University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science, Security and Reliability	Spring semester, 2021 Open

Author:
Seblewongel Minassie Wondimagegnehu

Programme Coordinator: Professor Reggie Davidrajuh
Supervisor: PhD Student Rituka Jaiswal

Title of master's thesis:

Fog Computing for Efficient Power Consumption Predictions in Smart Grids

Credits: 30

Keywords: cloud computing, fog computing, smart city, smart grids, deep learning, machine learning	Number of pages: 42 + Appendix **Stavanger, 14,06,2021**

University of Stavanger

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Fog Computing for efficient Power Consumption Predictions in Smart Grids

Master's Thesis in Computer Science
By

Seblewongel Minassie Wondimagegnehu

Programme Coordinator
Professor Reggie Davidrajuh

Supervisor
PhD Student Rituka Jaiswal

June 14, 2021

Abstract

Cloud Computing provides on-demand computing services like software, networking, storage, analytics, and so on over the Internet. However, the explosion of the Internet of Things (IoT) devices and the high volume and variety of data generated by these devices, is creating challenges in handling large and rapid computations. Such challenges require a reliable and secure system with low latency. Fog Computing is a promising solution to address these challenges. The approach allocates tasks down to devices that are closer to the sensors which will provide immediate feedback to systems of time sensitive IoT applications. Latency, location awareness and highly virtualized computational models are some of the advantages that Fog Computing has over Cloud Computing. In this thesis work, smart grid is used as a use case. Power consumption prediction was used as a task to perform model training and prediction. In addition, comparisons of the mathematical models were performed to find out which of the models perform better in providing sound predictions and which models are better in resource usage. Finally, the computations were performed on both Fog and Cloud devices to compare CPU time and memory usages of the models in each device.

Keywords: cloud computing, fog computing, smart city, smart grids, deep learning, machine learning

Acknowledgements

First, and foremost, I would like to sincerely thank my Professor Reggie Davidrajuh for bringing Fog computing to my attention with this thesis for I really value the knowledge I got out of it. Special thanks to PhD. student Rituka Jaiswal for her endless knowledgeable support, guidance, and valuable advice all the way to the end of this thesis work. I would also extend my thanks to my mentor, my brother, Hailu Minassie for being my role model, for his endless encouragement throughout my study. I am also sincerely grateful to my brother Beza Minassie for being there when I needed him. Last, but definitely not least, I would like to extend special thanks to the love of my life, the father of my children, Zelalem Teshome for being supportive and helpful throughout my study and taking care of our children throughout my studies.

Table of Contents

List of Tables

List of Figures

Abbreviations

IT	Information Technology
DDoS	Distributed Denial of Service
IoT	Internet of Things
HAN	House Area Network
NAN	Neighborhood Area Network
WAN	Wide Area Network
ICT	Internet Communication Technologies
CPU	Central Processing Unit
AI	Artificial Intelligent
ML	Machine Learning
SVR	Support Vector Regressions
SV	Support Vector
ANN	Artificial Neural Network
RNN	Recursive Neural Network
LSTM	Long short-term memory
CEC	Constant Error Carousels
UCI	University of California, Irvine
MSE	Mean Square Error
RMSE	Root Mean Square Error
RAM	Random Access Memory

1. Introduction

1.1 The Cloud

Cloud computing is revolutionizing the technology world with the provision of easy access and security to the huge set of data that's being produced by the ever-growing use of computing devices. Basically, cloud computing refers to an always available, on-demand computing service that is based off of data centers located in remote locations. The services that it offers include data storage, data computing power or resources, web or applications hosting, and so forth. As mentioned, a user will access all these and more such services from a remotely managed set of networked computer systems, over the internet, that are located in a data center, and hence the name 'cloud' [1][2]. Figure 1.1 depicts the general idea of the cloud technology.

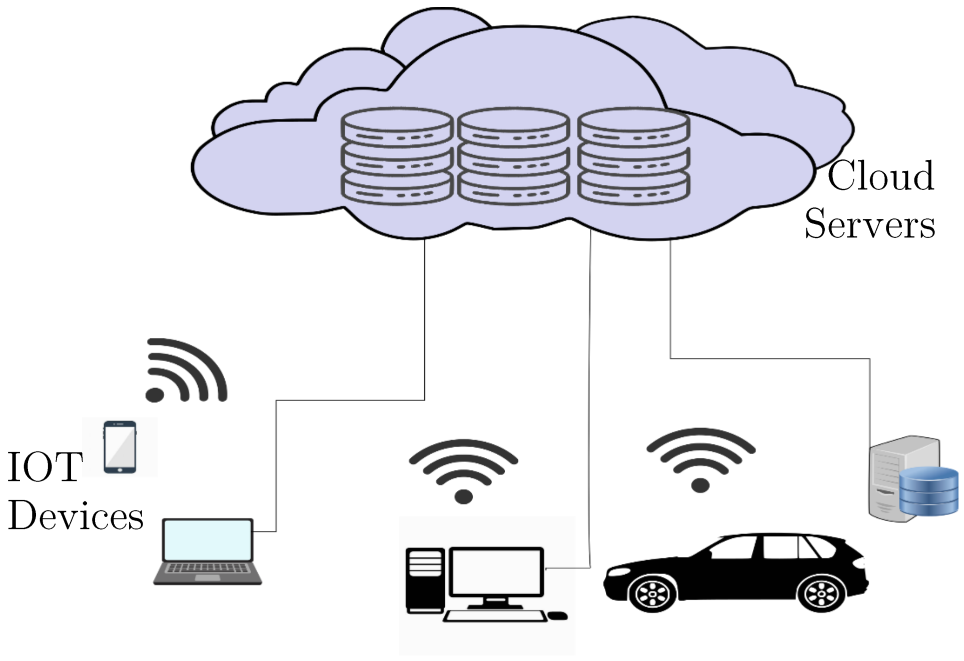


Figure 1.1 Basic depiction of Cloud Technology.

The advances made in the technology coupled with abundant availability of mobile or hand-held smart devices and increased access to better internet services had played an enormous role in making cloud computing a very important part of information technology (IT) [1]. Practically, cloud computing got prominence because it removes the need of many structural requirements for a typical setup. This is apparent especially when compared to a conventional set up. In the conventional setup a user, say a client, needs to make investment for setting up and maintaining the infrastructure and for the human resources that need to carry out these tasks. In contrast, the on-demand service approach of cloud computing provides a type of pay-as-you-go solution allowing clients to effectively cater the services as per their business' need. Moreover, durability, elasticity and accessibility are the most important and appealing features that cloud computing offers. Table 1.1 summarizes the characteristics of cloud computing.

Table 1.1: Summary of Characteristics of Cloud Computing.[1]

Features	**Description**
On-demand	Resources are accessed whenever needed.
Broad access points	Computing capabilities and resources are accessible through a variety set of platforms such as hand-held smart devices, workstation, laptop and so on.
Resource pooling	Resources such as computing, storage and others are pooled and get allocated based on consumers' needs. Customers has little to no say in requiring location of some resources as they can be reallocated based on demand.
Elasticity	Resources' capabilities can be almost unlimitedly increased or decreased depending on business demand.
Metered services	Use of resources like account management, bulk storage and processing power is easily controlled and metered for better optimization.
Multitenancy	Simultaneous sharing of same resources, like serverless computing, by multiple users (or tenants) allows for better use of resources and lower coast.
Fault tolerance	Creating duplicates of private cloud servers across multiple data centers allow for high availability of data and resources that can overcome physical and technical failures.
Security	Security can be improved by applying security at multiple levels, such as at the server level and at subnet level.

To summarize, benefits of cloud computing include lower coasts of operation and services, rapid scalability and elasticity, ability to pool resources, increased data safety and easy access to a wide range of computational resources or applications. On the other hand, the need for continuous fast internet, possibility of latency, security susceptibility of processes taking place or data stored, data loss or service interruption due to physical damage to data centers are some of the limitations and challenges faced by cloud computing technology. Moreover, account or service traffic hijacking, distributed denial of service (DDoS) attacks and malicious insiders are some of the threats to cloud computing as identified by the Cloud Security Alliance [3][4].

1.2 Smart cities

The progression towards smart cities has met a partner in the cloud technologies. Smart cities come about with the use of data collected from a vast number of sensors from that urban area. The data is used both to efficiently manage resources in real time and to forecast future challenges. The data originates from the dwelling individuals and the physical devices they use that are interconnected with each other. The interconnection and interaction of devices, known as Internet of things (IoT), creates a large network that advances to a better utilization and management of resources. Beyond the commonly known day-to-day household electronics we use, it's important to note that the "things" in the IoT does no longer refer to the typical personal computing systems. Rather it may refer to things as simple as a building door or a blood sugar monitor or a geo-locator on a wild animal, or as complex as a self-driving vehicle or as big as a building, and so on[5].

It is projected that in just the coming few decades population growth will significantly increase and, of these billions of people, as many as two-third are projected to reside in cities [6-8]. Moreover, in about the same time, it is estimated that about 75 billion devices will be in operation for day-to-day use [9]. This will in turn intensify the need for services and resources, that are already in short supply, and infrastructure which will be crucial for such sustainably manage resources. So, to efficiently accommodate urban communities, cities will have to employ intelligent technologies and provide efficient services in all aspects of life. This will mean that services such as healthcare, energy, transportation, construction, education and so on should become smart in their application [7].

