, f_t is calculated by the previous output h_{t-1}, the input vector x_t, and the matrix of weights from the forget layer W_f with an addition of bias b_f:

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] + b_f\right); \tag{4.16}$$

- the input gate sigmoid layer for the time t, i_t is calculated by the previous output h_{t-1}, the input vector x_t, and the matrix of weights from the input layer W_i with an addition of bias b_i:

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right); \tag{4.17}$$

- the cell state in time t, C_t, is calculated from the forget gate f_t and the previous cell state C_{t-1} by multiplicative operation $\otimes$. The result is applied as the first argument of the additive operation $\oplus$ and the input gate i_t, which is then applied as the first argument of the multiplicative operation of the cell update state $\tilde{c}_t$ which is a $tanh$ layer calculated by the previous output h_{t-1}, input vector x_t, and the weight matrix for the cell with an addition of bias b_C:

$$C_t = f_t\ \otimes C_{t-1} \oplus\ i_t \otimes \tanh\left(\ W_C \cdot [h_{t-1}, x_t] + b_C\right); \tag{4.18}$$

- the output gate sigmoid layer for the time t, o_t is calculated by the previous output h_{t-1}, the input vector x_t, and the matrix of weights from the output layer W_f with an addition of bias b_o:

$$o_t = \sigma\left(\ W_0 \cdot [h_{t-1}, x_t] + b_0\right). \tag{4.19}$$

The final stage of the LSTM cell is the output calculation of the current time h_t. The current output h_t is calculated with the multiplicative operation $\otimes$ between output gate layer and $tanh$ layer of the current cell state C_t.

$$h_t = o_t \otimes \tanh(C_t). \tag{4.20}$$

The current output, h_t, has passed through the network as the previous state for the next LSTM cell, or as the input for an ANN output layer. The operation connections $\otimes$ and $\oplus$, which correspond to multiplication and addition connections, allow the gates to process the information based on the previous cell output, as well as the previous cell state. The previous LSTM cell description represents one of the several variants which can be found in the literature [19].

4.2.4 Deep LSTM RNN

Deep ANN has proved to be very effective in solving complex problems. Similarly, deep LSTM RNN can be defined as more than one LSTM layer in the ANN. The fact is that an unrolled LSTM cell in time represents a deep FFN, which indicates the complex network architecture. Deep LSTM RNN can be defined with more than one LSTM layer. The LSTM layers are stacked vertically, with the output sequence of one layer forming the input sequence of the next one (Fig. 4.11).

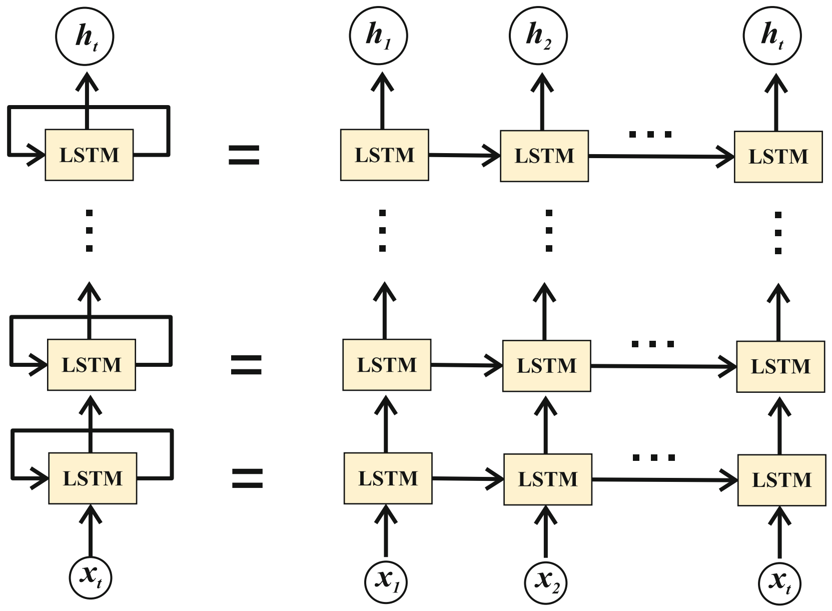


Fig. 4.11 Schematic representation of the deep LSTM RNN

The deep LSTM RNN has proved to be very effective and outperforms the standard implementation of the LSTM [9]. Deep LSTM RNN has the ability to learn at different time scales over the input [10]. In addition, deep LSTM is more efficient than standard LSTM, since parameter distribution over the space through multiple layers is handled with less memory [29].

4.3 Modeling Time Series Events

As previously mentioned, ML methods generally extract the knowledge from the data in two phases. The first phase is training the network configuration in order to get the best possible weights so that the model can predict the future values with minimum error. The second phase is model validation, where the trained model is validated against the validation dataset. Such dataset contains the data which are not used during the training phase. In both of the phases, the data is a crucial part of the ML solution. With noisy or inappropriate datasets, regardless of the implemented ML algorithm, a reliable prediction model would not be created. Therefore, the most important component of the ML solution is to prepare high quality datasets prior to the training phase.

Undoubtedly, modeling time series data is one of the most challenging tasks of ML. A time series represents a sequential set of data samples recorded over successive times. It can also be defined as a set of vectors $x\,(t)$, $t = 1, 2, 3, \ldots$, where t is the time. The time series data is always arranged in chronological order. Data usually contains a single variable, which represents the univariate time series. In case more than one variable is used, it is termed as a multivariate time series.

For better understanding time series events, the associated data can be decomposed into several components so that each component represents an important property of the event. Decomposition of the time series is usually based on rates of change. With this in mind, time series can be decomposed into three components: trend, seasonal, and random components [16]. As an underlying component, trend represents tendency of time series data to increase, decrease, or stagnate over a long period of time. It can be also described as long-term movement. The seasonal component represents the fluctuation within a year. The seasonal variation of time series is an important component specially in business related time series data, where the detection of a seasonal time interval can increase business values. The third component, the random component, represents everything else. It usually represents the randomness of the time series.

The time series components can be determined in two ways: as additive or as multiplicative models. In case of the additive model, the time series data can be expressed as:

$$y\,(t) = S\,(t) + T\,(t) + R\,(t) \tag{4.21}$$

where $y(t)$ is the data, $T(t)$ is the trend component, $S(t)$ is the seasonal component, and the $R(t)$ is the random component at time t.

