Heating Energy Consumption Forecasting Based on Machine Learning

HAMK
HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Bachelor's thesis

Degree Programme in Electrical and Automation Engineering

Spring 2018

Igor Trotskii

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Electrical and Automation Engineering
Valkeakoski

Author Igor Trotskii **Year** 2018

Subject Heating Energy Consumption Forecasting Based on Machine Learning

Supervisor(s) Jukka Pulkkinen

ABSTRACT

The author's aim in this thesis project was to develop a machine learning model, which could create short-term forecasts regarding heating energy consumption of a building. Even short-term energy consumption forecasts can have a major impact on building automation and energy distribution systems. Possible application spheres include smart grid development and simpler maintenance.

A feed forward artificial neural network was designed as a result of examination and testing of different models in order to get the most accurate predictions possible. To create an effective neural network various loss and activation functions as well as optimizers were reviewed.

To obtain better results some preprocessing techniques were applied to filter corrupted and unreliable data. The designed model was successfully trained to perform forecasting on data from the same distribution as the training data.

Keywords Time-series, forecasting, energy consumption, feed forward ANN

Pages 31 pages including appendices 40 pages

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The IoT sphere is experiencing a period of major development as the number of IoT devices increases rapidly and has already reached 23 billion. As a result, enormous amounts of data are being generated every second and classical data analytics tools are not nearly enough to handle it. To get the most from all collected information, machine learning techniques are used.

Machine Learning has tremendously evolved with stepping into the big data era. Such spheres as: Deep Learning, Computer Vision, Natural Language Processing have already brought a revolution into the information economy. But in spite of all technological advances in Machine Learning, it still has a long way to go in building automation and the aim of this thesis project was to look into the existing technologies used mainly for financial analysis and find their possible applications in the building automation domain.

Utilizing machine learning in building automation and in heating energy consumption forecasts particularly, can lead to many benefits, e.g. an easier deployment of Smart Grid energy distribution systems and better maintenance.

2 THEORETICAL FOUNDATIONS

2.1 Time series

Time series is a series of observations happening continuously over equal periods of time: year, week, days or minutes, depending on the characteristics of observable variable. Usually time series consists of several components:

- Trend – common long-term tendency of change of time series, which lies in the basis of its dynamics.
- Seasonal variance – short-term regularly appearing fluctuations of the observable variable over trend.
- Noise – chaotic fluctuations in a variable, caused by errors in measurements or by chaotic and unpredictable nature of the observable variable itself. (Hyndman & Athanasopoulos, 2018)

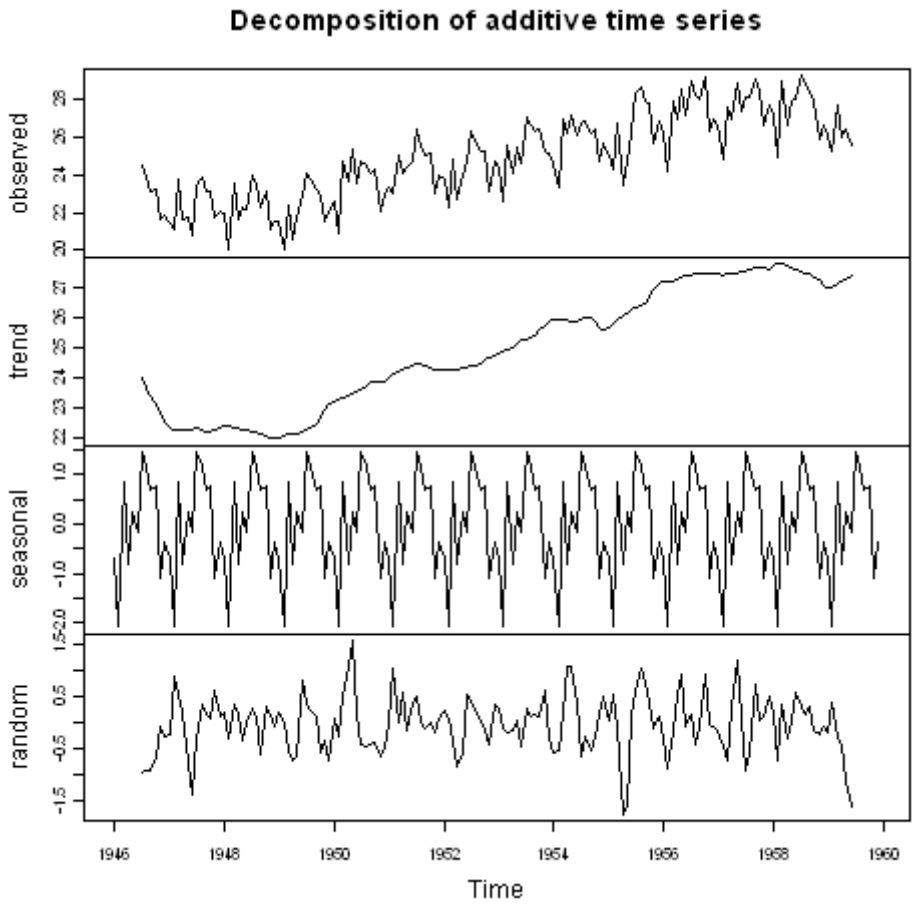

Figure 1. Decomposition of additive time series (A little book of R for time series, n.d.)

2.2 **Machine learning**

Machine learning is a field of computer science that uses statistical techniques to give computer systems the ability to “learn” (i.e. progressively improve their performance on a specific task) with data, without being explicitly programmed. (Samuel, 1959)

Machine learning is a class of methods of artificial intelligence, characteristic trait of which is not a direct solution of the task, but learning for a concrete task, based on set of similar tasks.

In general, the task for a machine learning system can be formulated as follows: there is a set of objects (situations) and a set of possible answers (responses). At the same time, certain dependence exists between these sets, which is unknown. If some finite ensemble of precedents (pairs of object – response), which is called a training set or a sample, obtained then it is possible to recreate this relationship between input and response to this object. This relationship is not necessarily expressed in an analytical form, often it is an empirically formed answer, which basically looks like a “black box” for a human. An important trait of the trainable system is an ability to generalize out of given examples, i.e. an ability to generate reasonable response to new, but related input data.

2.2.1 Supervised learning.

Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-outputs pairs (Russell & Norvig, 2010). In this kind of tasks, certain relationship between input and output (stimulus and response) exists, which is unknown. The aim is to learn the relationship by using known inputs and outputs. Special metrics, specific for each task, are used to determine the successfulness of the learned solution. From a cybernetics point of view, it appears to be one of the types of cybernetic experiment with feedback.

Input data can be represented by several types of data:

- Features – most common scenario. Every object is described with its set of characteristics. Characteristics can be numerical or categorical.
- Distance matrix – every object is described with distances to other objects of a training set. Most commonly used for classification problems.
- Time series – sequence of values. Each dimension can be a number, vector or feature in general.

- Image or video – an image can be represented as a set of features and a video as a time series.
- Some more complex data types like graphs, texts, results of database queries. Usually they are converted to simpler data types like features or distance matrices by pre-processing.

The output of a system can be characterized in the following way:

- If a set of possible answers is infinite, then it is a regression or approximation problem.
- If the set of possible answers is finite, then it is a classification problem or object recognition task.
- If the set of possible answers represents the future state of an observable system, then it is a forecasting problem.

To solve a given task of supervised learning the following steps should be carried out:

1. Training data should be determined. The data must be representative of the real-world of the function, which is going to be learned.
2. Collect the training set, which consists of input objects and corresponding output responses.
3. Determine the input feature representation of trainable model. Depending on a task, its complexity and amount of data to train on, different features have various effect on learning, e.g. if only small amount of training data is available, big number of features can lead to overfitting of the model.
4. Select the structure of the learned function and corresponding learning algorithm.
5. Training and fine-tuning of the model. Most of the models have hyper-parameters, which should be tuned for each task. Usually models are tuned by using special subset of training data, called validation set.
6. Evaluation of the model. Depending on the task, different metrics are used to evaluate the model, e.g. accuracy, error, recall, precision.

2.2.2 Unsupervised learning

Unsupervised learning is one of the methods of machine learning, in which a trainable system randomly learns to perform a given task without any intervention from the experimenter. (Duda, Hart, & Stork, 2000) Like supervised learning it can be classified as a type of cybernetic experiments, but without feedback.

Usually unsupervised learning can only be applied to tasks, which have feature-rich data, and which require determining some internal

relationships. Common tasks include clustering and anomaly detection. Unsupervised models do not have any evaluation metric, because there is no “ground-truth” labels for the input data.

2.3 **Statistical autoregressive models**

2.3.1 AR model

An autoregressive model is a model used for the time series analysis, in which the current value of a sequence linearly depends on previous values of the same sequence. An autoregressive process of the order *p* is defined as:

$$X_t = c + \sum_{i=1}^{p} a_i X_{t-1} + \mathcal{E}_t \quad (1)$$

Where $a_1, \dots, a_p$ are parameters of the model (coefficients of autoregression), *c* is constant and $\mathcal{E}_t$ is white noise. (Mills, 1991)

The autoregressive model has a range of limitations:

- It allows to analyze only one time series, which means that in order to get meaningful results with AR model, the studied system must be a closed system with no outside influence.
- It is linear in nature, so it can only approximate non-linear dependencies.
- Can only be applied to stationary data.
- It is unable to analyze errors or seasonal oscillation of data.

2.3.2 ARIMA model

ARIMA - Autoregressive integrated moving average model is a model obtained from combination of AR with Moving Average (MA) models.

Autoregressive integrated moving average models are usually applied when data shows evidence of non-stationarity, where an initial differencing step (corresponding to the “integrated” part of the model) can be applied one or more times. (Hyndman & Athanasopoulos, 2018)

ARIMA model with parameters (*p*, *d*, *q*), where *p* is the order (number of lags) of autoregression part, *d* – is a number of differencing steps, and *q* is the order of the moving-average model, can be defined as:

$$\left(1 - \sum_{i=1}^{p} \varphi_i L_i\right)(1 - L)^d X_t = \left(1 + \sum_{i=1}^{q} \theta_i L^i\right)\varepsilon_t \quad (2)$$

Where φ_i – coefficients of autoregression, θ_i – coefficients of moving averages and *L* is a Lag operator.

ARIMA model is more powerful than pure AR model:

- It enables analysis of non-stationary data.
- It considers not only previous values of the timeseries, but also its small fluctuations, errors.
- It is still linear in nature, but because of combination of AR and MA models can provide non-linear output.

Nonetheless, certain limitations still exist:

- Only one time series is considered, so it is still a single input model.
- It is unusual for an ARIMA model to represent any very complicated relationships in data.

2.3.3 ARIMAX model

ARIMAX extends the ARIMA model trough an inclusion of exogenous variables.

An exogenous variable is a variable that is not affected by any other variables in the system, e.g. energy consumption of the building is endogenous variable and outdoor temperature is exogenous variable. (Kongcharoen & Kruangpradit, 2013)

In comparison to ARIMA model, ARIMAX has several advantages:

- ARIMAX can take as an input multiple different time series and detect relationships between endogenous and exogenous data.
- Because models based on ARIMAX consider more data, they are more accurate.