However, it is sure that such rapid development and expansion of future cities creates some challenges:[7]

- Smart city goals are realized by constructing large scale infrastructures of city components such as roads, subways, power grid, pipelines, bridges and so on. So, it is as important building significant amount of sensing networks that are geospatially distributed which in real-time and accurately monitor the health of the infrastructure elements.
- These networks of geospatially distributed sensors generate enormous magnitude of information or "big data" that will require a fast and real-time analysis.[10]
- Besides the massive amount of data they generate, the Internet of Things mashed by the machine-to-machine communication between these large number of sensors will dominate the communication system traffic [11][12].

- For the systems in smart cities to efficiently work, all the elements of infrastructures demand intelligent monitoring mechanisms with rapid feed-back systems. In other words, the need for intelligent and efficient decision-making systems with integrated optimal feed-back loop is paramount.

Hence, smart cities need to deploy high performance computing systems that can interact with the huge number of sensors and handle the massive volume of data with capabilities of live system assessment and course correction. Moreover, the proximity of this location-ware system to the data sources determines the latency that is inherent to such geospatially distributed internet of things [7]. Table 1.2 summarizes the basic design considerations that are needed for industrial Internet of Things applications as outlined by [12].

Table 1.2: General Considerations for Designing Large Scale IoT Applications.

Goals	**Description**
Energy	How will the IoT device be powered and how efficiently will the system use power?
Latency	What is the optimal message propagation and processing mode and how fast will it be?
Throughput	What maximum amount of data can be transported through the network?
Scalability	How many devices are going to be supported?
Topology	What are the communicating agents in the system?
Safety and security	How safe and secure will the system be and what are the vulnerabilities?

Because of its distributed and scalable data management structures, cloud computing is used to address the challenges faced by smart city systems. However, the sheer volume of data needed to be handled and the necessity to have geolocation-aware smart system with fast response time calls for different paradigm all together. The new paradigm that addresses these challenges builds on the features of cloud computing where at the same time providing real time need based solutions. A system with such capabilities and ripe for a number of use cases is Fog Computing.

Table 1.3 Some examples of smart city applications and roles of Fog and Cloud Computing have. Adopted from [18].

Smart City Applications	**Sub-applications**	**Fog Roles**	**Cloud Roles**
Intelligent transportation	• Route planning and congestion avoidance • Intelligent traffic light control • Intelligent parking services • Accident avoidance • Self-driving vehicles	Fogs in the form of Road Side Units (RSUs) or other computerized units provide low-cost relays among vehicles, roads and parks sensors, traffic lights, and the cloud. They provide fast response and control services.	Clouds collects, filters, and stores traffic information. It helps in coordinating city traffic and parking optimizations. It also helps in planning for enhancing traffic systems.
Smart energy	• Smart grid • Smart buildings • Renewable energy plants • Smart meters • Wind farms • Hydropower plants	Fogs provide local control for energy systems, distribution units, and consumer locations. They also enable smooth integration of different energy systems.	Clouds collects, filters, and stores energy information. It supports decision making for utilizing smart grids and renewable energy features based on collected and analyzed data for consumers needs and renewable energy productions.
Smart water	• Leakage detections • Water leakage reduction • Water quality monitoring • Smart water meter • Smart irrigation	Fogs provide better and faster local monitoring and controls for smart water networks. They also offer real-time monitoring for faults and leakage and support repair and maintenance operations.	Smart water networks information is collected, stored, and utilized by cloud services to enhance the water networks, production, and quality and to reduce water losses.
City structure health monitoring	• Health monitoring for • Bridges • Large public buildings • Tunnels • Train and subway rails • Oil and gas pipelines	Fogs helps to reduce data traffic between the sensors monitoring the structures and their main control stations. In addition, they provide fast safety controls for some applications.	Cloud collects, filters, and stores structure health information. The cloud can help analyze collected data to enhance the maintenance processes and improve the health of the city structures.
Environmental monitoring	• Air quality monitoring • Noise monitoring • River monitoring • Coastal monitoring	Fogs helps enhance environmental monitoring process by providing smart environmental monitoring closer to the monitored area.	Cloud provides processes to collectively analyze city environmental and health status.
Public safety and security	• Crowed control (for sports games, parades, and so on) • Crime alerts • Emergency response service (floods, earthquakes, etc.)	Fogs help reduce the communication traffic between these places and the main security monitoring stations.	Cloud provides a powerful platform for analyzing the collected data about the current situation to help in providing possible actions for better control and emergency relief.

1.3 Fog Computing

As Cloud Computing, Fog Computing is built with storage, compute, and networking resources as integral components. The following characteristics define a few of these Cloud Computing extension services [13]:

- The location awareness property is essential especially to those applications that require low latency.
- Decentralized geospatial distribution that help provide services for non-stationary components.
- Handle data from large number of scale sensors that allows to monitor components of smart city infrastructures. Data analysis from these arrays of sensors performed close to the sources.
- Ability to communicate with mobile systems permitting dissociation of spatial information from the host.
- The ability to perform real time interactions as opposed to batched interactions.
- Allow for deployment in a variety of heterogeneous environments where components across systems interoperate.

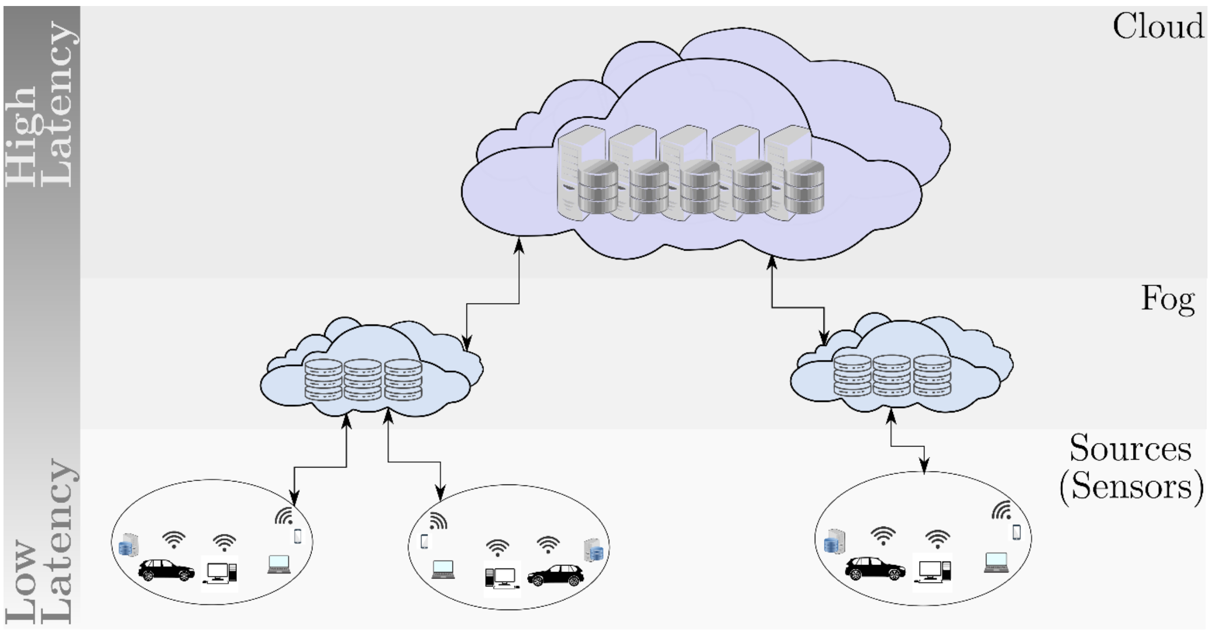


Figure 1.2 Depiction of basic Fog Computing architecture.

With these inherent properties, Fog architecture distributes computing tasks through the network of devices for enhanced computing capability. On top of helping improve efficiency and performance, Fog Computing alleviates the need to transfer all the data from sensors in the network to the cloud by temporarily storing and performing necessary analysis at the edge locations (or edge devices) [14]. Edge location, or edge device refers to a position in the IoT that is closest to the source of data or to the end user of a system. In other words, the fog computing approach allocates tasks down to devices closer to the sensors which will provide immediate feedback to systems of time sensitive IoT applications. Latency, location awareness and highly virtualized computational models are some of the advantages that Fog Computing has over the Cloud Computing. Table 1.4 lays out some of the features these systems have.

In summary, the expected future population growth, and thereby, the inevitable push towards urbanization, unavoidably will result in the increased need and use of a number of geospatially networking sensors and monitors [6]. This creates a challenge in handling the large volume of data [15,16]. To this end, studies done by [7] show ways of implementing a hierarchical Fog computing architecture to handle such big data analysis. Moreover, the study suggests that the communication among the increased number of sensors themselves will take up notable size of part of the communication traffic. In addition, works done by [11] and [17] also show how fog computing parallelizes the data handling at edge of network. Figure 1.2 represents the basic structure of Fog Computing.