The multiplicative model of time series can be expressed as:

$$y(t) = S(t) \times T(t) \times R(t) \tag{4.22}$$

In case when the seasonal variation is relatively constant over time, additive decomposition is recommended. On the other hand, the multiplicative decomposition is recommended when the seasonal component is proportional to the level of the time series [16], or when the seasonal variation increases over time. However, the multiplicative model can be expressed as additive, if the time series is transformed by a log transformation. In this case, using a log transformation any multiplicative decomposition can be expressed as additive:

$$\begin{aligned} y(t) = S(t) \times T(t) \times R(t) \Leftrightarrow Ln(y(t)) \\ = Ln(S(t)) + Ln(T(t)) + Ln(R(t)). \end{aligned} \tag{4.23}$$

In order to decompose time series data into its components several methods have been developed. One of the most popular methods is seasonal-trend decomposition based on Loess [5], which has been proven to be very effective for longer time series. There is a more recent seasonal-trend decomposition based on regression decomposition [7], which is more generic, and it allows for multiple seasonal and cyclic components, as well as multiple linear regressors with a constant. It also provides flexible, seasonal, and cyclic influences. Recently, Facebook has released a times series decomposition and forecasting tool called Prophet. The company used this tool for times series data analytics and forecasting [31]. All the mentioned decomposition methods are implemented in well-known statistical packages, e.g., *R* or Python languages.

4.3.1 Time Series Decomposition Procedure

In order to decompose the time series, the first step is the selection of a decomposition method. Once the decomposition method has been selected, the trend is the first component which should be estimated. De-trending the time series is the second step. In case of an additive model, the trend component is subtracted from the time series data. For the multiplicative model, time series data is divided by the estimated trend component.

The seasonal component is estimated from the de-trended part of the time series. Since the seasonal component is based on the underlying periodicity of the events, it can be weekly, monthly, yearly, or any customized seasonal length. One of the simplest methods to estimate, the seasonal component is to average the de-trended values for the specific season. For example, to get a seasonal effect for January, one can average the de-trended values for each January in the series. The seasonal value has to be adjusted depending of the decomposition type, zero for additive, and one for a multiplicative model.

The final step of the decomposition is to estimate the random component. The random component is simply estimated when the trend and season are subtracted from the data series. In case of the multiplicative decomposition, the random component is estimated by dividing the multiplication of trend and seasonal components by the time series data.

The following expressions for the random component can be stated as:

$$R(t) = y(t) - [S(t) - T(t)] - \text{for additive decomposition,} \tag{4.24}$$

$$R(t) = \frac{y(t)}{S(t) \times T(t)} - \text{for multiplicative decomposition,} \tag{4.25}$$

Time series decomposition with its components is very often graphed, since it provides a clear picture for the data behavior.

4.3.2 Energy Demand Analysis by Time Series Decomposition

Reliable electricity prediction models lead to a sustainable power supply and provide clear details about the health of a power system in a smart city setup. In this section, we demonstrate how a decomposed time series of energy consumption data can be used to develop accurate electricity demand prediction models. As an example, the daily series of electricity consumption in the northern part of Nicosia during the 2011–2016 period has been considered. Nicosia, the capital city of Cyprus, has a typical Mediterranean climate with an annual average electricity consumption about 4000 MWh in its northern part and about 6000 MWh in the southern part. The overall image of the electricity consumption time series is shown in Fig. 4.12. Based upon observations, the rapidly increasing trend has been observed since 2013. The figure also represents seasonal behavior, since the sinusoidal shape of the data can be visually recognized. The statistical properties of the electricity consumption series are shown in Table 4.2.

Quantile values indicate that the data is skewed to the left since the median has a lower value than the mean.

The decomposition of the dataset was performed by using the STL *R* package [5]. In order to get the best possible decomposition with maximum amplitude values of the seasonal component, the decomposition was performed for wide range of argument values. The best possible decomposition was estimated for the argument value of $f = 365$, which indicates yearly periodic behaviors of the dataset. Figure 4.13 shows the decomposition of the energy demand time series.

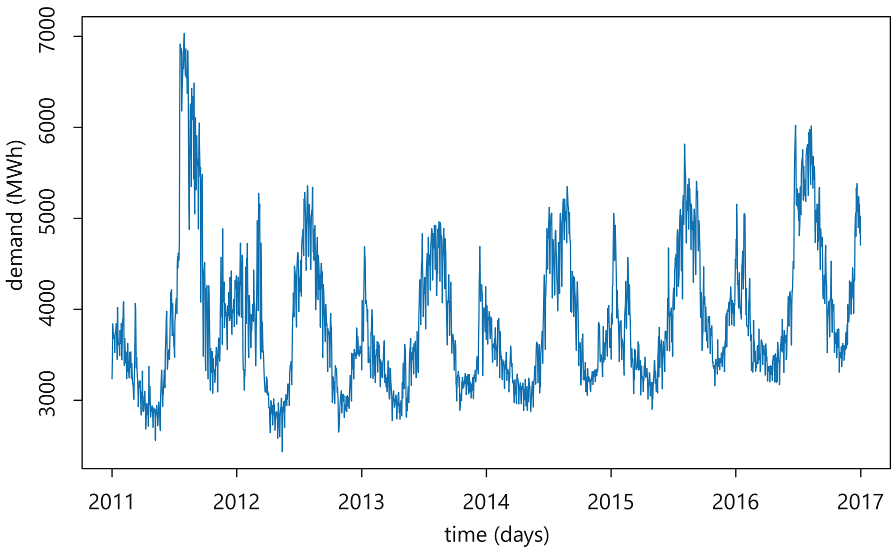


Fig. 4.12 Energy demand time series at northern part of Nicosia

Table 4.2 Summary of electricity consumption 2011–2016

Min	1st quarter	Median	Mean	3rd quarter	Max
2433	3311	3664	3879	4284	7032

Figure 4.13 also indicates that the random component has significant influence on the energy demand dataset. From Table 4.3, one can see that the basic statistical indicators show that the minimum value is less than −900 MWh, while the maximum value is greater than 1700 MWh. This implies that the forecasting model may produce 50% of the relative error.

The seasonal component can be described as a periodic function with two minimum and one maximum points. The first minimum point is reached in 122 days, the second week in May, while the second minimum point is reached in the first week of November. The maximum energy demand is reached in the first week of August. The maximum point of seasonal change is expected in the beginning of August because of high electricity demand due to high temperatures and increased tourisms. The trend component of the energy demand dataset shows growth in the last 4 years. This indicates that energy demand increases every year. The reason for the increasing trend can be found in the constant growth of infrastructure and population on the island.