However, certain major limitations still exist:

- An endogenous variable cannot have any effect on an exogenous variable.
- Several models should be used to get a single multi-output model.

These limitations can be avoided if neural network-based models are used.

2.4 **Deep learning models**

2.4.1 Fully Connected Feed Forward network

An artificial neural network (ANN) is a mathematical model built on principles of organization and functioning of biological neural network. The first try of creating ANN belongs to McCulloch and Walter (S. McCulloch & Pitts, 1943). After the development of learning algorithms ANN started to be used in practice for forecasting, object recognition and detection et al.

An artificial neural network represents a system of connected and interacting simple units, called neurons. These units are usually very simple, they only work with received signals performing simple mathematical operations, nevertheless, being connected into big network, neurons can represent very complicated functions and relationships. Example of a simple neural network is illustrated in Figure 2.

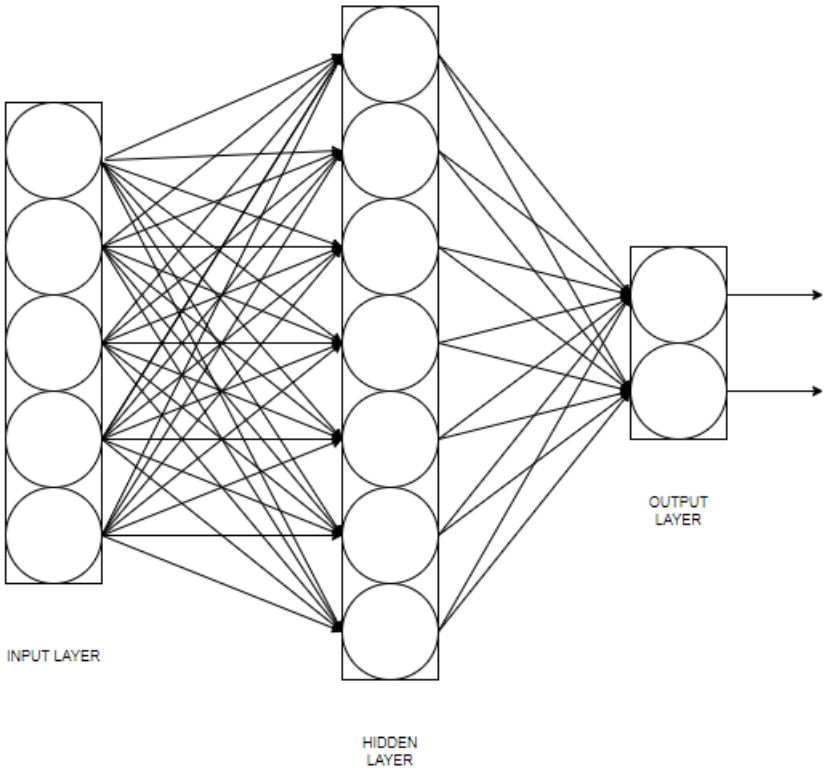

Figure 2. Fully Connected Feed Forward network, graphical representation

Neural networks are not programmed but trained. The possibility of training is one of the major advantages over traditional algorithms. Technically learning is finding coefficients of bonds between neurons,

which is multiparametric task of non-linear optimization. In the process of training network is able to detect complex relationships between input and output data and generalize it for a new data.

Typical behavior of a feed forward neural network can be described as follows:

1. Forward pass – calculation of the network's output by traversing through all neurons from input layer to output layer.
2. Backpropagation pass – calculation of the gradients, which are used to update weights (parameters) of the network (de facto learning). It is done by traversing through all neurons from output layer to input layer.

2.4.2 Recurrent Neural Networks

A recurrent neural network (RNN) is a class of neural networks where connections between the units form a directed sequence. This allows to process time series data and unlike usual feed forward neural networks, RNN-s can use their built-in memory to analyze sequences with different lengths. (Li & Wu, 2014)

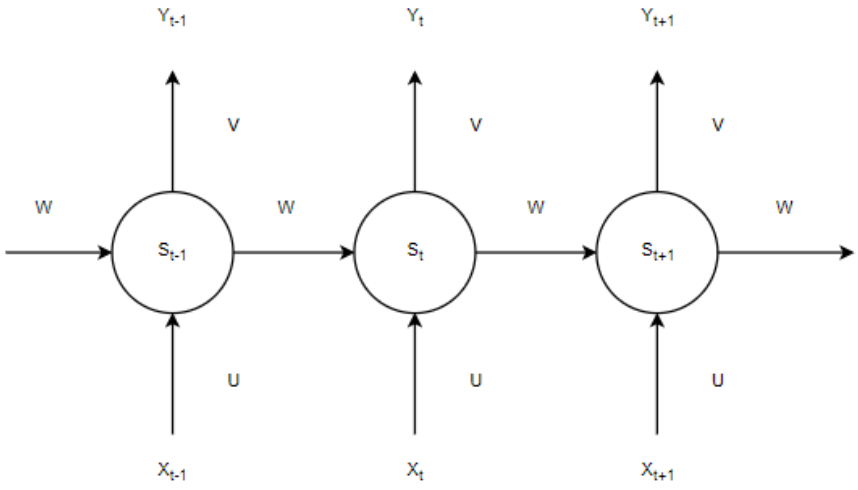

Figure 3. Unrolled RNN structure

The structure of a simple RNN is shown in Figure 3:

- X_t is the input to the network at time step *t*.
- S_t, is the hidden state at time step *t*, calculated as follows: $S_t = g(UX_t + WS_{t-1})$, *g(x)* is an activation function.
- Y_t is the output of the network at time step *t* equals to $Y_t = f(VS_t)$, *f(x)* is an activation function.
- *W*, *V*, *U*, are parameters of the network.

RNN are widely used in Natural Language Processing in speech recognition, machine translation, handwriting recognition.

In comparison with feed forward ANN, RNN have several major advantages:

- RNN usually has a lot less parameters to train, because most of them are shared among all units.
- RNN can handle time series of variable lengths.

To be efficient RNN requires to have a lot of separate units to be stacked in a sequence, what brings one major flaw – vanishing gradients. Simple RNN are unable to capture long-term dependencies between data points. One of the possible solutions is using LSTM units.

2.4.3 LSTM networks

A Long-Short Term Memory (LSTM) block is a building unit for layers of a recurrent neural network. A LSTM unit contains a cell, an input gate, an output gate and a forget gate. Unlike a traditional RNN, LSTM networks are well suited for processing and analysing time series even when sequences contain long-term dependencies.

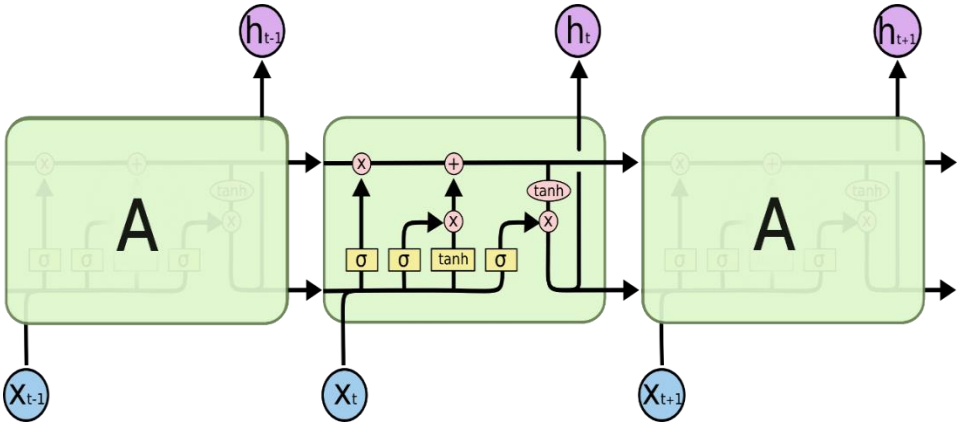

Figure 4. LSTM unit structure (Olah, 2015)

In Figure 4, the basic working principle of LSTM unit can be seen. Instead of only passing hidden state (activation) to the next unit as usual RNN block, LSTM handles two lines of data: cell state, which is responsible for long-term dependencies, and hidden state. And unlike basic RNN, LSTM unit basically is built out of 4 different layers, each of which controls corresponding gate.

- Forget gate. This gate is used to remove information from cell state. It is usually controlled with hard-sigmoid activation function, which has an output range of 0-1, so if some piece of data should be forgotten, it is multiplied by 0, if it should be kept - then by 1.
- Input gate controlled by two layers. One of them controls which cell states should be updated and has hard-sigmoid as an activation function. Another uses hyperbolic tangent.
- Output gate is used to pass information from cell state to hidden state.

It should be noted, that the cell state line is continuous, so it is very easy for information and gradients to flow without any changes, which gives LSTM the amazing ability to determine long-term relationships and helps to eliminate vanishing gradients problem.

LSTM nets are a much more powerful tool than a simple RNN and it is no surprise that LSTM-s are very popular in machine learning tasks related to sequence processing, such as time series prediction, speech recognition, music composition, sign language translation, time series anomaly detection and many others. (Li & Wu, 2014)

2.5 **Optimization**

2.5.1 Loss function

In the theory of point estimation, a loss function (objective function in general) quantifies the losses associated to the errors committed while estimating a parameter. (Loss Function, n.d.)

Depending on the task, different loss functions are preferred, e.g. regression or classification require completely different objective function. It is vital to select correct function for each task because it basically represents the objective of the model to learn.

Most of the common loss function can be divided into two groups depending on the type of task they are solving: regression or classification.

Loss functions used for classification:

1. Cross-entropy loss is the most common loss function for classification tasks. It is defined as

$$L(a, y) = \frac{1}{n}\sum_{i=1}^{n}(y_i \ln(a_i) + (1 - y_i)\ln(1 - a_i)) \quad (3)$$

where *a* is an output of the model in the range of 0 to 1, and *y* is ground-truth label for this particular data point, which is either 0 or 1.

2. Hinge loss. It is used for "maximum-margin" classification. For a labelled output $t = \pm 1$ and a classifier output score *y,* the hinge loss is defined as

$$L(t, y) = \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - t_i y_i) \quad (4)$$

It can be seen, that when *t* and *y* have the same sign (classification is correct), the loss is 0, and if classification is incorrect, then the loss increases linearly with *y*.

Loss functions for regression are more diverse:

1. Mean Absolute Error (MAE) is a measure of difference between two continuous variables. It is defined as

$$MAE = L(a, y) = \frac{1}{n}\sum_{i=1}^{n} |y_i - a_i| \quad (4)$$

Where y_i is a desirable output and a_i is an actual output of a model, *i* is the number of a data point from training set.

MAE uses the same scale as the data being analysed, which is known as scale-dependent accuracy, so it is impossible to compare results between data with different scales. MAE is used, then it is not necessary to additionally penalize big error in estimations.

2. Root Mean Squared Error (RMSE) measures the average of the errors or the difference between desirable and actual output of the model. Like Mean Absolute Error has the same units of measurement as quantity being estimated. It can be defined as follows

$$RMSE = L(a, y) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} (y_i - a_i)^2} \quad (5)$$

RMSE is preferable, when higher error means much bigger harm as this loss function additionally penalizes large error values.