Table 1.4: Basic features of both Cloud Computing and Fog Computing. Adopted from [9] and [14].

Features	**Cloud Computing**	**Fog Computing**
Latency	High	Low
Hardware and processing structure	Computing power and storage are scalable	Computing power and storage are limited
Communication mode	IP networks	WLAN, WiFi, LAN, WAN, cellular networks
Working location	Physical data warehouses	Closer to sensors (outdoor)
Security measures	Defined	Not easy to define
Architecture	Centralized	Distributed
Location and context aware system	No	Yes
Geographic coverage	Global	Local

1.3.1 Smart Grid as a Use Case

With these features as a major input, there are several applicable use cases where Fog computing plays an important role in the realization of smart cities. [7] considered smart pipeline monitoring as a use case. Smart buildings and intelligent traffic systems are explained as relevant smart city applications by [18]. [13] details how Connected Vehicle systems, Wireless Sensor and Actuator Networks and Smart Grid are important use cases. In this thesis we consider smart grid as a use case and how to best optimize power usage predictions using machine learning and deep learning. Table 1.3 outlines areas of smart city applications where both Cloud and Fog computing play roles as summarized by [18]. All these and other important applications where Fog computing has significant edge over Cloud computing, rely on its innate properties to achieve their goals. For example, if we consider smart traffic systems, if an accident happens on a specific road, response to manage and reroute traffic flow should be immediate. This is better achieved if data is collected and analyzed locally rather than if it was going to be handled by applications on the Cloud.

Another important application point, that this paper mainly focuses on, is Smart Grid. Smart Grids aims to efficiently manage power demand and supply, minimize power wastage, detect and correct faults in the power transmission and distribution lines and spot anomalous power usage patterns that may lead to hazardous conditions.[19]

Both the EU and the US define smart grid as incorporation or networks of electricity that can intelligently incorporate actions taken by both generators and consumers in order to intelligently deliver sustainable and secure electricity in a fully automated manner [20,21]. Smart grid totally revolutionizes all the major components of electric power system – namely, the generation, transmission, distribution and consumption of electricity power. These components, in Smart Grid, are executed across different network sizes. Table 1.5 summarizes the different network sizes and their respective characteristics [22,23]. At each network level, sensors for components of Smart Grid, produce a large volume of data that is used to monitor and manage such a system. Therefore, large-scale computation will have a vital role in fine tuning efficiency and avoiding disasters. Consequently, data analysis techniques and algorithms are needed for successful implementation of Smart Grids where fault detection and predictive distribution of power is possible.

Being located at the edge of network, Fog computing nodes will collect data from the power pipeline in Smart Grid where data analysis takes place. In contrast to having to collect the data at the Cloud Servers for analysis, Fog Computing allows operations to be more efficient and predictions relevant. Despite the technological advances and the ability to diversify power sources, nations are still prone to negative consequences of power outages. The report on the 2003 U.S. – Canada power outage shows that the blackout coasted the two countries billions of dollars [24].

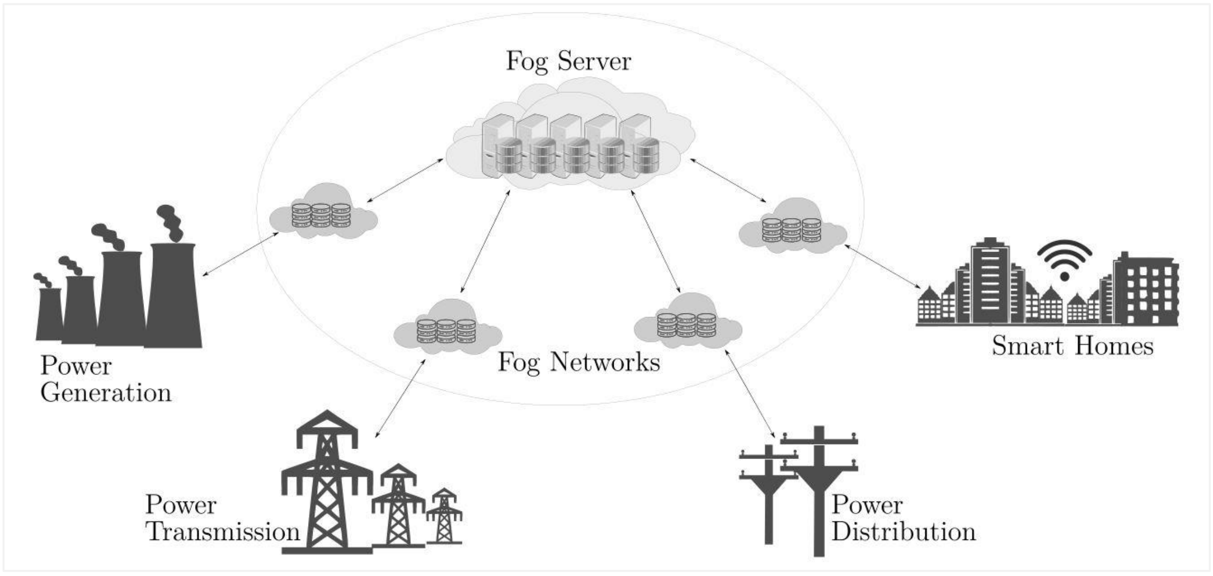


Figure 1.3 Depiction of the generation, transmission, distributions and consumption of power in a Smart Grid system.

Table 1.5: Summary of types and characteristics of network areas in Smart Grids.

Type of network	**Function and Characteristic**
House Area Network (HAN)	Installed nearest to the ground. Consists of smart devices that are used in houses or small offices. Enables smart meters for local energy management.
Neighborhood Area Network (NAN)	Deployed covering larger area. Connects smart metering devices across multiple HANs.
Wide Area Network (WAN)	Installed within a wide area that enables communication of all Smart Grid components. Facilitates data aggregation and proper synchronization among transmission systems.

More recently, cost of an unlikely weather condition that hit the state of Texas in the U.S. and caused a power outage has been estimated to be several billions [25]. This shows that power outages are very costly and need to be managed. Consequently, power consumption prediction and anomaly detection are an integral part of Smart Grids. To this end, Fog Computing plays an essential role because the internet communication technologies (ICT) used by Smart Grids can be better operated on its framework. Moreover, smart houses of today will have a substantially large number of sensors that produce massive amount of data that need to be analyzed in real-time for hazard prevention and intelligent power distribution. This need makes Fog technology and forecasting of power consumption using Fog technologies very important.

2. Problem Definition

As mentioned in the previous section, smart grid systems rely heavily on real-time data analysis and feedback loops that are efficiently run. Moreover, Fog computing is suitable for tackling drawbacks that are faced should the systems be deployed solely on the Cloud. Furthermore, due to the massive volume of data involved in such systems, big data analysis systems and computing algorithms and models are needed to be used.

In this work, mathematical models will be used to predict household electric power consumption. In addition, a comparison will be made as to which models of already established mathematical algorithms perform better in providing sound predictions and which models have better resource usage while reducing latency. The predictions will be performed on both Fog and Cloud devices to compare the memory usage and CPU time of the models on each of them.

3. Approach

3.1 Data Analysis

A large amount of real-time data is generated and collected from smart sensors and meters in Smart Grids. The collected data is essential to evaluate the overall status of the system and to direct everyday actions that will solve problems. This is possible through the use of data analyzing technics which will help in deriving meaning out of the large data sets at hand through a process generally known as Data Analysis.

Data analysis [23] is a process of computationally extracting possible relations among variables by use of technics such as statistics, pattern recognition, machine learning, databases and so on. In various parts of our life, such valuable information that is extracted in this manner is applied in supporting crucial decision makings. However, because data by its characteristic nature is not “clean”, any data set needs to go through preprocessing steps before it gets analyzed.

Data integration, data cleansing and transformation are the main data preprocessing steps that will improve the quality, and therefore, reliability of data. The data integration process mainly helps to aggregate datasets that are collected from multiple sources. This process identifies and normalizes data types and dispels out any redundant value. Data cleansing generally deals with abnormal values in the dataset of which missing values are common. And lastly, data transformation handles preprocessing procedures such as data standardization. Figure 3.1 shows some of the steps needed for preprocessing of data.