From the previous decomposition time series, over the last years, one can conclude that energy demand has constant growth. Over a period of a year, the

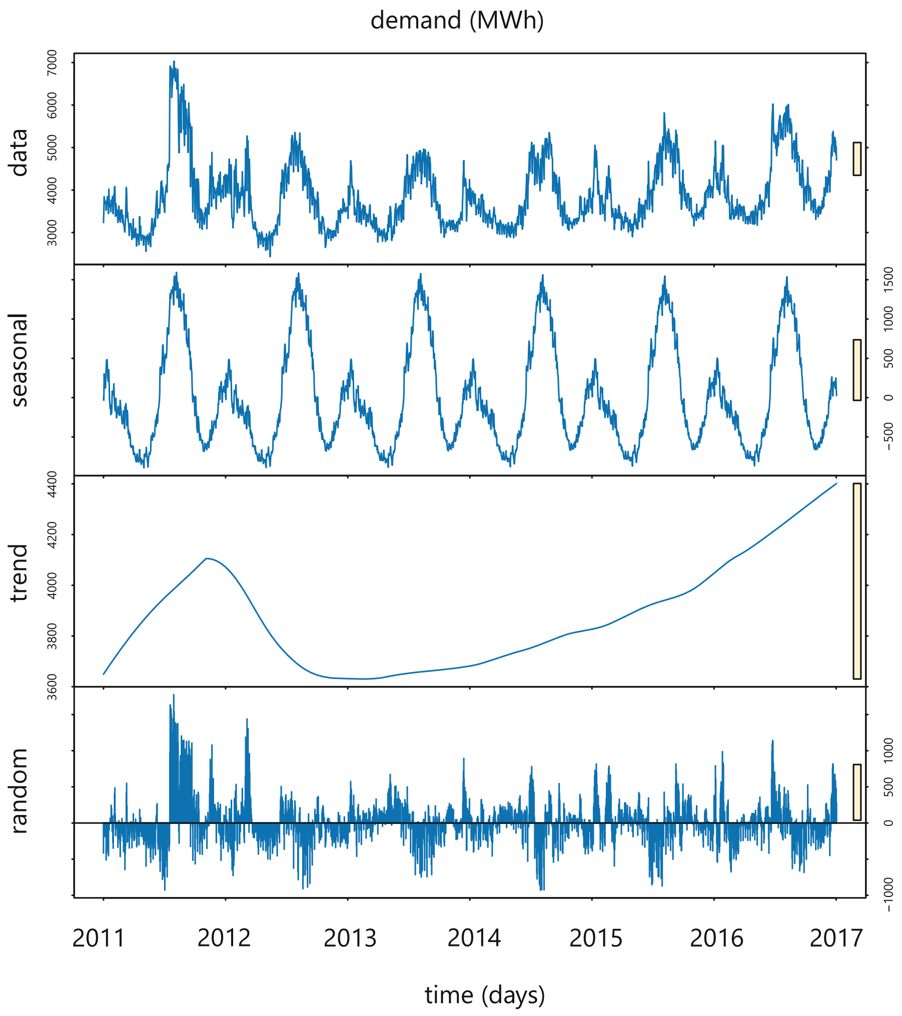


Fig. 4.13 Energy demand time series components

Table 4.3 Summary of random component of energy demand time series in MWh

Min	1st quarter	Median	Mean	3rd quarter	Max
−928.684	−201.329	−21.750	2.466	152.537	1776.062

energy demand reaches two minimum levels: one in the spring and one in the fall. The maximum energy demand is reached during the first days of August, due to maximum temperatures. Seasonal and random components have similar ranges that indicate the energy demand has strong stochastic behavior.

In order to get deeper into seasonal changes, one should see how demand changes weekly, monthly, and quarterly. In order to display different seasons, the Prophet *R* package is used [31]. Figure 4.14 shows the trend and four different seasonal components: weekly, monthly, quarterly, and yearly. The yearly component in Fig. 4.14 clearly shows two minimum and one maximum point, which were previously described. Weekly seasonal changes indicate that the energy demand is higher on work days rather than on the weekend. The monthly seasonal changes do not clearly show increasing or decreasing demand, but it roughly indicates that energy demand is on the highest level in the middle of the month. Similarly, quarterly seasonal changes are increased at the start and the end of the quarter, while the first month of the quarter shows gradual decreased demand.

4.3.3 Energy Demand Forecasting Using Decomposed Series

In the previous section, the energy demand time series was analyzed by decomposing it into three main components: trend, seasonal, and random. Moreover, different types of seasonality were considered in order to get a deeper knowledge of the energy demand data. Based on the previous analysis in this section, forecasting will be performed in order to see how energy demand is propagated in the future. The forecasting procedure was performed using the additive model where nonlinear trends are fit with yearly, weekly, and monthly seasonality. Since the seasonal components have significant impact on the time series data, the prophet forecasting package is used. The package combines many different forecasting methods (e.g., ARIMA, exponential smoothing, etc.) in order to get the best possible model. The forecasting is based on using a flexible regression model or curve-fitting model instead of a traditional time series model that leads to better and more accurate forecasting. The time series decomposition, TSD, model is built on the energy demand dataset from January 2011 to September 2015. From October 2015 to December 2016 the dataset is defined as the validation and testing dataset.

Figure 4.15 shows the TSD model of the energy demand time series data. The black dots are actual values of the daily energy demand, whereas the blue (dark gray) line shows predicted values. Besides prediction values, the image also shows the confidence interval of the prediction.