3. Root Mean Squared Logarithmic Error (RMSLE) is defined as

$$RMSLE = L(a, y) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(ln\left(\frac{a_i + 1}{y_i + 1}\right)\right)^2} \quad (6)$$

RMSLE is used when underprediction is more undesirable than overpredicting, that means if model's output is smaller than actual value, the loss is going to be larger, than if the output was instead bigger by the same margin. It can also be used then both true and predicted values expected to big, because for RMSLE only percental difference matters.

2.5.2 Activation functions

An activation function is a function, which defines the output of a node given a set of inputs. It is a vital part of any neural network. It is activation function's responsibility to bring non-linearity to neural networks. (Haykin, 1999)

As it was mentioned earlier, any network has two steps to perform during one iteration: forward propagation and back propagation. During forward propagation each layer (it is "layer" here and not "neuron" because of vector notation) calculates a weighted sum of all received inputs:

$$Z_i = W_i a_{i-1} + b_i \quad (7)$$

Where Z_i is a matrix of weighted sums for each neuron of layer *i*, W_i is a weight matrix, a_{i-1} is an activation vector from previous layer and b_i is a bias vector.

The next step is to find the activation of this weighted sum:

$$a_i = g(Z_i) \quad (8)$$

Where a_i is an activation of layer *i* and *g* is an activation function.

If there is no activation function or it is just linear then the layer doesn't contribute any computational power to the network. It can be demonstrated with using a simple ANN as an example:

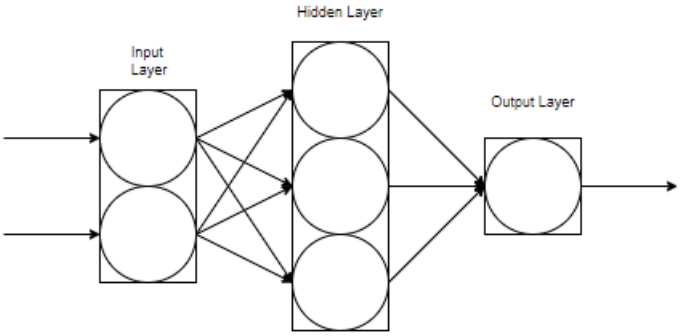

Figure 5. Simple feed forward neural network

The network in Figure 5 is a small feed forward ANN with two layers: one hidden layer and one output layer. Its whole computation for forward propagation can be described with the following equations:

1. Computing weighted sum for hidden layer.

$$Z_1 = W_1 x + b_1 \tag{9}$$

Where *Z* is weighted sum, *W* is a weights matrix or parameters of the layer, *b* is a bias vector and *x* is an input to the network.

2. Computing weighted sum for output layer. Because there is no activation function for the hidden layer, following equation is obtained:

$$Z_2 = W_2 Z_1 + b_2 = W_2(W_1 x + b_1) + b_1 = W_2 W_1 x + W_2 b_1 + b_1 \tag{10}$$

In that equation $W_2 W_1$ term can be replaced with equivalent matrix *W*, which is equal to matrix product $W_2 W_1$. $W_2 b_1$ can be replaced as well in the similar way. These transformations result in a new structure with only one layer – output and it can no longer be considered neural network as it behaves as usual linear regression. This shows that proper activation function is vital for a neural network to operate and learn properly.

Non-linearity is not the only requirement for an activation function, it must also be differentiable overwise finding gradients for weights of the network is not possible.

Considering these requirements the following activation functions are widely used:

1. Sigmoid function
   A sigmoid function is mathematical function having a characteristic "S"-shaped curve as illustrated in Figure 6. It is defined by equation:

$$S(x) = \frac{1}{1 + e^{-x}} \tag{11}$$

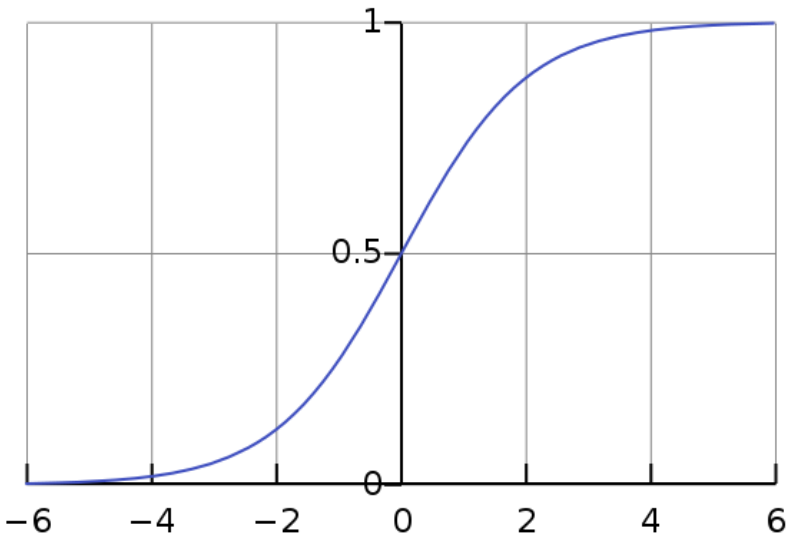

Figure 6. Sigmoid function

A sigmoid is historically one of the first activation functions used in building neural networks.
It is widely used in binary classification problems as it always produces output in the range of 0 to 1. Another reason of sigmoid's popularity is its simple derivative:

$$\frac{dS(x)}{d(x)} = S(x)\big(1 - S(x)\big) \tag{12}$$

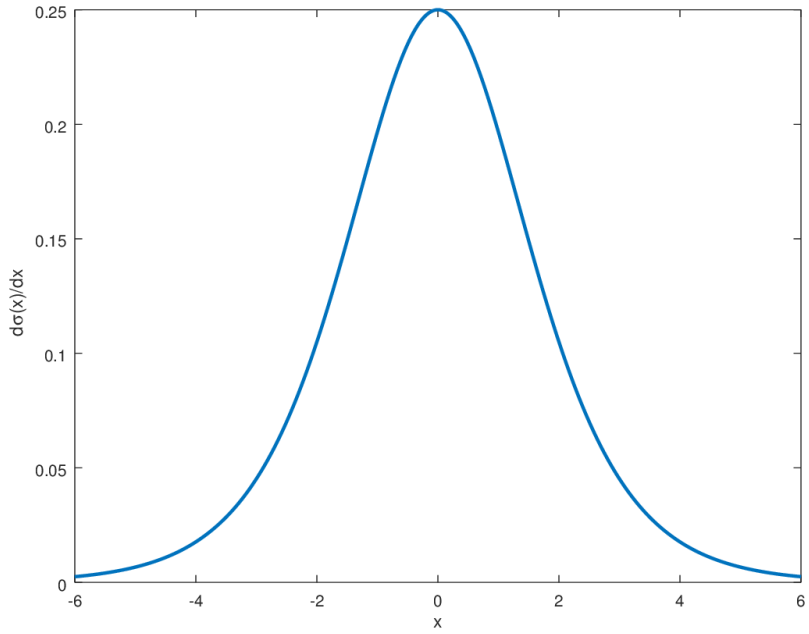

Figure 7. Derivative of sigmoid activation function

2. Hyperbolic tangent is a mathematical function defined as

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{13}$$

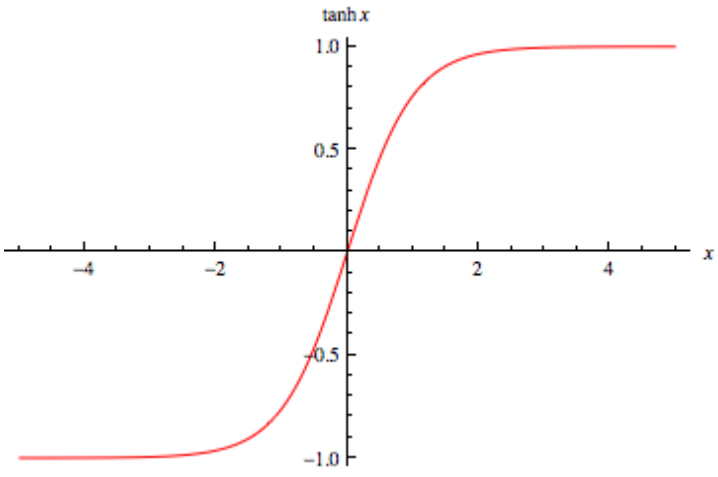

Figure 8. Hyperbolic tangent

Hyperbolic tangent is widely used activation function and it is often preferred to sigmoid in hidden layers as it is centered around zero, so it allows to save zero-mean an unit-variance. The second reason is hyperbolic tangent has “stronger” gradients, what can speed up training and slow down vanishing gradients. (LeCun, Bottou, B. Orr, & Müller, 2002)

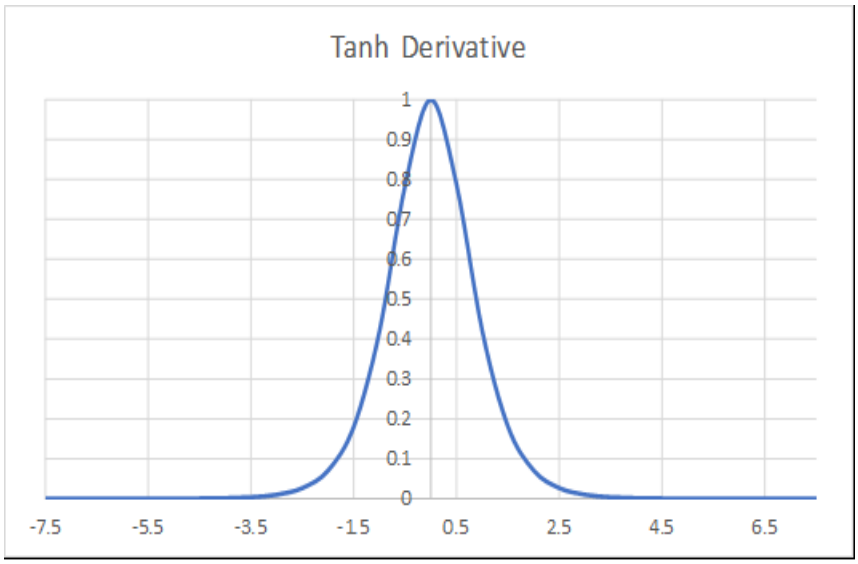

Figure 9. Derivative of hyperbolic tangent

3. Rectified Linear Unit – is an activation function defined as the positive part of its argument:

$$ReLU(x) = \max(0, x) \tag{14}$$

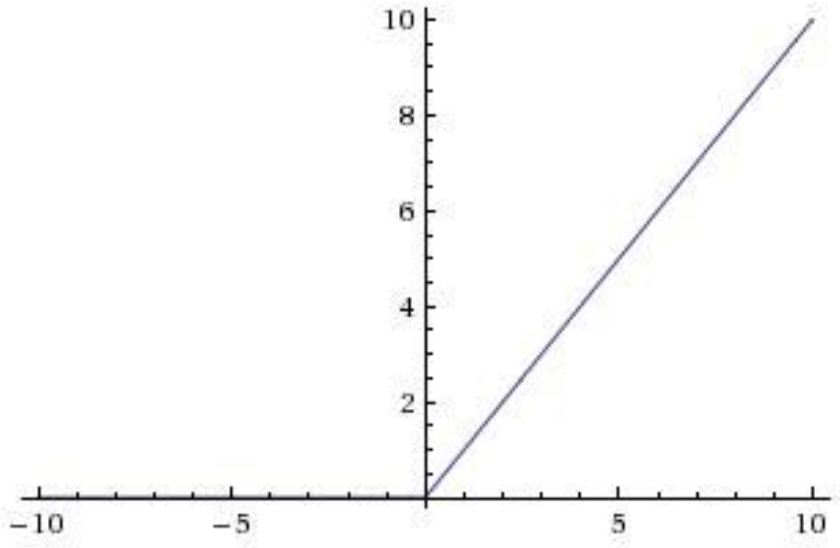

Figure 10. ReLU activation function

ReLU is the most popular activation function for very deep neural networks as it has fewer problems with vanishing gradients, which is the main limit for neural network depth, in comparison with sigmoidal activation functions.