[23] describes the possible useful application areas where data analytics in smart grid is especially useful. The main few of these applications are listed as follows:

- Fault detection
- Predictive maintenance
- Transient stability analysis
- Electric devices health monitoring
- Power quality monitoring
- Load profiling, monitoring and forecasting
- Power loss detection

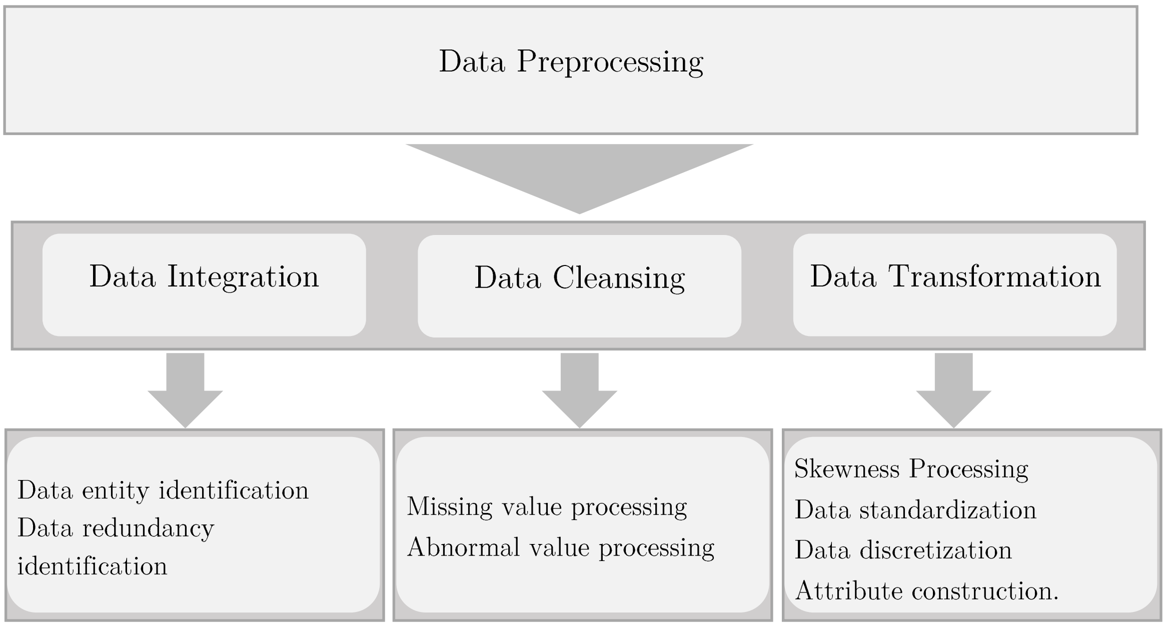


Figure 3.1 Hierarchical flow of data preprocessing techniques.[23]

3.2 Data Analysis Techniques

As a branch of artificial intelligent (AI), Machine Learning (ML) employs mathematical and statistical methods for driving at probabilities and yielding approximations. Using ML technics and models, a system can predict a certain outcome based on inputted data. In other words, by means of ML models, systems that are trained with a historical data will give best fitting results to a particular problem. In general, ML, as the name indicates, is a mechanism used to teach machines how to spot pattern and interpret meaning from data [26].

There are numerous ML algorithms available for use and which particular one gets to be used depends on the type of problem at hand and the property of the available data. Some of the categories of data analytic algorithms are supervised learning, unsupervised learning, semi-supervised learning, correlation, dimensionality reduction. Table 3.1 summarizes basic concepts related to data analysis where Table 3.2 describes some of the algorithms used in data analysis.

As the name implies, the supervised machine learning algorithms need outside assistance. Here, input dataset is divided into two, train and test datasets. The train dataset is used to train the system. Such algorithms learn to identify patterns from the training dataset and apply the same patterns to the test dataset to predict or for classification [26]. On the other hand, unsupervised learning algorithms learn main features that define the data and use these learned features to identify the class of the data [26,27].

Table 3.1: Category and basic description of common data analytic algorithms.

Category	**Algorithm**	**Description**
Supervised Learning	Decision tree	A non-parametric method with a tree-like method whose leaves represent class labels and branches represent conjunctions of features
	Naive Bayes	A probabilistic method based on Bayes theorem with the assumption of independence between every pair of features
	Support vector machine classifier	An algorithm to find a separating hyperplane between the two classes by mapping the labelled data to a high-dimensional feature space
	K Nearest Neighbor	A non-parametric method based on the minimum dissimilarity between new items and the labelled items in different classes
	Random Forest	An algorithm consisting of a collection of simple tree predictors independently for the estimation of the final outcome
Unsupervised Learning	K-means	An unsupervised learning method with a given number of clusters to sort the data based on the average value of data in each group as the centroid
	K-medoids	An unsupervised learning method similar to k-means by assigning the centroid of each group with an existing data point instead of the average value
	Hierarchical Clustering	An alternative approach which aims to build a hierarchy of clusters in a dendrogram without a given number of clusters
	DBSCAN	A density-based clustering algorithm to identify clusters with specific shape in distribution
	Expectation-Maximization	An iterative way to approximate the maximum likelihood estimates for model parameters
Correlation	FP-Growth Algorithm	An efficient method for mining the complete set of frequent patterns with a special data structure named frequent-pattern tree with all the association information reserved
	Apriori Algorithm	A classical data analytics algorithm to discover the potential association rules among frequent items
Dimensionality reduction	Principal Component Analysis	An orthogonal transformation of data with a new coordinate system with the greatest variance projected to the first coordinate

Category	Algorithm	Description
	Self-organizing Map	A type of artificial neural network for a low-dimensional representation of the training data space
	Random Matrix	An algorithm which reveals potential regulations with high order matrices for large data by eigenvalue analysis

Table 3.2 Concepts that are commonly used in data analysis. [23]

Terms	Description
Statistics	The study of collecting, analyzing and interpreting data with mathematics to discover possible associations based on a particular rule
Machine learning	A method for understanding the rule in the data and pulling useful information out of the data by means of algorithms
Data mining	Computing data for learning valuable information in large data sets by employing statistical models, machine learning and database systems
Pattern recognition	A kind of machine learning that mainly handles consistencies in a dataset
Deep learning	A type of machine learning for complex structures of neural networks
Artificial intelligence	The study of intelligent systems with ability to learn from circumstances by solving problems

The Machine Learning techniques that are employed in this work are Support Vector Regressions (SVR), Artificial Neural Network (ANN), Long Short-Term Memory (LSTM) and Recursive Neural Network (RNN).

3.2.1 Support Vector Regression (SVR)

The Support Vector (SV) algorithm [28] is based on the context of statistical learning theory. It is a nonlinear generalization of the Generalized Portrait algorithm and enables learning machines to generalize into unseen data. Since its formulation and development in the 1960s by Vapnik, Lerner and Chervonenkis in Russia, SV it has become an important tool in the Machine Learning field, especially in applications like object recognition and in regression and time series predictions.

3.2.1.1 Mechanism

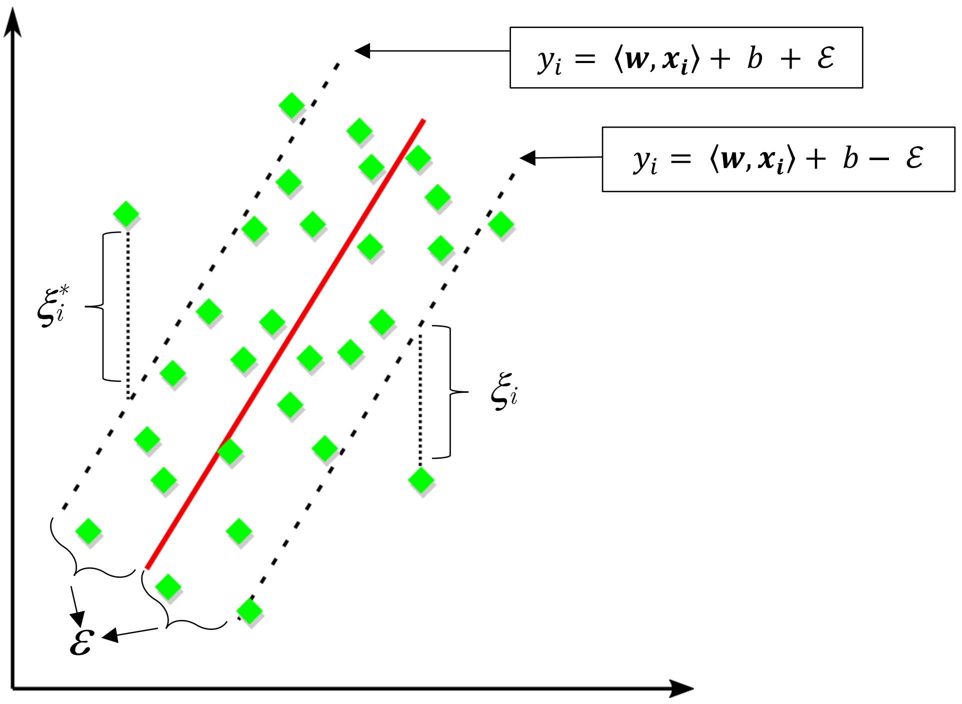
To describe the basic principles involved in SVR [28-30], consider a training dataset $\{(x_1,y_1), \ldots ,(x_l,y_l)\} \subset \chi \times \boldsymbol{R}$, where χ represents the space of the input patterns, say $\boldsymbol{R}^d$. So, the goal in ε-SVR is to find a function $f(x)$ that has at most ε deviation from the actually obtained targets y_i for all data in the training set while at the same time it is as flat as possible. This is to mean that errors are acceptable so long as they do not exceed the threshold ε. As shown in Figure 3.2 where a one-dimensional SVR is described, the data points represent the predicted values y_i and the solid red line depicts the reference data, y. Found at exactly at the distance of ε on either side are the two dashed lines which bound the tubular area out of which the data points used for prediction are found. So, training an SVR entails solving the function

$$\text{minimize} \quad \frac{1}{2}\|\boldsymbol{w}\|^2 + C\sum_{i=1}^{n}(\xi_i^* + \xi_i) \qquad 3.1$$

$$\text{subject to} \quad \begin{cases} yi - \langle \boldsymbol{w}, \boldsymbol{x_i} \rangle - b \leq \mathcal{E} + \xi_i^* \\ \langle \boldsymbol{w}, \boldsymbol{x_i} \rangle + b - y_i \leq \mathcal{E} + \xi_i \end{cases} \qquad 3.2$$

where, $\boldsymbol{w}$ is the learned weight vector, $\boldsymbol{x}_i$ is the i^{th} training occurrence, y_i is the training label and ξ_i is the distance between the bounds – represented by the dotted line in Figure 3.2 One-dimensional representation of SVR model. – and predicted values outside of these bounds. Another important constant parameter, C, in equation 3.1 is a constraint set by the user to control the penalty imposed on those observations outside the y_i bounds which helps to prevent overfitting.