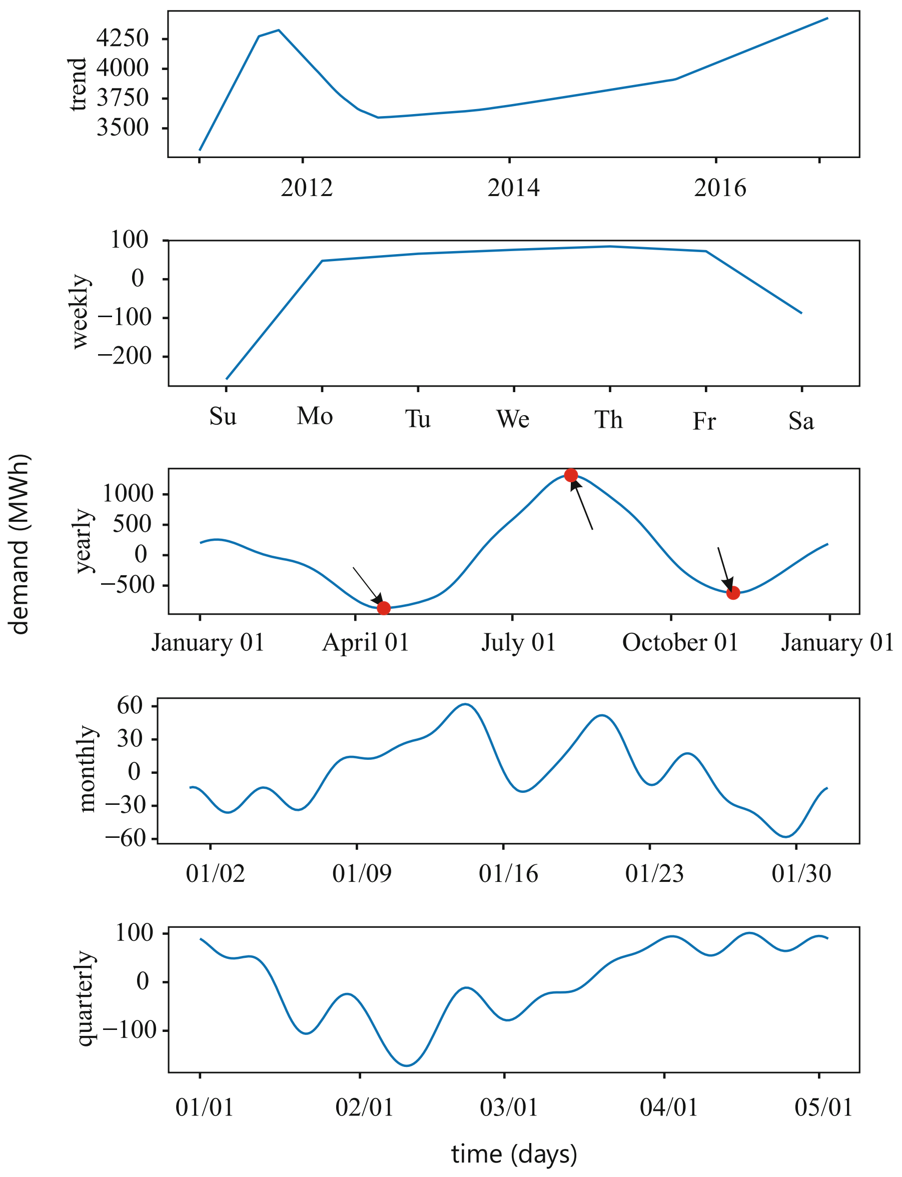


Fig. 4.14 Different seasonal component types of energy demand time series data

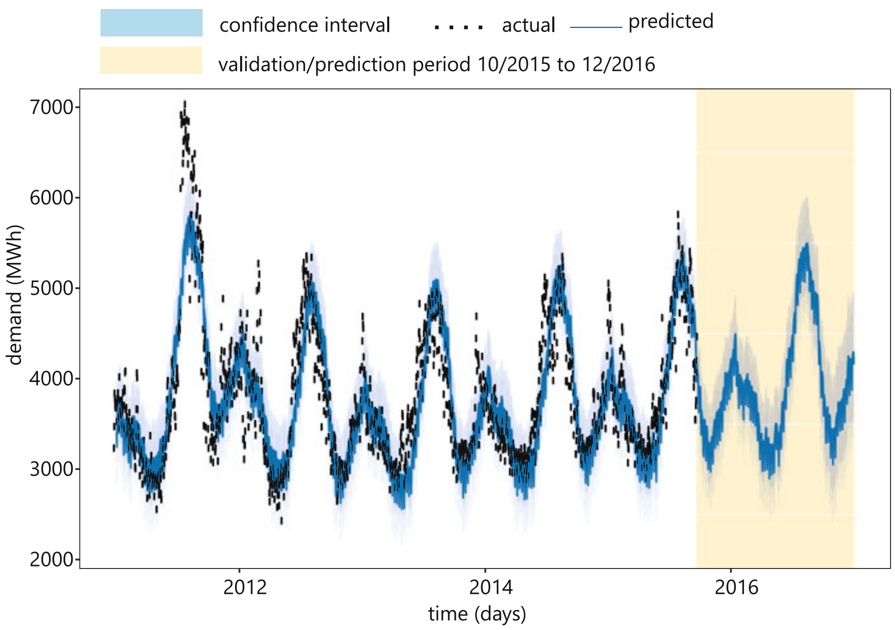
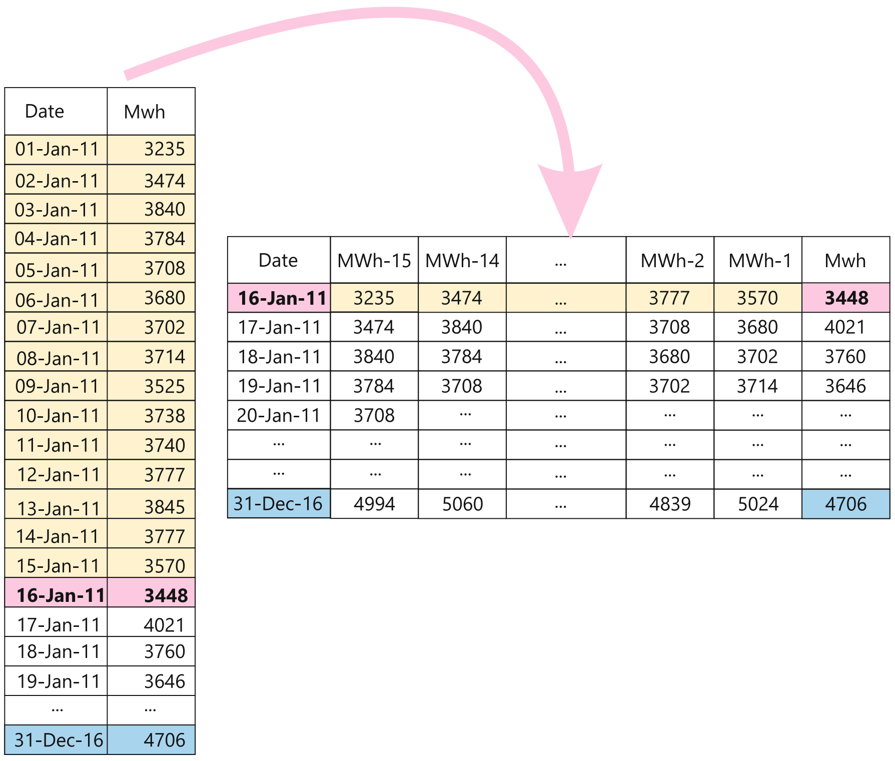


Fig. 4.15 TDS model of energy demand for period 2011–2016

4.4 LSTM Deep Learning Model for Energy Demand Prediction

Using a deep learning technique for modeling time series events seems natural due to the complex nature of such events. In this section, a deep learning model has been developed in order to predict energy demand. Since the energy demand represents a typical time series non-stationary dataset, the LSTM RNN was used. In order to transform the time series into a data frame, 15 days of time lags were used. Once the data frame was created, the three datasets were created to configure the ML workflow. In order to transform the data, configure the neural network, train and evaluate the model, the ANNdotNET [14]—deep learning tool on .NET platform was used. The ANNdotNET is a deep learning tool that implements the ML Engine that is based on the Microsoft Cognitive Toolkit, CNTK [34]. The ML Engine is responsible for training and evaluating deep learning models. Besides the GUI tool that is used for handling data transformation and model training, the ANNdotNET provides a set of APIs which could be integrated into a bigger smart cities cloud solution, SCCL.