Rectified Linear Unit is the most widely used activation function in computer vision domain as its tasks require very deep neural networks and involve large data sets (LeCun, Bengio, & Hinton, 2015). ReLU sometimes is used instead of linear activation function for output layer when negative output is undesirable.

2.5.3 Optimizers

Optimization is the selection of a best element with regards to some criterion from some set of available alternatives (B. Dantzig, et al., 2007). In a simple case optimization involves maximizing or minimizing some target function, in machine learning the role of a target function belongs to loss function.

There are many optimizing algorithms for neural networks, but most of them are based on simple gradient descent:

1. Gradient descent.

Gradient descent or batch gradient descent is a method of determination a local minima of a loss function by moving alongside the gradient as illustrated in Figure 11.

It can be defined with the following equation, which represents the update rule for weights of a network:

$$W_{i+1} = W_i - \gamma \nabla L(W_i) \tag{15}$$

Where W_i is weights of a network on iteration *i,* γ is a learning rate – hyperparameter, which determines the speed of learning, $\gamma \nabla L(W_i)$ is a gradient of a loss function with respect to current weights of a network. Gradient descent updates parameters once for all input data, meaning that all training data must be processed for one step of gradient descent.

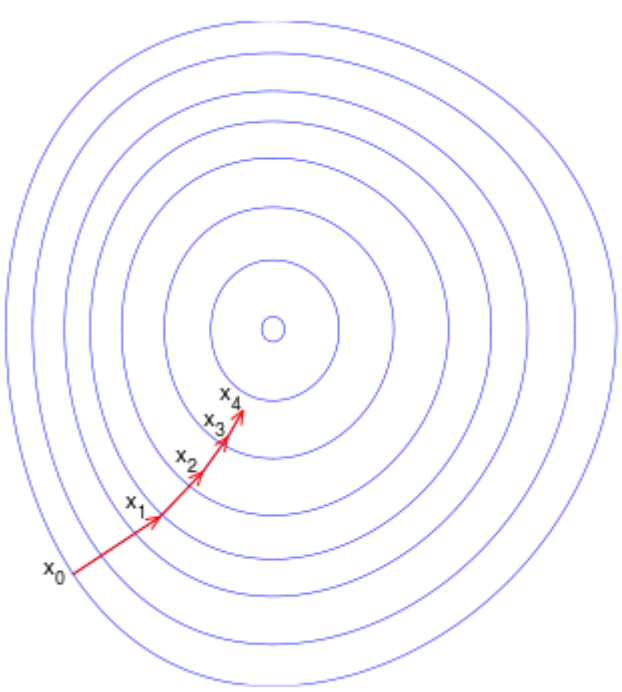

Figure 11. Gradient descent

Gradient descent is the simplest method of local optimization, it is not guaranteed, that it will find global minima and the speed of convergence is low. These factors limit the use of pure Gradient descent, but it still plays an enormous role in machine learning as it is used as a base for most of other optimization algorithms.

2. Stochastic gradient descent.

Stochastic gradient descent (SGD) – is an improvement over gradient descent. Another name for stochastic gradient descent is minibatch gradient descent as it updates the weights not for whole data set, but for small minibatch. This approach considerably increases speed of convergence of the algorithm if proper learning rate is used.

Because only a small fraction of data is taken into account for each update, SGD is very noisy. The noise leads to both favorable and disadvantageous consequences: this algorithm is more likely to find global minima than usual batch gradient descent, but it is not guaranteed to converge with big learning rate, so it should be reduced as training continues. As long as learning rate is controlled properly minibatch gradient descent can converge much faster than basic gradient descent and often ends up in a better local minima.

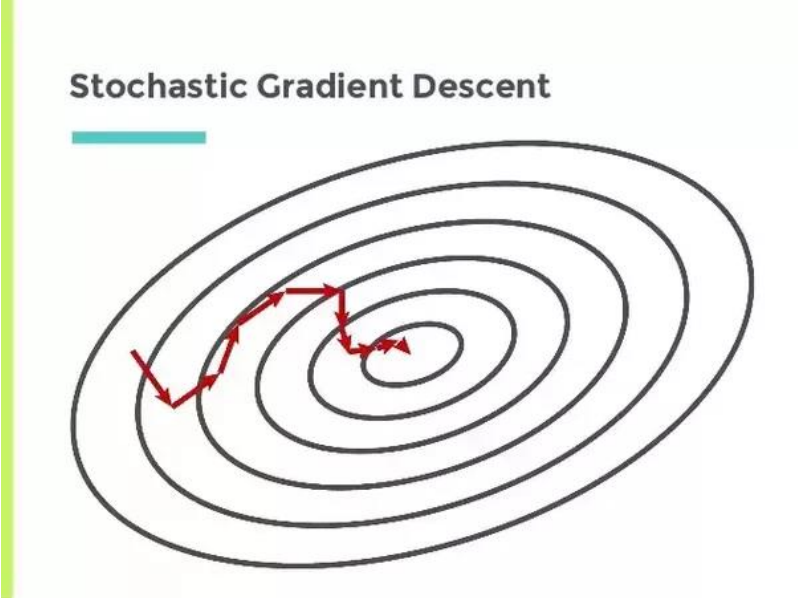

Figure 12. SGD (Fregly, 2016)

SGD has similar mathematical definition to Batch Gradient Descent:

$$W_{i+1} = W_i - \gamma \nabla L(W_i, x^n, y^n) \tag{16}$$

Where x^n, y^n are corresponding training data pairs inside one mini-batch.

As mentioned earlier high learning rates and noise negatively affect on convergence rate. Some other algorithms handle this problem by adding momentum to new weights calculations, e.g. Adam algorithm.

3. Adaptive Moment Estimation (Adam) – is an optimization algorithm, which uses running averages of both gradients and the second moments of the gradients. Given parameters w^t and a Loss function $L^{(t)}$, where t indexes the current training iteration, Adam's parameter update is given by:

$$m^{t+1} = \beta_1 m_w^t + (1 - \beta_1) \nabla L(w)^t \tag{17}$$

$$v^{t+1} = \beta_2 v_w^t + (1 - \beta_2) (\nabla L(w)^t)^2 \tag{18}$$

$$\widehat{m_w} = \frac{m^{t+1}}{1 - \beta_1} \tag{19}$$

$$\widehat{v_w} = \frac{v^{t+1}}{1 - \beta_2} \tag{20}$$

$$w^{t+1} = w^t - \gamma \frac{\widehat{m_w}}{\sqrt{\widehat{v_w}} + \mathcal{E}} \tag{21}$$

Where $\mathcal{E}$ is a small number used to prevent division by zero, and β_1 and β_2 are forgetting factors for gradients and second moments of gradients, respectively.

Adam combines the best from both worlds: it is able to converge a lot faster than SGD or Batch Gradient Descent. Thanks to momentum and mini-batch nature turns it is unlikely for this algorithm to stuck in non-optimal local minima. Currently it is one of the most popular and widely used optimization algorithms for training neural networks.

3 MODELLING

3.1 Data overview

In order to forecast the next 24 hours of heating energy consumption the following features were selected:

- Indoor temperature
- Outdoor temperature
- Energy consumed for air conditioning
- Energy consumed for ceiling radiant
- Energy consumed for floor heating

It should be noted that the database included a lot more parameters about the building used for the analysis. But considering the total duration of a recording – 4 months and very limited computational power, only the most influential features got selected for training.

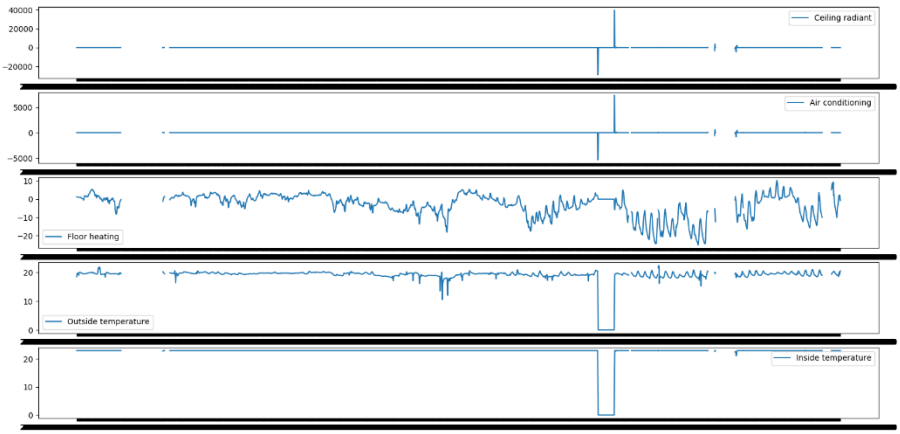

Figure 13. Data before pre-processing

3.2 Pre-processing

As can be seen in Figure 13, not all the data can be used for modelling. The first thing that catches the eye is the presence of several outliers for ceiling radiant and air conditioning energy consumptions and missing measurements for indoor and outdoor temperatures. The most probable reason for this is a lost connection to the server or a malfunction in the measurement devices themselves.

It was impossible to use the data in that state to train a model, that is why some pre-processing was required. To “clean” the data the following actions were undertaken:

1. All the “null” values were removed. If some of the parameters missed measurements, then all other parameter’s values corresponding to the same timestamp must be discarded.
2. All the outliers were removed. The data point was considered to be outlier or invalid point, if it had negative or higher than 100 kWh energy consumption.
3. Consistency check. As some data points were removed, the time series sequence was broken, meaning that the series was no longer continuous. If the data was to be used in that state, a model would try to learn on measurements from different days or even weeks, what does not represent the assigned task.

To address the aforementioned problem, the longest uninterrupted sequence was chosen. If from the very beginning more data had been available for training, then the neural network would be able to train on data with broken sequences, as these interruptions would have been almost unnoticeable for a model.

As a result of pre-processing out of 2800 points (4 months of reading with 1 hours sampling time) only 1464 remained, which was around half of the initially available data.