Another important task needed to make SVR capable of non-linear predictions. For such cases, we need to introduce a radial basis function (RBF) kernel by replacing the inner product part of equation 3.2 with it. This helps map the data to a higher dimensional feature space without which the SVR remains linear.



$y_i = \langle \mathbf{w}, \mathbf{x}_i \rangle + b + \mathcal{E}$
$y_i = \langle \mathbf{w}, \mathbf{x}_i \rangle + b - \mathcal{E}$
ξ_i^*
ξ_i
$\mathcal{E}$

Figure 3.2 One-dimensional representation of SVR model.

3.2.2 Artificial Neural Network

Artificial Neural Network (ANN) [30-32] is another most effective machine learning approach that is effective in pattern recognition. The principle in ANN is drawn from the working concept of the biological neurons in the living body in that learning in ANN happens in the same way like in the brain. The brain is excellent in recognizing patterns. This is evident in the fact that it is immediately that we recognize an object which we are looking at is what it is, as long as we have come across it in the past. This is because we have trained our brain with the information about the specific object. So, with the help of large number of brain cells called neurons, at the mere sight of the object the information associated with the description (pattern) is recognized, and therefore, knowledge is recalled.

In the same manner, ANN is a computational model made up of hundreds of artificial neurons. These single units of artificial neurons are connected with weights (coefficients) that make up the neural structure. Because they process input information, they are also called processing elements and each of them have weighted inputs, transfer function and one output. Each artificial neuron is constructed in the image of a biological neuron in that the functions of the two are similar. To produce the output for a particular artificial neuron, the inputs are multiplied by the combined connection weights and then passing them through a transfer function. The sigmoid function is the most commonly used transfer function. Figure 3.3 represents a model of a single artificial neuron.

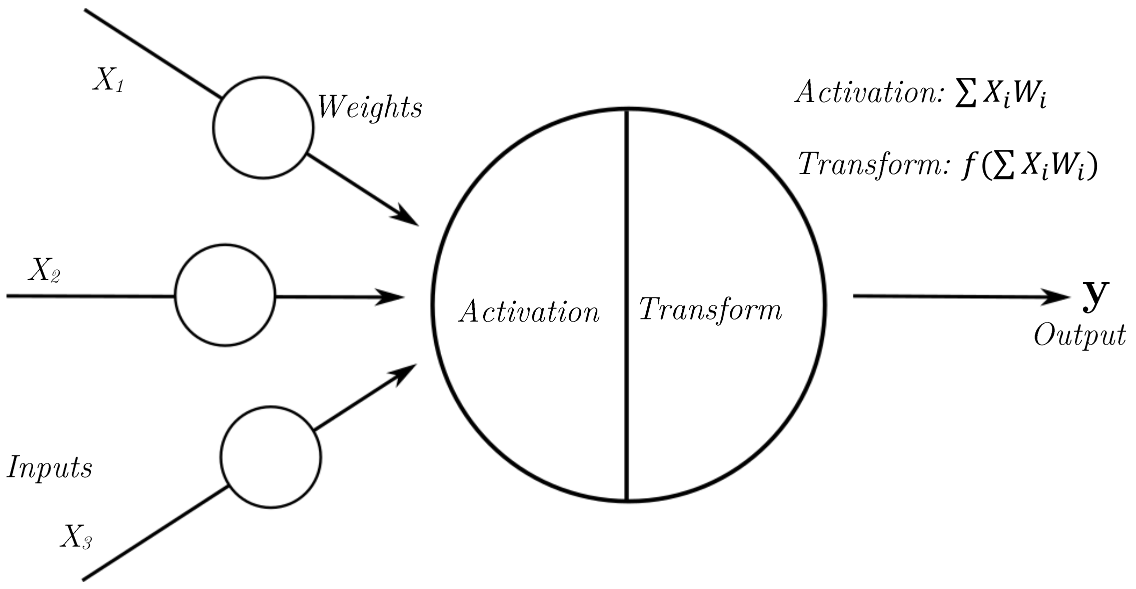


Figure 3.3 Representation of a single artificial neuron.

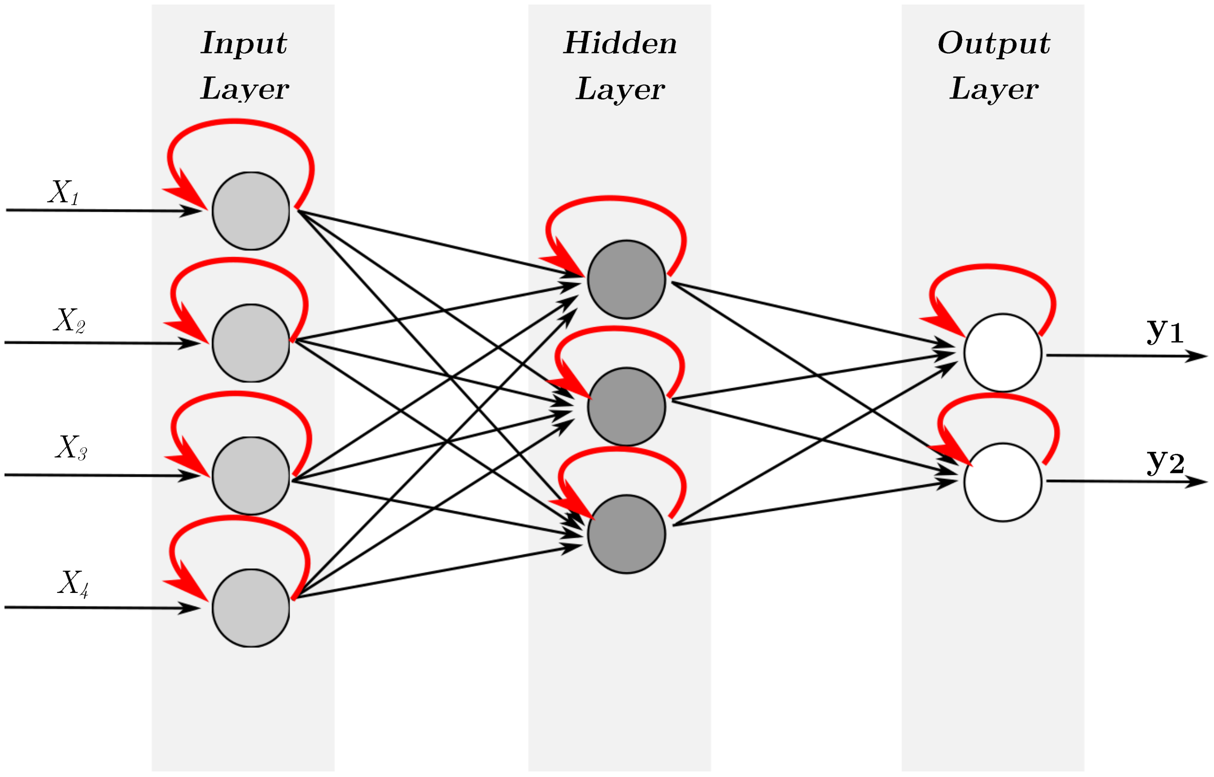


Figure 3.4 Representation of both feedback and feedforward connection types. The red lines denote the additional input direction in a network with feedback mechanism.

Artificial Neural Networks architecturally consists three layers: [31,33]

- **Input layer** of the neural network is where neurons receive data or inputs,
- **Output layer** is the layer where the output neurons deliver responses based on the inputs and
- **Hidden layer** is the intermediate layer of the neural network found between the Input and Output layers that is made up of hidden neurons. This is layer is essential for learning as it is where the mapping of inputs to outputs takes place. Moreover, because during the training phase inputs are transformed by the connection weights, the number of neurons in this layer has an impact on the network performance, and ultimately, learning process. This means, having too few neurons in the hidden layer slows down the learning process while having too many results in overtraining which leads to weakened prediction abilities.[33]

Connection of these neural units forms the artificial neural network. There are however a number of different types of connections that impact the operation of the network. Figure 3.4 illustrates the feedforward network and the feedback network connection types. Feedback connection type is a common one where the output of a layer goes back to the input of the prior layer. In Figure 3.4 the red arrows path denotes the backward flow. On the other hand, in feedforward connections, there is no such backward input from the output neurons and therefore such connections do not have the one extra degree of freedom that comes with having an additional weight.