The previous section is correlated for data analysis, where the time series data were decomposed and analyzed. The time series is represented with only one variable (energy demand) which is the example of a univariate time series. In order to prepare it for deep learning, it must be transformed into a data frame-based set,



Date	Mwh
01-Jan-11	3235
02-Jan-11	3474
03-Jan-11	3840
04-Jan-11	3784
05-Jan-11	3708
06-Jan-11	3680
07-Jan-11	3702
08-Jan-11	3714
09-Jan-11	3525
10-Jan-11	3738
11-Jan-11	3740
12-Jan-11	3777
13-Jan-11	3845
14-Jan-11	3777
15-Jan-11	3570
16-Jan-11	**3448**
17-Jan-11	4021
18-Jan-11	3760
19-Jan-11	3646
...	...
31-Dec-16	4706

Date	MWh-15	MWh-14	...	MWh-2	MWh-1	Mwh
16-Jan-11	3235	3474	...	3777	3570	**3448**
17-Jan-11	3474	3840	...	3708	3680	4021
18-Jan-11	3840	3784	...	3680	3702	3760
19-Jan-11	3784	3708	...	3702	3714	3646
20-Jan-11	3708	...	...	...	...	...
...	...	...	...	...	...	...
...	...	...	...	...	...	...
31-Dec-16	4994	5060	...	4839	5024	4706

Fig. 4.16 Time series transformation into a 15 features data frame

with features (input) and label (output). The features are generated by the previous values, so-called time lag values, while the label is the time series value at the current time step. Figure 4.16 shows how a univariate time series can be transformed into a data frame with 15 features and one label. In this way, the time series data is transformed so that historical changes have an influence on the current value. For the deep learning that is studied in this chapter, 15 past values are used for the feature generation.

Once the preparation process has been completed, the next step in a deep learning model development is the configuration of the neural network. In order to create a suitable neural network, several different network configurations are prepared. Recently, a special version of the RNN, the LSTM has been providing great results in many engineering fields. However, modeling complex time series events using LSTM is still a challenging task. In order to provide a more accurate network, the LSTM network is combined with additional neural network types.

The time series dataset used for the training network model contains features generated from the same variable. In such conditions, the features have a very strong inter-correlation, often causing overfitting and less reliable results. To avoid overfitting, the features need to be transformed be less inter-correlated, and more independent from each other. There are several techniques to overcome this phenomenon, but one of the most popular is to use autoencoders [33].

4.4.1 Autoencoder Deep Neural Network

Autoencoders are neural networks that can achieve unsupervised learning. Simply said, it uses backpropagation for learning, by setting the target value the same as the input. In other words, it tries to learn features from features, or approximate an identity function. In the neural network context, autoencoders are a set of fully connected layers, that the input and output dimensions of the autoencoder network are the same. Hidden layers always have less neurons than the input/output layer. Figure 4.17 shows an autoencoder neural network used for the energy demand network configuration. As can be seen, the autoencoder is built with fully connected layers, where the first layer starts with the same number of neurons as the input dimension. Then, the neurons in the second layer are reduced by 50%, and the middle-hidden layer is defined with only four neurons. This is called a bottle neck. After the bottle neck, the dimension of the hidden layers is increased, first to 8, and then to 15.

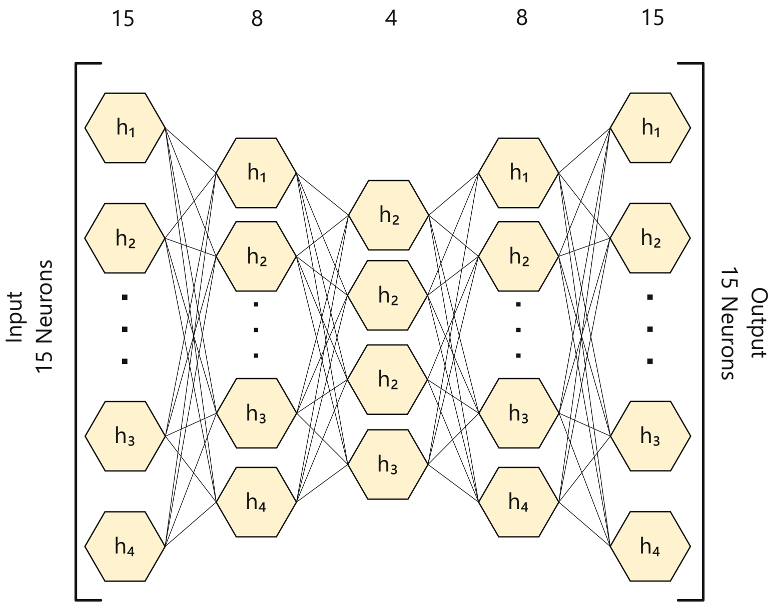


Fig. 4.17 15-8-4-8-15 autoencoder neural network architecture

4.4.2 Training Process of LSTM Deep Learning Model

In order to configure and define the DL model for energy demand, the ANNdotNET tool is used. Once the data preparation has been performed, the network configuration is set up by adding the LSTM layer with a 100 LSTM cell dimension. The LSTM cell is placed after the autoencoder network layer. As a next layer, a DroupOut layer with 30% of dropped values was added. The last layer in the network is the output layer with one neuron. Figure 4.18 shows a schematic deep learning model for energy demand.

The defined model is trained and validated on a 15 features dataset, created from January 2011 to November 2016. The first 80% of the dataset belongs to the training set and the remaining 20% to the validation set. The December 2016 values are used as a test set for comparison analysis between a deep learning model and the TDS model described in the previous section. The training network configuration has been performed using the Adam learner, with the squared error, SE as the loss function, and the root mean square error, RMSE, as the evaluation function. The final model is trained after 5000 epochs.