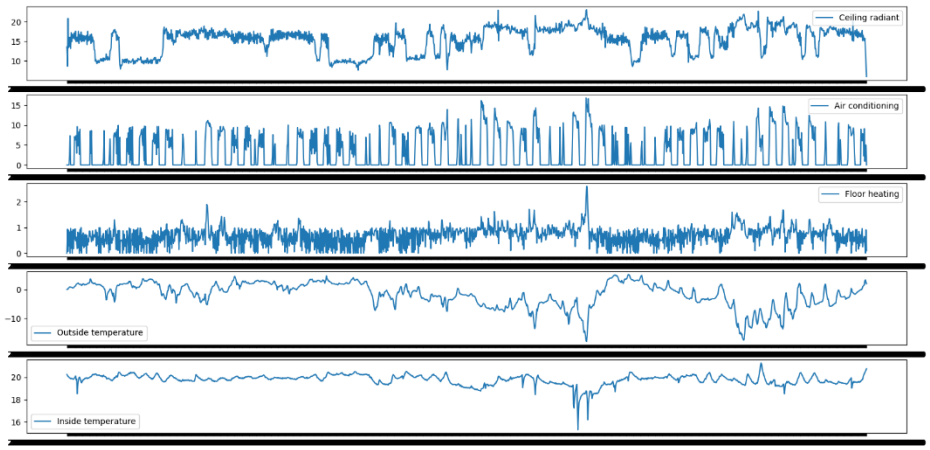

Figure 14. Preprocessed data

The next step was to transform the data from the time series format to a supervised one. Initially the data is a time series, which cannot be used with most of machine learning models.

Table 1. Example of data in time series form

Time	Ceiling Radiant
2017-12-13T20:00:00+02:00	9.86
2017-12-13T21:00:00+02:00	10.75
2017-12-13T22:00:00+02:00	2.125
2017-12-13T23:00:00+02:00	13.6
2017-12-14T00:00:00+02:00	8.6131

In a Table 1 a sample of data in time series format is shown. For simplicity only one parameter is taken into account.

A supervised learning problem is comprised of input values and output values, so the model can learn the function, which can map input data to output. In order to get these "labeled" output data for each point of time series, shift function is used: all observation are shifted by one time step, so next time step is a label for previous point.
In a table below the same time series is shown in the supervised form. In this example, three last measurements are used to predict next value.

Table 2. Time series in the supervised format

Ceiling Radiant (t-2)	Ceiling Radiant (t-1)	Ceiling Radiant (t)	Ceiling Radiant (t+1)
9.86	10.75	2.125	13.6
10.75	2.125	13.6	8.6131
2.125	13.6	8.6131	20.89
13.6	8.6131	20.89	13.83
8.6131	20.89	13.83	16.02

3.3 **Building a model**

To simulate heating energy consumption the next day the following feed forward network was designed.

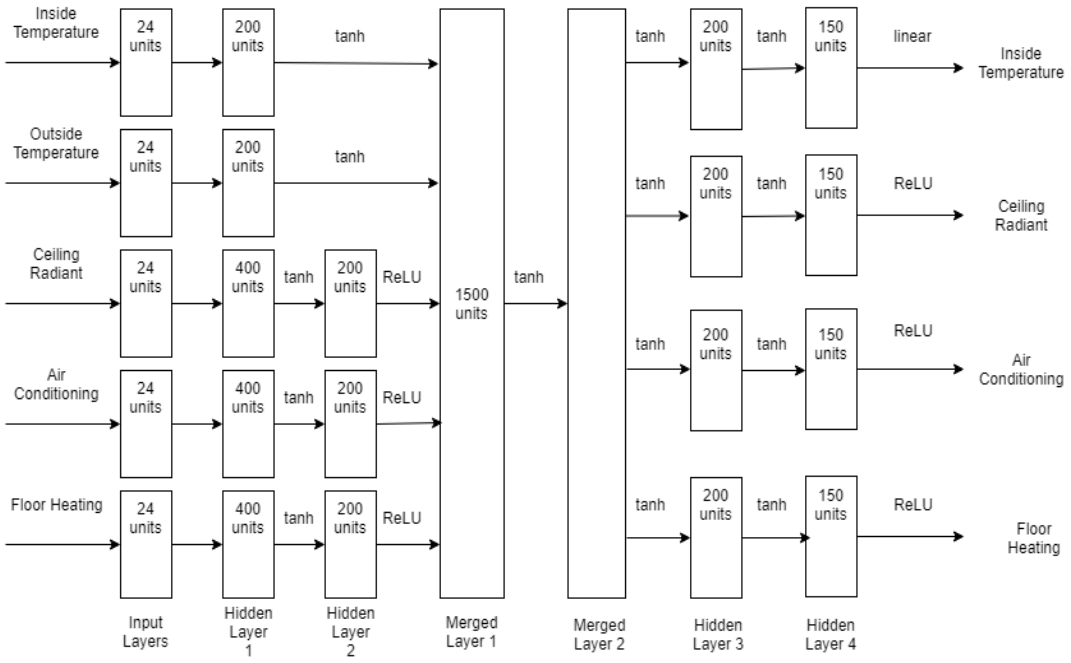

Figure 15. Structure of the model.

The motivation behind this exact structure is hard to explain as the model was mostly designed by trial and error. But some intuition in general can still be given:

- Usual feed forward ANN is used instead of sequential models like RNN or LSTM, because dimensionality of input sequences is always known. It is five vectors, containing 24 values each, which represent the data from the previous day for each feature.
- LSTM networks are harder to train: an LSTM unit has a lot of parameters to learn, meaning that more data is required to get the model working. If more data were available, it would be reasonable to use LSTM.
- Feed forward ANN is relatively inexpensive in terms of computational power, so no expensive hardware is required to train a model in reasonable amount of time.

This structure was inspired by NARX (Nonlinear Autoregressive exogenous model). The first unconnected layers have a role of autoregression, they transform the corresponding feature vector, containing measurements of 24 hours of some parameter, to some other vector with 200 values, which can be thought of as encoded time series. As it was assumed that these five parameters were related to each other, two merged layers were used. If some dependency existed, then it would be discovered in the merged layers. The last two layers were used to transform the found relationships into final predictions.

Adam optimizer was used for the training. The results are shown in Figure 16.

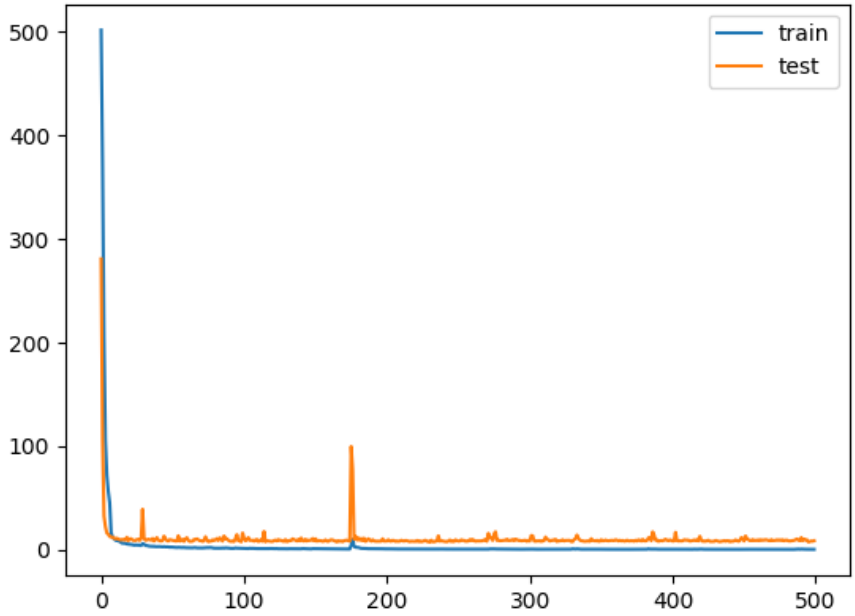

Figure 16. Loss function over epochs of training

Other optimizers were also tested, but they either performed worse or gave the same results, but took a lot more time to train. It should also be noted that the network is fully fitted for available data as the Loss function does not change for hundreds of epochs.

3.4 **Validation and testing**

The model was fully trained and could be tested. During the training the network predicted only values for the next hour, but 24 were required. To solve this problem a method called sliding windows was used as illustrated in Figure 17.

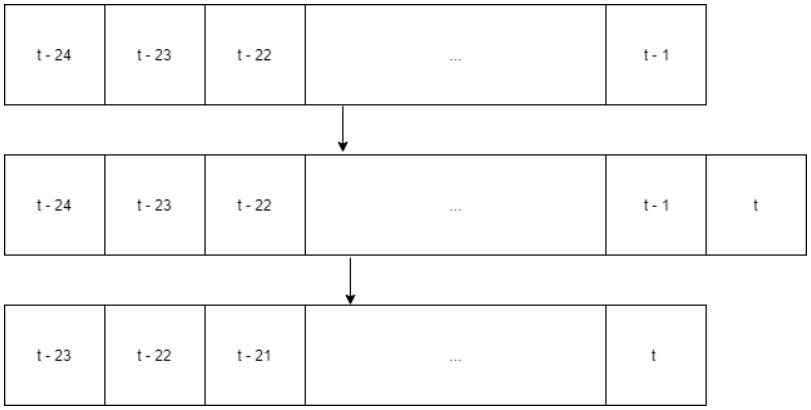

Figure 17. Using sliding windows method

In the Figure 17 the sliding windows method is shown in action. The model predicts values for the next hour and then appends them to the input data, removing the first value in the input sequences. This cycle is repeated several more times to get predictions for a full day.

The usage of sliding windows is also the reason why the prediction of the indoor temperature is important. The outdoor temperature can be obtained from weather forecasts, but it is not the same with the indoor temperature. In order to use it in predictions, the temperature should itself be predicted.

3.5 **Results**

The following results were achieved on a training set:

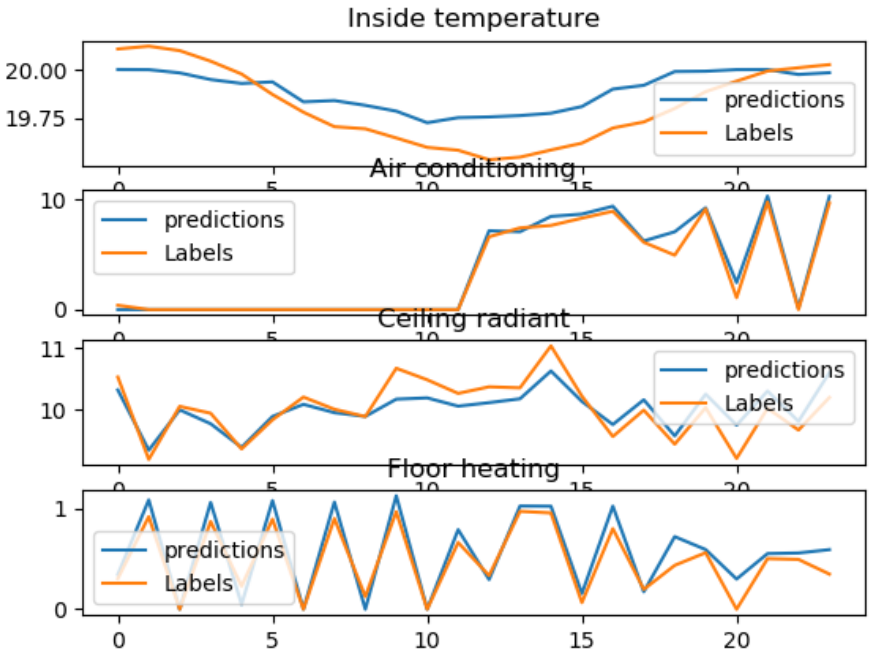

Figure 18. Predictions on training set 1

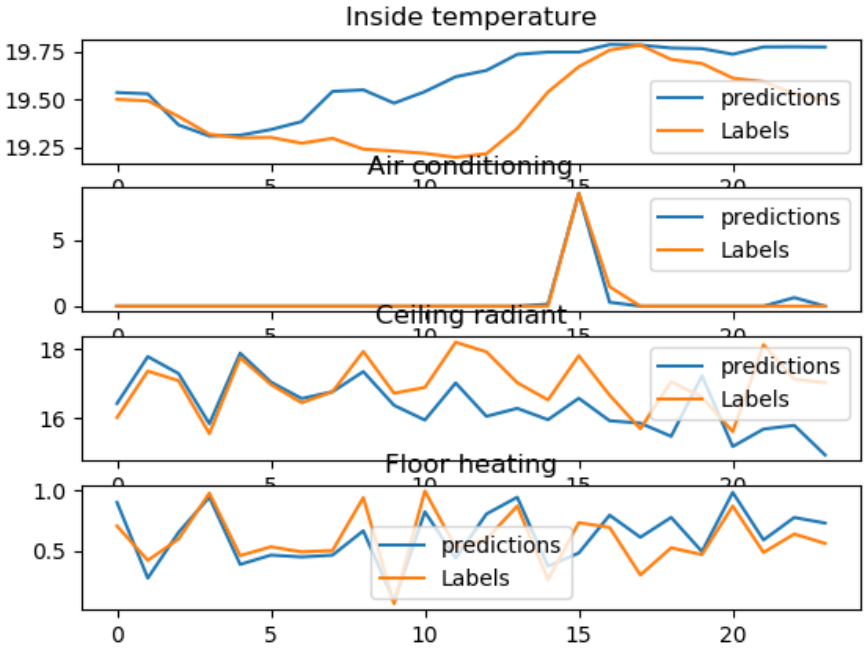

Figure 19. Predictions on training set 2

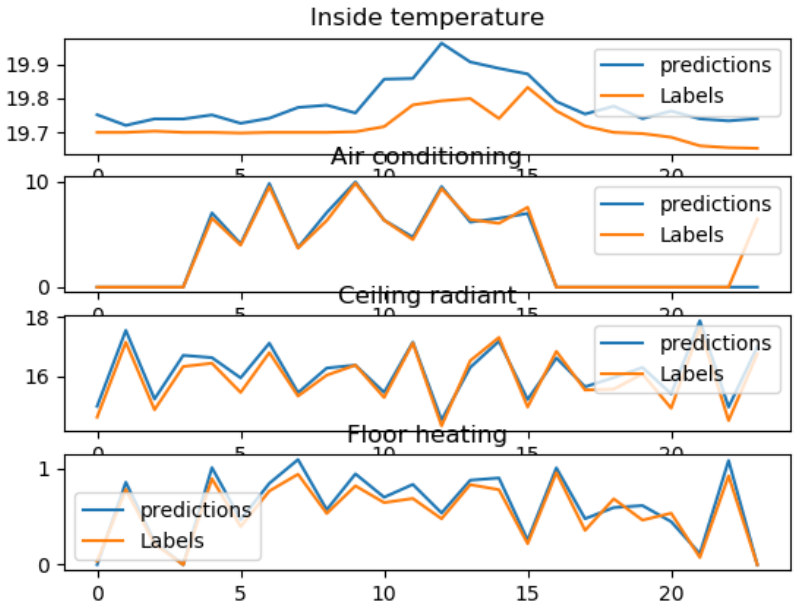

Figure 20. Predictions on training set 3

These results manifest that the model is able to fit all the required relationships and dependencies from the training set, so the selected features fully represent the modelled building in terms of heating energy consumption.

The following results were achieved on the test set:

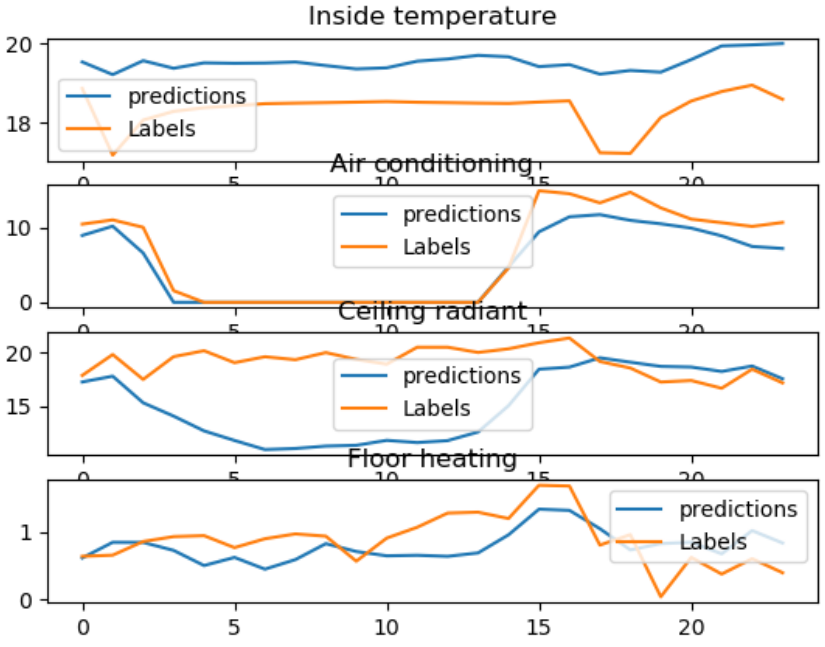

Figure 21. Predictions on test set 1

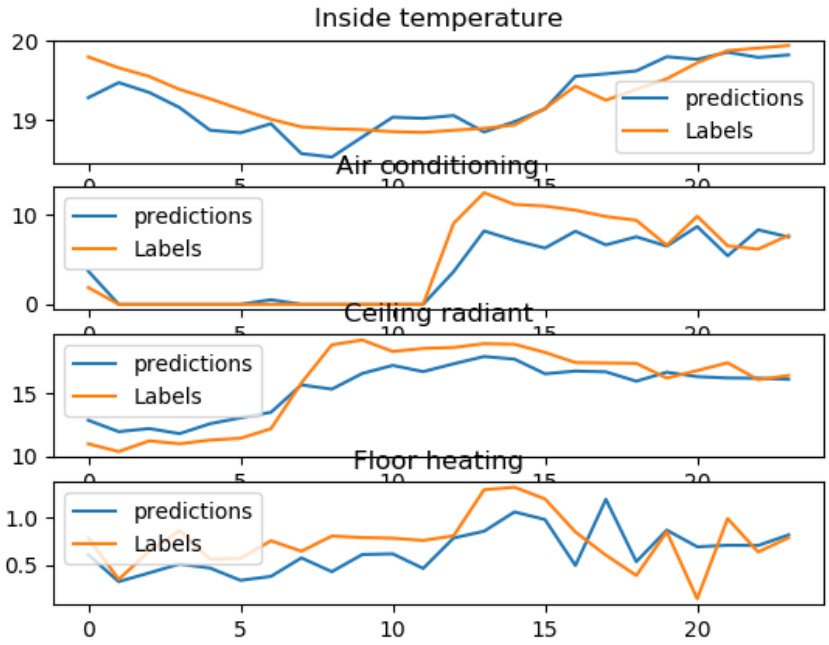

Figure 22. Predictions on test set 2

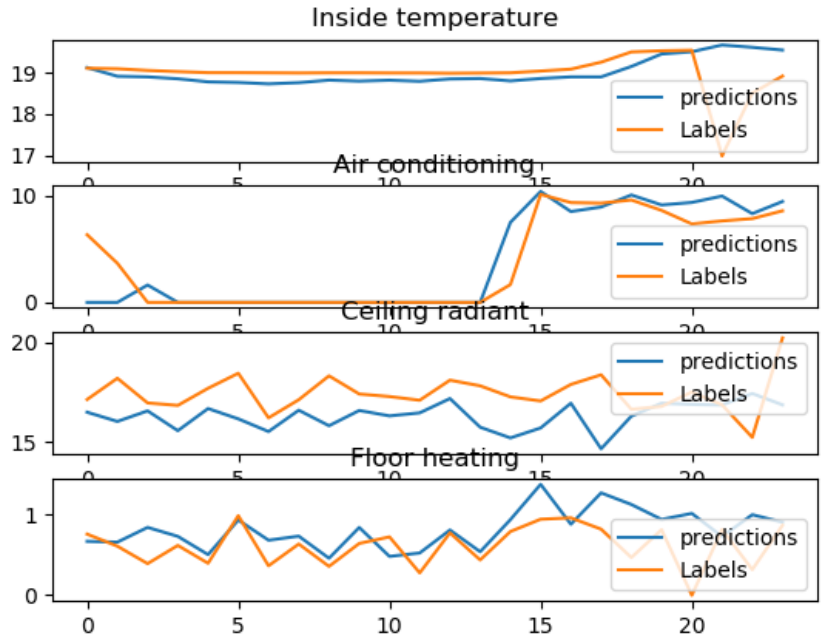

Figure 23. Predictions on test set 3

The model not only fitted the training data, but also learned to generalize to forecast accurately enough based on new data from the same distribution. However, the quality of the forecasts dropped dramatically if the training and the test data were from different distributions. The training data belonged to December and January,

and the results given in Figures 21, 22 and 23 were obtained for the beginning of February. The following results were received for March and April respectively:

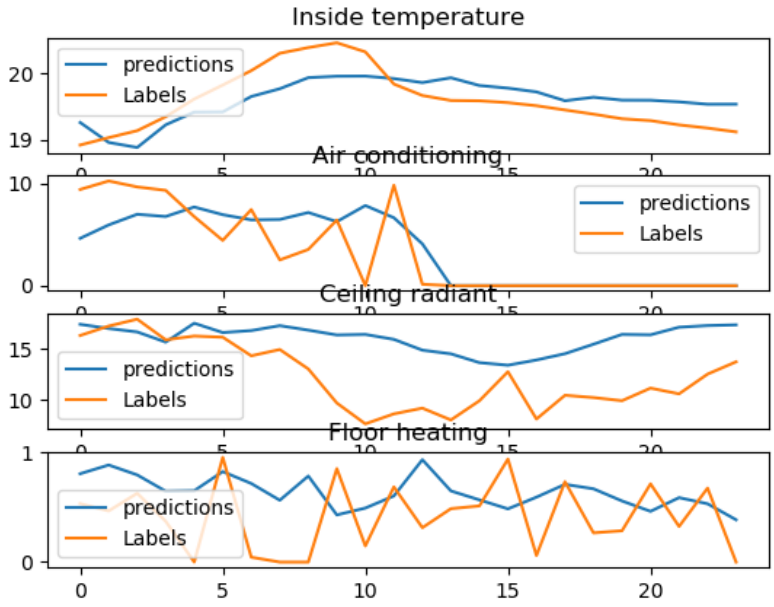

Figure 24. Forecasts for data, obtained in March

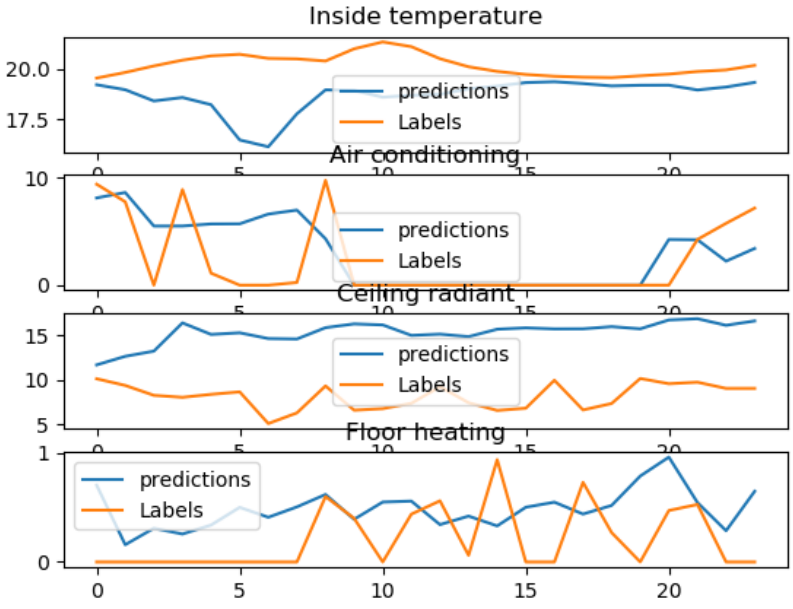

Figure 25. Forecasts for data, obtained in April

4 POSSIBLE IMPROVEMENTS

All the current limitations of the model are directly connected to the quality and amount of data available for analysis, that is why the most important improvements are only available for data collection stage:

- Initially the data obtained from building automation has very limited variance, which is a major obstacle in training machine learning models. A possible solution is changing the control system settings periodically, e.g. changing the set point for indoor temperature by a fraction of a degree.
- Having more data in general. For this thesis project only two months of data was used, so forecast accuracy is limited, what restricts possible applications. If several years of data were available, it would be possible to build a forecast for any month or season in general.

5 POSSIBLE APPLICATIONS

The model, obtained as a result of this thesis project, can have different applications:

- Smart greed applications. The simulation can be used to specify a required amount of energy ahead of time, making it possible to plan the total energy consumption ahead of time.
- Maintenance. The model considers normal functions of a building, so if a malfunction appears, it will be represented as an extreme difference between forecasted and actual data.

6 CONCLUSION

In this thesis project, a machine learning based model was built to simulate the heating energy consumption of a building for the following day based on the previous 24 hours. In order to achieve this a feed forward NARX-like neural network was designed and trained on pre-processed data obtained in advance. The obtained model was able to predict relatively accurately (average error less than 10%) the next 24 hours of heating energy consumption, considering that data used for forecasting came from similar distribution as the training data.

7 REFERENCES

A little book of R for time series (n. d.) Time series analysis. Retrieved 25 April 2018 from https://a-little-book-of-r-for-time-series.readthedocs.io/en/latest/src/timeseries.html#time-series-analysis

B. Dantzig, G., E. Gill, P., Murray, W., A. Saunders, M., A. Tomlin, J., & H. Wright, M. (2007). *Systems Optimization (pp. 152-153).*

Duda, R. O., Hart, P. E., & Stork, D. G. (2000). Unsupervised Learning and Clustering. In *Pattern Classification* (pp. 517-528). Wiley.

Fregly, C. (2016). Gradient descent, back propagation and auto differentiation advanced spark and tensorflow meetup. Retrieved 28 April 2018 from https://www.slideshare.net/cfregly/gradient-descent-back-propagation-and-auto-differentiation-advanced-spark-and-tensorflow-meetup-08042016

Haykin, S. (1999). *Neural Networks: A Comprehensive Foundation (pp. 10-11).* Prentice Hall.

Hyndman, R. J., & Athanasopoulos, G. (n. d.). *Stationarity and differencing.* Retrieved 4 May 2018 from https://www.otexts.org/fpp/8/1

Hyndman, R. J., & Athanasopoulos, G. (n. d.). Time series components. Retrieved 4 May 2018 from https://www.otexts.org/fpp/6/1

Kongcharoen, C., & Kruangpradit, T. (2013). *Autoregressive Integrated Moving Average with Explanatory Variable (ARIMAX) Model for Thailand Export (p. 3).* Bangkok: Thammasat University.

LeCun, Y., Bengio, Y., & Hinton, G. (2015). Deep Learning (pp. 436-444). *Nature*

LeCun, Y., Bottou, L., B. Orr, G., & Müller, K.-R. (2002). *Efficient BackProp (pp. 10-12).*

Li, X., & Wu, X. (2014). *Constructing Long Short-Term Memory based Deep Recurrent Neural Networks for Large Vocabulary Speech Recognition (p. 1).*

Loss Function. (n.d.). Retrieved 2 May 2018 from: https://www.statlect.com/glossary/loss-function

Mills, T. C. (1991). *Time Series Techniques for Economists (pp. 61-63).*

Olah, C. (n.d.). *Understanding LSTM Networks.* Retrieved from Colah's blog: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach, Third Edition (pp. 695 - 696).*

S. McCulloch, W., & Pitts, W. (1943). A Logical Calculus of Ideas Immanent in Nervous Activity (pp. 115-133).

Samuel, A. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development* (pp. 210-229).

Appendix 1

Contents of the file ‘filter-data.py’

```
from pandas import read_csv
import numpy

"""
This function loads data exported from database and removes all "null"
values and outliers. Then result is saved in a separate file
"""
dataset = read_csv('grafana_data_export.1.csv', index_col=0, sep=';')
dataset.replace(to_replace='null', value=numpy.nan, inplace=True)
dataset.dropna(axis=0, how='any', inplace=True)

df = dataset[['Ceiling radiant', 'Air conditioning', 'Floor heating']]
dataset['Energy consumption'] = df.sum(axis=1)

dataset = dataset[dataset['Energy consumption'] > 0]
dataset = dataset[dataset['Energy consumption'] < 100]
dataset = dataset.drop('Temperature set point in room 101', axis=1)


dataset.to_csv('clean_data.csv')
```

Appendix 2

Contents of the file ‘to_supervised.py’

```
from pandas import read_csv, DataFrame
from timeseries_to_supervised import series_to_supervised


"""
This file transforms data from time series to supervised form by calling
series_to_supervised function
"""
dataset = read_csv('clean_data.csv', header=0, index_col=0, sep=',')

temp_out = series_to_supervised(dataset['Outside temp'].values.tolist(),
24, 1)
temp_in = series_to_supervised(dataset['Temperature in room
101'].values.tolist(), 24, 1)
ceiling_radiant = series_to_supervised(dataset['Ceiling
radiant'].values.tolist(), 24, 1)
air_conditioning = series_to_supervised(dataset['Air
conditioning'].values.tolist(), 24, 1)
floor_heating = series_to_supervised(dataset['Floor
heating'].values.tolist(), 24, 1)


temp_out.to_csv('temp_out.csv')
```

```
temp_in.to_csv('temp_in.csv')
ceiling_radiant.to_csv('ceiling_radiant.csv')
air_conditioning.to_csv('air_conditioning.csv')
floor_heating.to_csv('floor_heating.csv')
```

Appendix 3

Contents of the file ‘train_model.py’

```
from math import sqrt
import numpy as np
from matplotlib import pyplot
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from sklearn.metrics import mean_squared_error
from keras.models import Model
from keras.layers import Input, Dense, BatchNormalization
from keras.layers import Dropout, concatenate
from keras.optimizers import Adam

#load preprocessed data
temp_in_df = read_csv('temp_in.csv', index_col=0)
temp_out_df = read_csv('temp_out.csv', index_col=0)
air_cond_df = read_csv('air_conditioning.csv', index_col=0)
ceiling_radiant_df = read_csv('ceiling_radiant.csv', index_col=0)
floor_heating_df = read_csv('floor_heating.csv', index_col=0)

#transform pandas dataframe to numpy array and set precision to float 32
temp_in = temp_in_df.astype('float32').values
temp_out = temp_out_df.astype('float32').values
air_cond = air_cond_df.astype('float32').values
ceiling_radiant = ceiling_radiant_df.astype('float32').values
floor_heating = floor_heating_df.astype('float32').values

#divide all the data into test and train sets
train_size = int(temp_in.shape[0] * 0.85)
train_X_temp_in, train_Y_temp_in = temp_in[:train_size, 0:24],
temp_in[:train_size, -1]
train_X_temp_out, train_Y_temp_out = temp_out[:train_size, 0:24],
temp_out[:train_size, -1]
train_X_air_cond, train_Y_air_cond = air_cond[:train_size, 0:24],
air_cond[:train_size, -1]
train_X_ceiling_radiant, train_Y_ceiling_radiant =
ceiling_radiant[:train_size, 0:24], ceiling_radiant[:train_size, -1]
train_X_floor_heating, train_Y_floor_heating = floor_heating[:train_size,
0:24], floor_heating[:train_size, -1]
```

```
test_X_temp_in, test_Y_temp_in = temp_in[train_size:, 0:24],
temp_in[train_size:, -1]
test_X_temp_out, test_Y_temp_out  = temp_out[train_size:, 0:24],
temp_out[train_size:, -1]
test_X_air_cond, test_Y_air_cond = air_cond[train_size:, 0:24],
air_cond[train_size:, -1]
test_X_ceiling_radiant, test_Y_ceiling_radiant  =
ceiling_radiant[train_size:, 0:24], ceiling_radiant[train_size:, -1]
test_X_floor_heating, test_Y_floor_heating = floor_heating[train_size:,
0:24], floor_heating[train_size:, -1]


#model definition
input_out_temp = Input(shape=(24,))
input_in_temp = Input(shape=(24,))
input_air_cond = Input(shape=(24,))
input_ceiling_radiant = Input(shape=(24,))
input_floor_heating = Input(shape=(24,))

dense_out_temp_1 = Dense(200, activation='tanh')(input_out_temp)
dense_in_temp_1 = Dense(200, activation='tanh')(input_in_temp)
dense_air_cond_1 = Dense(400, activation='relu')(input_air_cond)
dense_air_cond_2 = Dense(200, activation='tanh')(dense_air_cond_1)
dense_ceiling_radiant_1 = Dense(400,
activation='relu')(input_ceiling_radiant)
dense_ceiling_radiant_2 = Dense(200,
activation='tanh')(dense_ceiling_radiant_1)
dense_floor_heating_1 = Dense(400,
activation='relu')(input_floor_heating)
dense_floor_heating_2 = Dense(200,
activation='tanh')(dense_floor_heating_1)

merge_layer = concatenate([dense_out_temp_1, dense_in_temp_1,
dense_air_cond_2, dense_ceiling_radiant_2, dense_floor_heating_2 ],
axis=-1)
dense_merged_1 = Dense(1500, activation = 'tanh')(merge_layer)
norm1 = BatchNormalization()(dense_merged_1)
drop1 = Dropout(0.2)(norm1)
dense_merged_2 = Dense(2000, activation='tanh')(drop1)

dense_in_temp_3 = Dense(200, activation='tanh')(dense_merged_2)
dense_air_cond_3 = Dense(200, activation='tanh')(dense_merged_2)
dense_ceiling_radiant_3 = Dense(200, activation='tanh')(dense_merged_2)
dense_floor_heating_3 = Dense(200, activation='tanh')(dense_merged_2)
drop_in_temp = Dropout(0.2)(dense_in_temp_3)
drop_air_cond = Dropout(0.2)(dense_air_cond_3)
drop_ceiling_radiant = Dropout(0.2)(dense_ceiling_radiant_3)
drop_floor_heating = Dropout(0.2)(dense_floor_heating_3)

dense_in_temp_4 = Dense(150, activation='tanh')(drop_in_temp)
```

```
dense_air_cond_4 = Dense(150, activation='tanh')(drop_air_cond)
dense_ceiling_radiant_4 = Dense(150,
activation='tanh')(drop_ceiling_radiant)
dense_floor_heating_4 = Dense(150, activation='tanh')(drop_floor_heating)

out_in_temp = Dense(1, activation='linear')(dense_in_temp_4)
out_air_cond = Dense(1, activation='relu')(dense_air_cond_4)
out_ceiling_radiant = Dense(1,
activation='relu')(dense_ceiling_radiant_4)
out_floor_heating = Dense(1, activation='relu')(dense_floor_heating_4)

model = Model(inputs=[input_out_temp, input_in_temp, input_air_cond,
input_ceiling_radiant, input_floor_heating], outputs=[out_in_temp,
out_air_cond, out_ceiling_radiant, out_floor_heating])
#configuration of Optimizer
opt = Adam(lr=0.0001, beta_1=0.9, beta_2=0.999, epsilon=1.e-9, decay=0.0,
amsgrad=False)
model.compile(loss='mean_squared_error', optimizer=opt)

#actual training
history = model.fit([train_X_temp_out, train_X_temp_in, train_X_air_cond,
train_X_ceiling_radiant, train_X_floor_heating], [train_Y_temp_in,
train_Y_air_cond, train_Y_ceiling_radiant, train_Y_floor_heating],
epochs=500, batch_size=24, validation_data=
([test_X_temp_out, test_X_temp_in, test_X_air_cond,
test_X_ceiling_radiant, test_X_floor_heating],
 [test_Y_temp_in, test_Y_air_cond, test_Y_ceiling_radiant,
test_Y_floor_heating]),
 verbose=2, shuffle=True)

pyplot.plot(history.history['loss'], label='train')
pyplot.plot(history.history['val_loss'], label='test')


pyplot.legend()
pyplot.show()

#saving model for later use
model_json = model.to_json()
with open("model2.json", 'w') as json_file:
    json_file.write(model_json)

model.save_weights('model2.h5')
print("Model saved")
```