There are several types of ANNs in use. The network's arrangement or the ANN models and the type of computations or algorithm are the two most important factors needed to be considered when considering neural networks for data analysis.

3.2.3 Recurrent Neural Network (RNN)

Recurrent Neural Network (RNN) [34,35] is a machine learning archetype that is similar to the structure and working theory of that of ANN but with one or more feedback loop(s) built in the structure. The feedback loops in RNN are sequences or recurrent cycles over time. Requiring a dataset of input-target pairs, the objective in this architecture is to decrease the variance between the output and the target pairs which is called the loss value. This minimizing of difference achieved by finetuning the weights of the network. Because of the feedback loops and the circular connections between higher and lower layers, data propagates from earlier events to current steps. This propagation of data between the processing steps forms a memory of time in RNNs.

As mentioned above, just like ANNs, RNN has the three layers: input layer, output layer and hidden layer. However, in this case, in addition to new inputs, the state of the previous sequential input is also entered which will provide a short memory regarding the previous state. Figure 3.5 depicts the basic architecture of RNN.

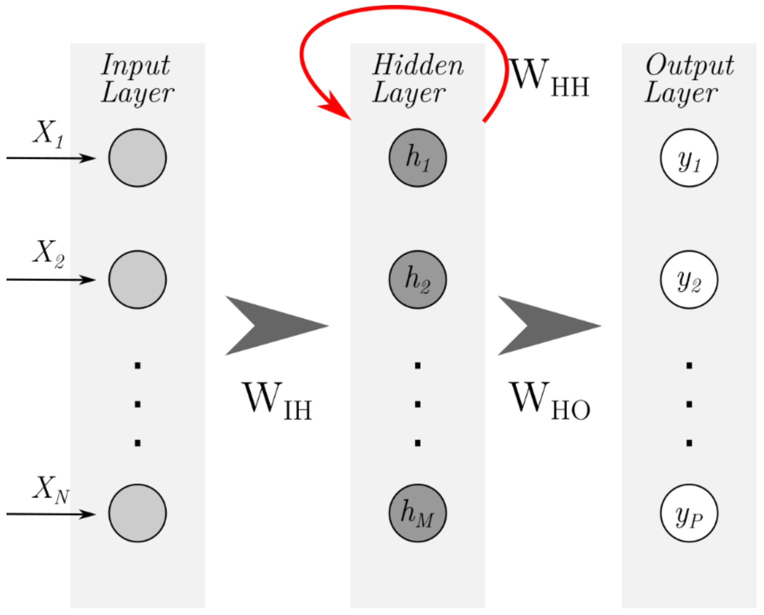


Figure 3.5 Depiction of basic architecture of folded RNN.

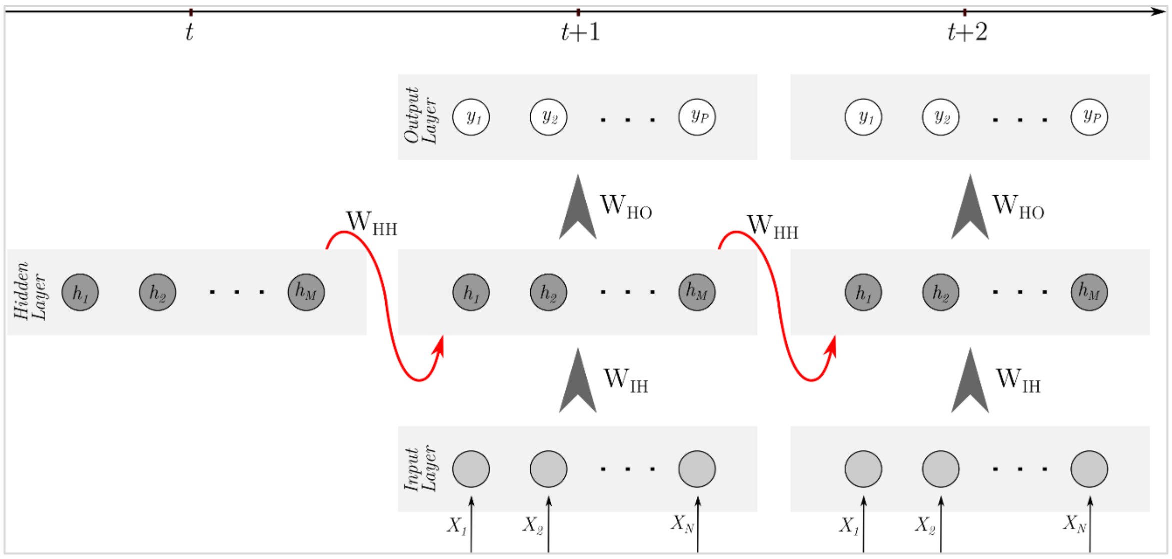


Figure 3.6 Basic depiction of Unfolded RNN across timesteps.

Consider the basic RNN structure represented in Figure 3.5 and Figure 3.6 with the input layer having N units, the hidden layer M units and the output layer P units. The inputs to the input layer are a sequence of vectors across time t such that $\{..., X_{t-1}, X_t,$

X_{t+1}, ...}, where $X_t = (x_1, x_2, ..., x_N)$. These inputs are connected to the units in the hidden layer with a weight matrix noted as W_{IH}. And, the units in the hidden layer form a recurrent connection to each other across time such that $\mathbf{h}_t = (h_1, h_2, ..., h_M)$. With these parameters, the "memory" of the whole system is defined by the hidden layer as

$$\boldsymbol{h}_t = f_H(\boldsymbol{O}_t), \qquad 3.3$$

where
$$\boldsymbol{O}_t = W_{IH}X_t + W_{HH}\boldsymbol{h}_{t-1} + \boldsymbol{b}_h, \qquad 3.4$$

$\mathbf{b}_h$ is the bias vector and $f_H(\cdot)$ is the activation function of the hidden layer. W_{HO} represents the weighted connection between the hidden and the output layers. The units in the output layer $y_t = (y_1, y_2, ..., y_P)$ are then calculated as

$$y_t = f_O(W_{HO}\boldsymbol{h}_t + \boldsymbol{b}_O) \qquad 3.5$$

where, $\mathbf{b}_O$ is the bias vector and $f_O(\cdot)$ is the activation function in the output layer. These steps are repeated over time $t = (1, 2, ..., T)$ because the pairs formed by the input and the target are serial across the timesteps. Moreover, based on the input vector, the hidden units provide a prediction at the output layer in each timestep.

As mentioned earlier, the objective in RNNs is to minimize the loss value – the difference between the output and the target pairs. The loss function $\mathcal{L}$ is used to perform this evaluation by comparing the output $\mathbf{y}_t$ with its target pair $\mathbf{z}_t$. The loss function calculates the sum total of all losses in each timestep as

$$\mathcal{L}(\boldsymbol{y},\boldsymbol{z}) = \sum_{t=1}^{T} \mathcal{L}_t\,(\boldsymbol{y}_t,\boldsymbol{z}_t). \qquad 3.6$$

Of the many available recursive neural networks, most common mechanisms are real-time recurrent learning (RTRL), where the gradient information is propagated forward and backpropagation through time (BPTT) learning model where the network is unfolded to form a feedforward neural network to help update the weights.

3.2.4 Long Short-Term Memory (LSTM)

Long short-term memory (LSTM) [34-39] model is a model that is based on recursive neural network architectures. The variable size of input/output neurons in RNN is very large and so there needs to be too much computation demanding computational resources. This, coupled with the lack of parameter sharing between the hidden units, limits the memory achieved from the recurrent connections. Though RNN connections provide learning opportunity through the sequential dependencies, the model fails to deliver one that can be considered long term learning.

Instead of activation functions, LSTM model makes use of memory cells where flow of information is controlled by blocks called “gates”. While preserving the extracted features from previous timesteps, the gates manage information flow to the hidden neurons. The need for control of flow is to avoid input and output weight conflicts. The gates involved in this architecture are: input gate which controls flow of inputs to the memory cell, the output gate that controls the output control of cell activations in the network and the forget gate, one that is added to the memory block. So, these gating units along with one or more memory cells form the basic or memory unit of an LSTM model. Such a memory cell has a unit called constant error carousels (CEC) which is recurrently self-connected.

Due to inability to detect relevant information in naturally reoccurring continuous sequences, the forget gate learns weights controlling the rate of decay for values stored in the memory cell. So, a memory cell holds its value over a period of time when the forget gate is not the cause of decay and when both the input and output gates are off. This mechanism enables the network structure to hold information or “remember” for longer period time. Generally speaking, though the memory is improved, LSTM employs more parameters than a simple RNN, making it complex. Figure 3.7 depicts shows a typical LSTM cell.