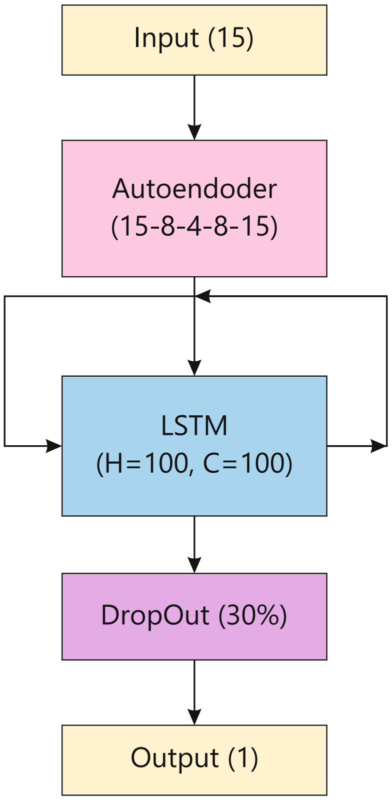


Fig. 4.18 LSTM deep learning model for energy demand

4.4.3 Evaluation Process of the LSTM Deep Learning Model

In order to provide the model evaluation, several performance parameters were calculated against a training and validation set and presented in Table 4.4. A description of each of the performance parameters is given in the literature [15].

From Table 4.4, one can conclude that the model has a high performance for both datasets. The RMSE and R values are roughly equal for both the training and the validation sets. Moreover, the rest of the performance parameters except SE show the same behavior. This is an indication that the model is well trained with a high percentage of accuracy. Figure 4.19 graphically shows the model prediction for the validation set with respect to actual values. The chart series are drawn in different colors (shades) that can be easily seen how prediction values are close to actual values.

Table 4.4 Performance values for the LSTM deep learning model for the training and validation datasets

	SE[a]	RMSE[b]	NSE[c]	PB[d]	R[e]	R^2[f]
Train. set	1.20459	0.02648	0.94370	0.035758	0.97363	0.94795
Valid. set	0.27896	0.02550	0.93856	0.03350	0.97096	0.94277

[a]Squared error
[b]Root mean squared error
[c]Nash–Sutcliffe efficiency
[d]Percentage bias
[e]Pearson's coefficient
[f]Determination coefficient

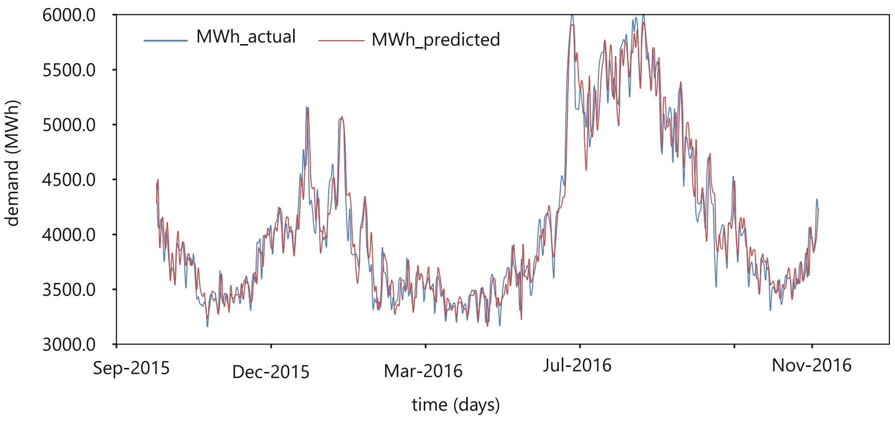


Fig. 4.19 Predicted values calculated by the deep learning model for the validation set

The conclusion that can be established from the model evaluation is that the deep learning model can accurately predict the time period defined by the validation set.

4.4.4 Testing Process of the LSTM Deep Learning Model

Deep learning provides models which can predict future values. Data used for training the deep learning model were not decomposed. In contrast to the TDS model prediction, the data were decomposed, and the trend and random components were included in the modeling. Once the model is calculated, the seasonality component was added, and the prediction (see Fig. 4.15) was calculated. The time series decomposition gives answers for the question as to how data behave in different time periods (seasons), what is the trend of the data in those time periods (seasons)? Deep learning is a black box which trains the model to predict the future values with no additional answers. For this reason, time series decomposition is important in order to prepare and transform the dataset prior to starting deep learning, and also as a comparison analysis between the model predictions.

In order to show how much the deep learning model is accurate, a comparison analysis is performed between the LSTM deep learning and TSD models, using the same dataset. The testing set is represented by the energy demand from December 2016. Table 4.5 summarizes the comparison results.

From Table 4.5 it can be seen that the LSTM deep learning model is significantly better than the TSD model because it is closer to the observed series. However, the comparison chart shown in Fig. 4.20 also shows that the TSD model follows the weekly peaks, but those values are lower than actual values. It can also be noticed that as the prediction period gets longer, the TSD model predicts the values with higher error, while the deep learning model predicts values with a much lower error. The LSTM deep learning model has better RMSE values in all prediction periods: 5 days, 15 days, and monthly. The TSD model has a higher Pearson coefficient, for 5 and 15 days of the prediction, while for the 1 month prediction, the Pearson coefficient is higher for the LSTM deep learning model. The reason why the Pearson coefficient is better for 5 and 15 days for the TDS model may be ignored due to the small dataset, and the RMSE parameter in such a case is more relevant.

Table 4.5 Performance analysis between LSMT and TSD models

	1–5 December 2016		1–15 December 2016		December 2016	
	LSTM	TSD	LSTM	TSD	LSTM	TSD
RMSE	198	353	320	386	281	720
R	0.489	0.923	0.659	0.808	0.829	0.760