Appendix 3

Contents of the file ‘test_model.py’

```
from keras.models import model_from_json
from pandas import read_csv
import numpy as np
import os
from matplotlib import pyplot as plt
from sklearn.preprocessing import MinMaxScaler

#loads pretrained in run.py file model
def load_model():
    json_file = open('model2.json', 'r')
    loaded_model_json = json_file.read()
    json_file.close()
    loaded_model = model_from_json(loaded_model_json)
    loaded_model.load_weights('model2.h5')
    return loaded_model

#loads preprocessed data
def load_data():
    temp_in_df = read_csv('temp_in.csv', index_col=0)
    temp_out_df = read_csv('temp_out.csv', index_col=0)
    air_cond_df = read_csv('air_conditioning.csv', index_col=0)
    ceiling_radiant_df = read_csv('ceiling_radiant.csv', index_col=0)
    floor_heating_df = read_csv('floor_heating.csv', index_col=0)

    temp_in = temp_in_df.astype('float32').values
    temp_out = temp_out_df.astype('float32').values
    air_cond = air_cond_df.astype('float32').values
    ceiling_radiant = ceiling_radiant_df.astype('float32').values
    floor_heating = floor_heating_df.astype('float32').values

    train_size = int(temp_in.shape[0] * 0.85)
    #separated data into training and learning sets as well as into X and
Y labels
    train_X_temp_in, train_Y_temp_in = temp_in[:train_size, 0:24],
temp_in[:train_size, -1]
    train_X_temp_out, train_Y_temp_out  = temp_out[:train_size, 0:24],
temp_out[:train_size, -1]
    train_X_air_cond, train_Y_air_cond = air_cond[:train_size, 0:24],
air_cond[:train_size, -1]
    train_X_ceiling_radiant, train_Y_ceiling_radiant =
ceiling_radiant[:train_size, 0:24], ceiling_radiant[:train_size, -1]
    train_X_floor_heating, train_Y_floor_heating =
floor_heating[:train_size, 0:24], floor_heating[:train_size, -1]


    test_X_temp_in, test_Y_temp_in = temp_in[train_size:, 0:24],
temp_in[train_size:, -1]
```

```
    test_X_temp_out, test_Y_temp_out  = temp_out[train_size:, 0:24],
temp_out[train_size:, -1]
    test_X_air_cond, test_Y_air_cond = air_cond[train_size:, 0:24],
air_cond[train_size:, -1]
    test_X_ceiling_radiant, test_Y_ceiling_radiant  =
ceiling_radiant[train_size:, 0:24], ceiling_radiant[train_size:, -1]
    test_X_floor_heating, test_Y_floor_heating =
floor_heating[train_size:, 0:24], floor_heating[train_size:, -1]
    return test_X_temp_out ,test_X_temp_in, test_X_air_cond,
test_X_ceiling_radiant, test_X_floor_heating, test_Y_temp_out,
test_Y_temp_in, test_Y_air_cond, test_Y_ceiling_radiant,
test_Y_floor_heating

#this function is a wrapper to predict whole next day instead of only 1
hour (as it would happen if the usual function provided by keras was
called)
def predict_data(model, test_X_temp_out, test_X_temp_in, test_X_air_cond,
test_X_ceiling_radiant, test_X_floor_heating, test_Y_temp_out):
    result_temp_in = []
    result_air_cond = []
    result_ceiling_radiant = []
    result_floor_heating = []

    temp, air_cond, ceiling_radiant, floor_heating =
model.predict([test_X_temp_out, test_X_temp_in, test_X_air_cond,
test_X_ceiling_radiant, test_X_floor_heating])
    result_temp_in.append(np.squeeze(temp).tolist())
    result_air_cond.append(np.squeeze(air_cond).tolist())
    result_ceiling_radiant.append(np.squeeze(ceiling_radiant).tolist())
    result_floor_heating.append(np.squeeze(floor_heating).tolist())
    test_X_temp_in = np.append(test_X_temp_in, temp)
    test_X_temp_in = np.delete(test_X_temp_in, 0, axis=0)
    test_X_temp_in = test_X_temp_in.reshape(1, 24)

    test_X_air_cond = np.append(test_X_air_cond, air_cond)
    test_X_air_cond = np.delete(test_X_air_cond, 0, axis=0)
    test_X_air_cond = test_X_air_cond.reshape(1, 24)

    test_X_ceiling_radiant = np.append(test_X_ceiling_radiant,
ceiling_radiant)
    test_X_ceiling_radiant = np.delete(test_X_ceiling_radiant, 0, axis=0)
    test_X_ceiling_radiant = test_X_ceiling_radiant.reshape(1, 24)

    test_X_floor_heating = np.append(test_X_floor_heating, floor_heating)
    test_X_floor_heating = np.delete(test_X_floor_heating, 0, axis=0)
    test_X_floor_heating = test_X_floor_heating.reshape(1, 24)

    test_X_temp_out = np.append(test_X_temp_out, test_Y_temp_out[0])
    test_X_temp_out = np.delete(test_X_temp_out, 0, axis=0)
    test_X_temp_out = test_X_temp_out.reshape(1, 24)
```

```
    for i in range(23):
        temp, air_cond, ceiling_radiant, floor_heating =
model.predict([test_X_temp_out, test_X_temp_in, test_X_air_cond,
test_X_ceiling_radiant, test_X_floor_heating])
        result_temp_in.append(np.squeeze(temp).tolist())
        result_air_cond.append(np.squeeze(air_cond).tolist())

result_ceiling_radiant.append(np.squeeze(ceiling_radiant).tolist())
        result_floor_heating.append(np.squeeze(floor_heating).tolist())
        test_X_temp_in = np.append(test_X_temp_in, temp)
        test_X_temp_in = np.delete(test_X_temp_in, 0, axis=0)
        test_X_temp_in = test_X_temp_in.reshape(1, 24)

        test_X_air_cond = np.append(test_X_air_cond, air_cond)
        test_X_air_cond = np.delete(test_X_air_cond, 0, axis=0)
        test_X_air_cond = test_X_air_cond.reshape(1, 24)

        test_X_ceiling_radiant = np.append(test_X_ceiling_radiant,
ceiling_radiant)
        test_X_ceiling_radiant = np.delete(test_X_ceiling_radiant, 0,
axis=0)
        test_X_ceiling_radiant = test_X_ceiling_radiant.reshape(1, 24)

        test_X_floor_heating = np.append(test_X_floor_heating,
floor_heating)
        test_X_floor_heating = np.delete(test_X_floor_heating, 0, axis=0)
        test_X_floor_heating = test_X_floor_heating.reshape(1, 24)

        test_X_temp_out = np.append(test_X_temp_out, test_Y_temp_out[0])
        test_X_temp_out = np.delete(test_X_temp_out, 0, axis=0)
        test_X_temp_out = test_X_temp_out.reshape(1, 24)

    return result_temp_in, result_air_cond, result_ceiling_radiant,
result_floor_heating


test_X_temp_out, test_X_temp_in, test_X_air_cond, test_X_ceiling_radiant,
test_X_floor_heating, test_Y_temp_out, test_Y_temp_in, test_Y_air_cond,
test_Y_ceiling_radiant, test_Y_floor_heating = load_data()
model = load_model()
temp_in, air_cond, ceiling_radiant, floor_heating = predict_data(model,
test_X_temp_out[2:3],test_X_temp_in[2:3], test_X_air_cond[2:3],
test_X_ceiling_radiant[2:3], test_X_floor_heating[2:3],
test_Y_temp_out[2:])

plt.subplot(4, 1, 1)
plt.plot(temp_in, label="predictions")
plt.plot(test_Y_temp_in[2:26], label="Labels")
```

```
plt.gca().set_title('Inside temperature')
plt.legend()
plt.subplot(4, 1, 2)
plt.plot(air_cond, label="predictions")
plt.plot(test_Y_air_cond[2:26], label="Labels")
plt.gca().set_title('Air conditioning')
plt.legend()
plt.subplot(4, 1, 3)
plt.plot(ceiling_radiant, label="predictions")
plt.plot(test_Y_ceiling_radiant[2:26], label="Labels")
plt.gca().set_title('Ceiling radiant')
plt.legend()
plt.subplot(4, 1, 4)
plt.plot(floor_heating, label="predictions")
plt.plot(test_Y_floor_heating[2:26], label="Labels")
plt.gca().set_title('Floor heating')
plt.legend()
plt.savefig('test5.png')
plt.show()
```