With $\boldsymbol{W}_{Ig^i}$ as the weight matrix from the input layer to the input gate, $\boldsymbol{W}_{Hg^i}$ that from hidden state to the input gate, $\boldsymbol{W}_{g^c g^i}$ from cell activation to the input gate and $\boldsymbol{b}_{g^i}$ is the bias of the input gate, the input of LSTM can be defined as:

$$g_t^i = \sigma(\boldsymbol{W}_{Ig^i}\boldsymbol{X}_t + \boldsymbol{W}_{Hg^i}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{g^c g^i}\boldsymbol{g}_{t-1}^c + \boldsymbol{b}_{g^i}. \qquad 3.7$$

Similarly, with $\boldsymbol{W}_{Ig^f}$ as the weight matrix from the input layer to the forget gate, $\boldsymbol{W}_{Hg^f}$ as that from hidden state to the forget gate, $\boldsymbol{W}_{g^c g^f}$ as that from cell activation to the forget gate and $\boldsymbol{b}_{g^f}$ as the bias of the forget gate, the forget gate is defined as:

$$g_t^f = \sigma(\boldsymbol{W}_{Ig^f}\boldsymbol{X}_t + \boldsymbol{W}_{Hg^f}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{g^c g^f}\boldsymbol{g}_{t-1}^c + \boldsymbol{b}_{g^f}. \qquad 3.8$$

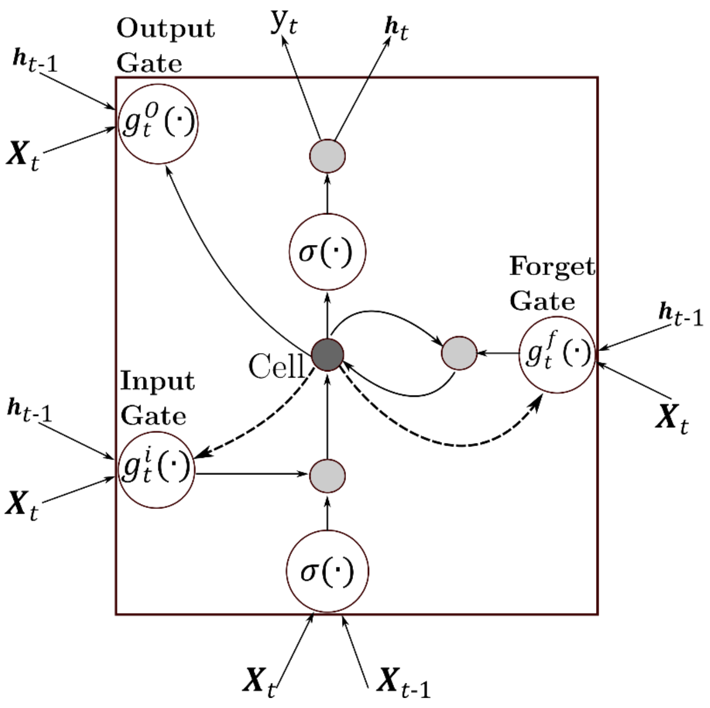


Figure 3.7 The basic LSTM memory unit where the dashed line represents time lag.

In addition, with $\boldsymbol{W}_{Ig^o}$ as the weight matrix from the input layer to the output gate, $\boldsymbol{W}_{Hg^o}$ as that from hidden state to the output gate, $\boldsymbol{W}_{g^cg^o}$ as that from cell activation to the output gate and $\boldsymbol{b}_{g^o}$ as the bias of the output gate, the output gate is defined as:

$$g_t^O = \sigma(\boldsymbol{W}_{Ig^o}\boldsymbol{X}_t + \boldsymbol{W}_{Hg^o}\boldsymbol{h}_{t-1} + \boldsymbol{W}_{g^cg^o}\boldsymbol{g}_{t-1}^c + \boldsymbol{b}_{g^o}. \qquad 3.9$$

Likewise, with $\boldsymbol{W}_{Ig^c}$ as the weight matrix from the input layer to the cell gate, $\boldsymbol{W}_{Hg^c}$ as that from hidden state to the cell gate, and $\boldsymbol{b}_{g^c}$ as the bias of the cell gate, the cell gate is defined as:

$$g_t^c = g_t^i tanh(\boldsymbol{W}_{Ig^c}\boldsymbol{X}_t + \boldsymbol{W}_{Hg^c}\boldsymbol{h}_{t-1} + \boldsymbol{b}_{g^c} + \boldsymbol{g}_t^f\boldsymbol{g}_{t-1}^c. \qquad 3.10$$

And finally, the hidden state can be calculated as

$$\boldsymbol{h}_t = \boldsymbol{g}_t^o \tanh(\boldsymbol{g}_t^c). \qquad 3.11$$

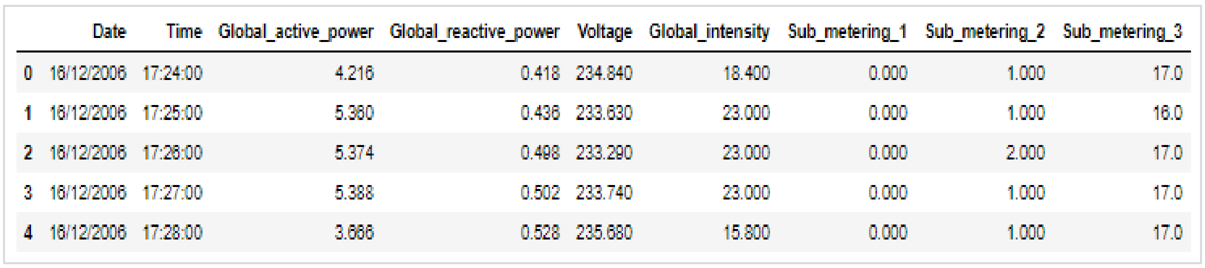
4. Experiment and Result

4.1 Experiment

As mentioned in Section 2, the aim of this work is to predict power consumption using mathematical models on both Fog and Cloud devices and compare the resource usage among them. Furthermore, this work also compares prediction results among the models. The mathematical models that are used for this task are those discussed in Section 3.

4.1.1 Data Preprocessing

The dataset used in this thesis is the “Individual household electric power consumption Data Set” from UCI machine learning repository [40]. Data computation was performed both a personal computer and on a cloud server.



	Date	Time	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2	Sub_metering_3
0	16/12/2006	17:24:00	4.216	0.418	234.840	18.400	0.000	1.000	17.0
1	16/12/2006	17:25:00	5.360	0.436	233.630	23.000	0.000	1.000	16.0
2	16/12/2006	17:26:00	5.374	0.498	233.290	23.000	0.000	2.000	17.0
3	16/12/2006	17:27:00	5.388	0.502	233.740	23.000	0.000	1.000	17.0
4	16/12/2006	17:28:00	3.666	0.528	235.680	15.800	0.000	1.000	17.0

Figure 4.1 Sample of the top five records of power consumption data.

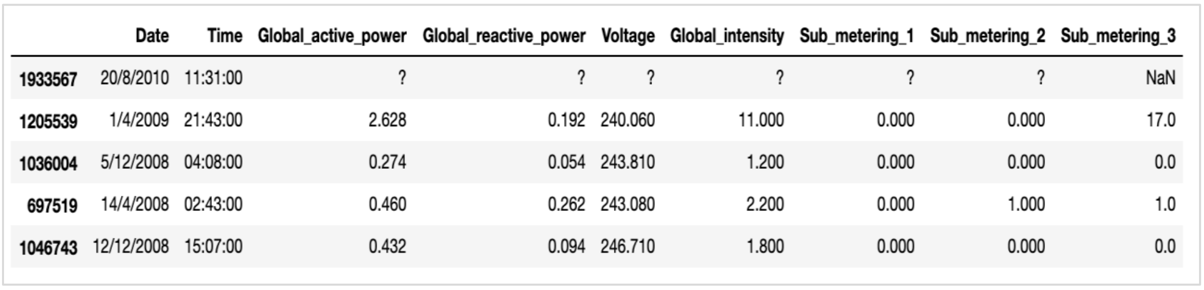
The data from machine learning repository of the Center for Machine Learning and Intelligent Systems at the University of California, Irvine measures electric power consumption by a household [40]. The data represents a reading of close to four years of consumption taken every minute. This multivariate time series real data of a house located in Sceaux in France, contains 2075259 readings in total and the measurements were taken from December 2006 to November 2010.

As can be seen from Figure 4.1, the dataset has nine variables:

- date: Date in dd/mm/yyyy format
- time: time in hh:mm:ss format
- global_active_power: total minute-averaged active power of the household in kilowatt
- global_reactive_power: total minute-averaged reactive power of the household in kilowatt
- voltage: minute-averaged voltage in volt

- global_intensity: total minute-averaged current intensity for the household in amperes
- sub_metering_1: energy sub-metering No. 1 in watt-hour which corresponds to appliances in kitchen such as a dishwasher, an oven and a microwave.
- sub_metering_2: energy sub-metering No. 2 in watt-hour that corresponds to appliances in the laundry room such as washing-machine, tumble-drier, a refrigerator and a light.
- sub_metering_3: energy sub-metering No. 3 in watt-hour which corresponds to an electric water-heater and an air-conditioner.