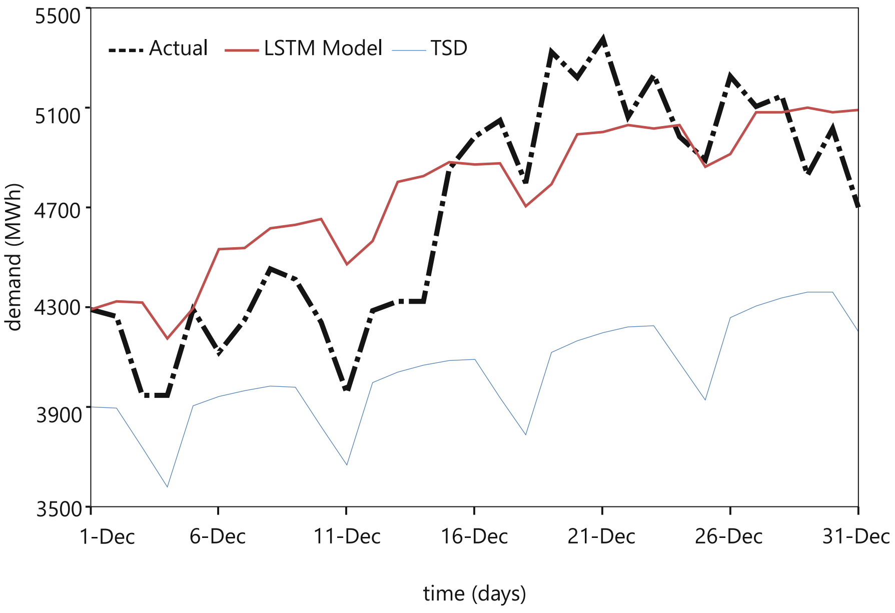


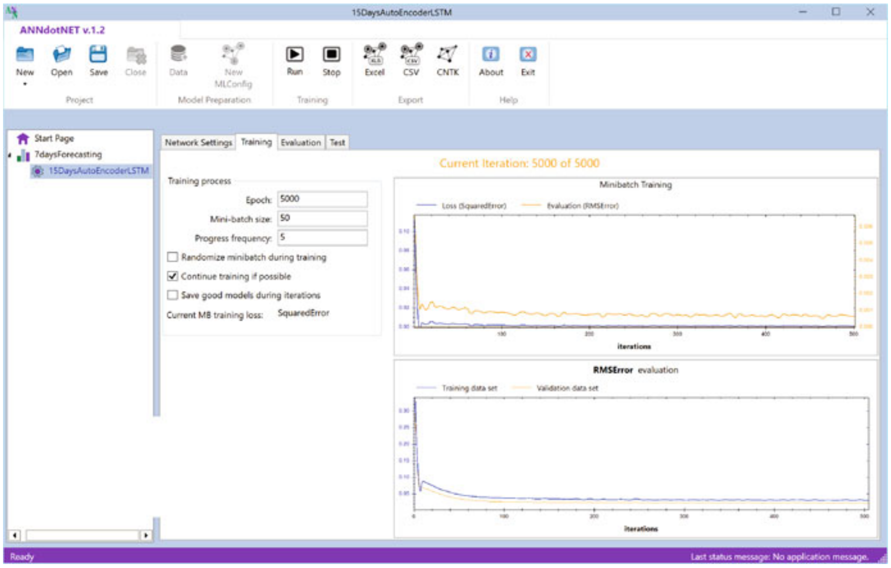
Fig. 4.20 Energy demand prediction for December 2016

Energy demand predictions for December 2016, using the LSTM deep learning model, and the TDS model, with respect to actual values are illustrated in Fig. 4.20. It can be clearly seen that the dashed curve which represents the LSTM deep learning model is much closer to red line, than the TDS model marked with the blue color.

In general, one can say that deep learning model provides a better prediction than the TSD model in all aspects of the analysis.

4.4.5 Deep Learning Model as a Cloud Solution for Smart Cities

In order to prepare, train, and evaluate the deep learning model, the ANNdotNET deep learning tool was developed and used in this study. By using the ANNdotNET, it is possible to incorporate ML tasks into a cloud solution, so that the complete ML process can be automatized and defined into one workflow using cloud services. Once the ML task is incorporated as a cloud solution, it can be part of the bigger smart cities project. In this section details of the possible cloud solution are presented.



Current Iteration: 5000 of 5000

Training process

Epoch: 5000

Mini-batch size: 50

Progress frequency: 5

☐ Randomize minibatch during training

☑ Continue training if possible

☐ Save good models during iterations

Current MB training loss: SquaredError

Fig. 4.21 Training module of ANNdotNET deep learning tool

It is very common that an ML solution is split into three phases:

- Data preparation
- Training ML model
- Model Deployment

The first phase consists of a set of data related tasks responsible for the transformation of raw data into a machine ready (mlready) dataset. This phase may include data transformation, outliers identification, features selection, features engineering, cross validation analysis, etc. Once the data is transformed into an mlready dataset, the next phase starts by defining the input and output layers of the deep neural network that is based on the mlready dataset. The input dimension defines the input dimension for the next layer in the network. The network configuration is initialized by providing the mlconfig file [14] that holds information about the network configuration, learning and training parameters.

Once the model configuration is loaded using the mlconfig file, the training process can be started by defining the number of epochs, or by defining the early stopping criteria. The training process can be monitored by reading the training progress information. The information helps the user to decide the training process converging at the expected speed, or when to stop the training process in order to prevent model overfitting. The training module that shows training history is shown in Fig. 4.21. The model deployment is the last phase of the ML cloud solution, and defines several options that can be used for different scenarios. The most common option is to generate a simple web service that contains the implementation of

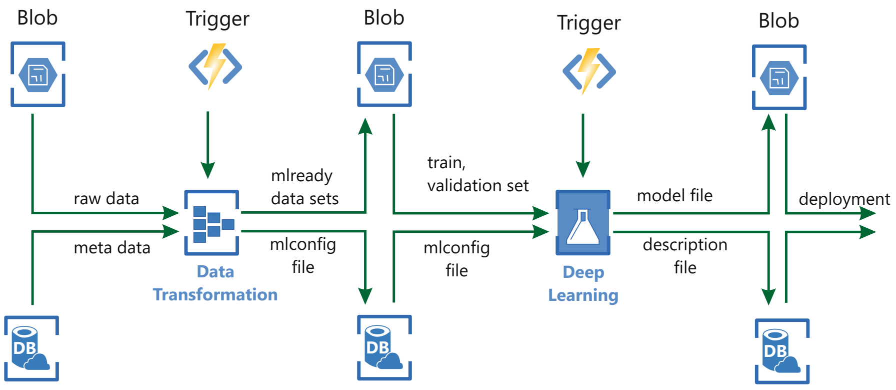


Fig. 4.22 Architecture of an ML cloud solution

the model evaluation. The web service returns the model output in an appropriate format. The model can also be deployed in Excel, to allow the model to behave as an Excel formula. Excel deployment is achieved by implementing additional Excel add-in. The deployment ML model in Excel is usually suitable when dealing with the input data which is relatively easy to represent in Excel.

The complete cloud ML solution is depicted in Fig. 4.22. To implement such a cloud solution, Microsoft Azure Cloud [21] platform can be used. By using the ANNdotNET open source computer program, it is possible to transform data and prepare it for training. Moreover, ANNdotNET provides components for training, evaluation, testing, and deploying deep learning models. Its components can be used in similar cloud solutions depicted in Fig. 4.22, particularly for "data transformation" and "deep learning" cloud solution components.

References

1. Al-Turjman, F., & Alturjman, S. (2018). Context-sensitive access in industrial internet of things (IIoT) healthcare applications. *IEEE Transactions on Industrial Informatics, 14*(6), 2736–2744. https://doi.org/10.1109/TII.2018.2808190
2. Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks, 5*(2), 157–166. https://doi.org/10.1109/72.279181
3. Bonissone, P. P. (2015). *Springer handbook of computational intelligence*. https://doi.org/10.1007/978-3-662-43505-2
4. Cao, Q., Ewing, B.T., & Thompson, M.A. (2012). Forecasting wind speed with recurrent neural networks. *European Journal of Operational Research, 221*(1), 148–154. https://doi.org/10.1016/j.ejor.2012.02.042
5. Cleveland, R. B., Cleveland, W. S., McRae, J. E., & Terpenning, I. (1990). STL: A seasonal-trend decomposition procedure based on loess. *Journal of Official Statistics*. https://doi.org/citeulike-article-id:1435502

6. Danandeh Mehr, A. (2018). An improved gene expression programming model for streamflow forecasting in intermittent streams. *Journal of Hydrology, 563*, 669–678.
7. Dokumentov, A., & Hyndman, R. J. (2015). STR: A seasonal-trend decomposition procedure based on regression, Department of Econometrics and Business Statistics, Monash University.
8. Gers, F. A., Schraudolph, N. N., & Schmidhuber, J. (2002). Learning precise timing with LSTM recurrent networks. *Journal of Machine Learning Research, 3*(1), 115–143. https://doi.org/10.1162/153244303768966139
9. Graves, A., Mohamed, A., & Hinton, G. (2013). Speech recognition with deep recurrent neural networks. In *IEEE International Conference on Acoustics, Speech and Signal Processing* (Vol. 3, pp. 6645–6649). https://doi.org/10.1109/ICASSP.2013.6638947
10. Hermans, M., & Schrauwen, B. (2013). Training and analyzing deep recurrent neural networks. *NIPS* 2013.
11. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation, 9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735
12. Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the National Academy of Sciences, 79*(8), 2554–2558. https://doi.org/10.1073/pnas.79.8.2554
13. Hoptroff, R. G. (1993). The principles and practice of time series forecasting and business modelling using neural nets. *Neural Computing Applications, 1*(1), 59–66. https://doi.org/10.1007/BF01411375
14. Hrnjica, B. (2018). ANNdotNET- deep learning tool on .Net platform https://doi.org/10.5281/ZENODO.1756095
15. Hrnjica, B., & Danandeh Mehr, A. (2018). Optimized genetic programming applications. IGI Global. https://doi.org/10.4018/978-1-5225-6005-0
16. Hyndman, R. J., & Athanasopoulos, G. (2018). *Forecasting: Principles and practice* (2nd ed.). Melbourne: OTexts. http://OTexts.com/fpp2. Accessed 1 Feb 2019.
17. Kaastra, I., & Boyd, M. (1996). Designing a neural network for forecasting financial and economic time series. *Neurocomputing, 10*(3), 215–236. https://doi.org/10.1016/0925-2312(95)00039-9
18. Lee, T. L. (2008). Back-propagation neural network for the prediction of the short-term storm surge in Taichung harbor, Taiwan. *Engineering Applications of Artificial Intelligence, 21*(1), 63–72. https://doi.org/10.1016/j.engappai.2007.03.002
19. Lu, Y., & Salem, F. M. (2017). Simplified gating in long short-term memory (LSTM) recurrent neural networks. CoRR, abs/1701.0, 5. https://doi.org/10.1109/MWSCAS.2017.8053244
20. Mehrotra, K., Mohan, C. K., & Ranka, S. (1997). *Elements of artificial neural networks*, A Bradford Book (The MIT Press, Cambridge)
21. Microsoft. (2015). Microsoft azure. https://doi.org/10.1007/978-1-4842-1043-7
22. Muhammad, K., Ahmad, J., Lv, Z., Bellavista, P., Yang, P., & Baik, S. W. (2018). Efficient deep CNN-based fire detection and localization in video surveillance applications. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 1–16. https://doi.org/10.1109/TSMC.2018.2830099
23. Muhammad, K., Hussain, T., & Baik, S. W. (2018). Efficient CNN based summarization of surveillance videos for resource-constrained devices. *Pattern Recognition Letters*. https://doi.org/10.1016/J.PATREC.2018.08.003
24. Pineda, F. J. (1987). Generalization of back-propagation to recurrent neural networks. *Physical Review Letters, 59*(19), 2229–2232. https://doi.org/10.1103/PhysRevLett.59.2229
25. Rodriguez, C. P., & Anders, G. J. (2004). Energy price forecasting in the Ontario competitive power system market. *IEEE Transactions on Power Systems, 19*(1), 366–374. https://doi.org/10.1109/TPWRS.2003.821470
26. Rosenblatt, F. (1960). Perceptron simulation experiments. *Proceedings of the IRE, 48*(3), 301–309. https://doi.org/10.1109/JRPROC.1960.287598
27. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature, 323*(6088), 533–536. https://doi.org/10.1038/323533a0

28. Russell, S., & Norvig, P. (2015). *Artificial intelligence a modern approach* (3rd edn.). London: Pearson Education.
29. Sak, H., Senior, A., & Beaufays, F. (2014). Long short-term memory recurrent neural network architectures for large scale acoustic modeling. Interspeech 2014, (September), pp. 338–342. https://doi.org/arXiv:1402.1128
30. Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks, 61*, 85–117. https://doi.org/10.1016/j.neunet.2014.09.003
31. Taylor, S. J., & Letham, B. (2018). Forecasting at Scale. *American Statistician, 72*, 37–45. https://doi.org/10.1080/00031305.2017.1380080
32. Ullah, A., Muhammad, K., Del Ser, J., Baik, S. W., & Albuquerque, V. (2018). Activity recognition using temporal optical flow convolutional features and multi-layer LSTM. *IEEE Transactions on Industrial Electronics*. https://doi.org/10.1109/TIE.2018.2881943
33. Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., & Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*. https://doi.org/10.1111/1467-8535.00290
34. Yu, D., Eversole, A., Seltzer, M., Yao, K., Kuchaiev, O., Zhang, et al. (2014). An introduction to computational networks and the computational network toolkit. Microsoft Research.