The number of rows, records, and the data types of the fields can be learned from the metadata of the dataset. The dataset has 2,075,259 and the file type is object. Moreover, as can be seen from the random sample data in Figure 4.2, the data contains missing records.

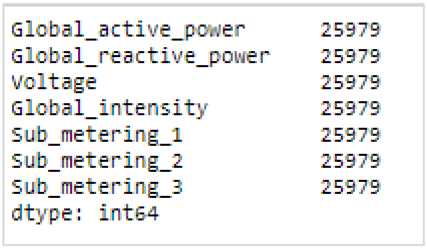


	Date	Time	Global_active_power	Global_reactive_power	Voltage	Global_intensity	Sub_metering_1	Sub_metering_2	Sub_metering_3
1933567	20/8/2010	11:31:00	?	?	?	?	?	?	NaN
1205539	1/4/2009	21:43:00	2.628	0.192	240.060	11.000	0.000	0.000	17.0
1036004	5/12/2008	04:08:00	0.274	0.054	243.810	1.200	0.000	0.000	0.0
697519	14/4/2008	02:43:00	0.460	0.262	243.080	2.200	0.000	1.000	1.0
1046743	12/12/2008	15:07:00	0.432	0.094	246.710	1.800	0.000	0.000	0.0

Figure 4.2 Sample of the data taken randomly form the set.

The data and time fields were combined and used to create a new field, named date_time. The new field that replaced the two date and time.

The missing records, specified by the character '?' in Figure 4.2, were replaced with 'NaN' values. This step will help to keep the datatype of the records as float making computations possible. Along with this process, the data type of the Global_active_power field was also changed to float data type. After replacing the missing values with 'NaN' values, inquiring for the number of missing records show that there were 25,979 records of them in each field as can be seen in Figure 4.3. The missing values were then replaced and filled with the mean values to have a complete set of data.

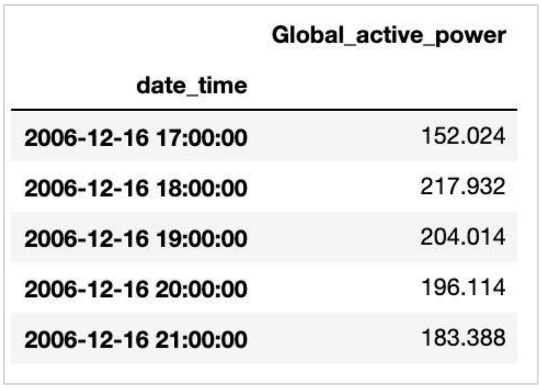


```
Global_active_power      25979
Global_reactive_power    25979
Voltage                  25979
Global_intensity         25979
Sub_metering_1           25979
Sub_metering_2           25979
Sub_metering_3           25979
dtype: int64
```

Figure 4.3 Number of missing records in the dataset.

Next, the Global_active_power attribute along with the date_time field was extracted from the set as it is the only attribute this work needs. Moreover, to make sure the data time series is not broken, the data was sorted in an ascending order and the newly added date_time field was used to index the data.

As mentioned before and as can be seen from Figure 4.1, the data was collected every minute. So, the data was resampled to create an hourly record by summing up each all the values in each hour. Figure 4.4 shows the first few lines of the data that is ready for use in the models for calculation.



date_time	Global_active_power
2006-12-16 17:00:00	152.024
2006-12-16 18:00:00	217.932
2006-12-16 19:00:00	204.014
2006-12-16 20:00:00	196.114
2006-12-16 21:00:00	183.388

Figure 4.4 First few lines of the final preprocessed dataset

After the preprocessing was all done, the data was divided in two sets for use in the computational models: training dataset and testing dataset. For training, the data record with date ranges from 01/01/2007 – 31/12/2008, two years' data, was used. And for testing, a data set with date range from 01/01/2009 – 31/01/2009, which is a month's data, was used. The mean and standard deviation values for the two year Global_active_power hourly data, are 66.22 (KW) and 57.30 (KW) respectively. The standard deviation will be used as a threshold in error calculations to evaluate the performances of the models.

Due to the nature of the data, in that values vary from very high to low, using it directly in the models for prediction calculation will not be optimal. Therefore, before using the "Global_active_power" values in the model calculations, the time series values were normalized to be scaled between 0 and 1 using Min Max Scaler.

The machine learning computation was performed using this normalized data set. However, after each computation was performed, each data value was inverse transformed to the original value which was then used to check for error value.

4.1.2 Error Calculations

4.1.2.1 Mean Squared Error

Mean Square Error (MSE) is calculated as the average of the squared differences between the predicted and actual values. The result is always positive in the range between 0 and 1 and a perfect value is 0. For *n* numbers of observations, MSE is defined by the formula

$$MSE = \frac{\sum_{i=1}^{n}(\boldsymbol{P}_i - \boldsymbol{O}_i)^2}{n} \qquad 4.1$$

where, $\boldsymbol{P}_i$ is the predicted time series value and $\boldsymbol{O}_i$ is the observed time series value.

Training loss can be seen as a distance between the true values of the problem and the values predicted by the model. Hence, the greater the loss means the more erroneous the data is.

4.1.2.2 Root Mean Square Error

To calculate the accuracy of the predictions obtained from the models used Root Mean Square Error (RMSE) is used. RMSE [41] is a standard means of measuring the error of a model in predicting time series numerical data. The technique takes the square of the difference between the observed and the predicted values and divide the sum of all such values by the number of all observations and then calculate the square root to calculate the error. For *n* number of observations, RMSE is calculated by the formula

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(\boldsymbol{P}_i - \boldsymbol{O}_i)^2}{n}} \qquad 4.2$$

where, $\boldsymbol{P}_i$ is the predicted time series value and $\boldsymbol{O}_i$ is the observed time series value.

In this work, the standard deviation of the two year Global_active_power hourly data used as a threshold, lower value of RMSE indicates that the performance of the prediction model is good.

4.1.3 System Specification

The prediction computations were performed on two different devices: on the cloud and on a local personal computer acting as fog. The system specifications such as the memory or RAM and processor speed of both machines used as the cloud and the fog are summarized in Table 4.1.

Table 4.1 System specifications of devices used for computing the prediction models.

Device	Memory (RAM)	Processor
Fog	12 GB	2.70GHz dual-core Intel Core i5, 64- bit
Cloud	25GB	2.30GHz quad-core Intel(R) Xeon(R)

4.1.4 Tools

The computing tool used are the libraries in the Python programing language; mainly, Tensorflow, pandas, numpy sklearn, matplotlib. Jupyter, a browser-based development environment which runs on Anaconda system was the main environment used to perform the computations for this work.

4.2 Results

The machine learning calculations in all the models were done using the training and testing data sets assigned for the task and prediction are performed for one month. The following are the results obtained from computations made on a Fog device for each model.

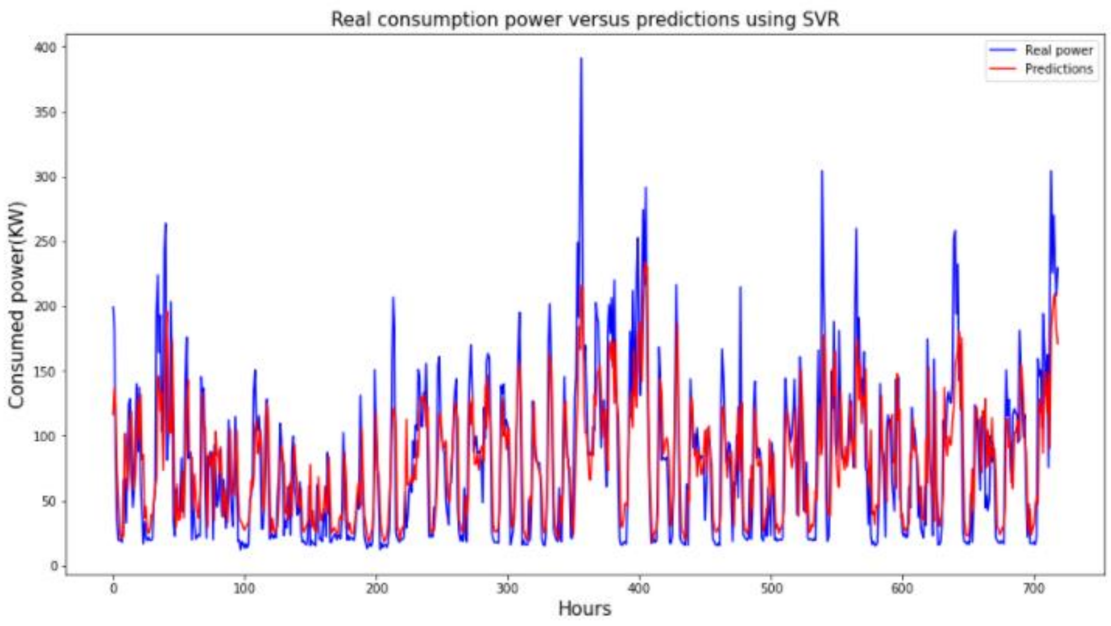


Figure 4.5 Plots of real power consumption and predicted data that were generated using one month test data in *SVR* machine learning model on Fog device.

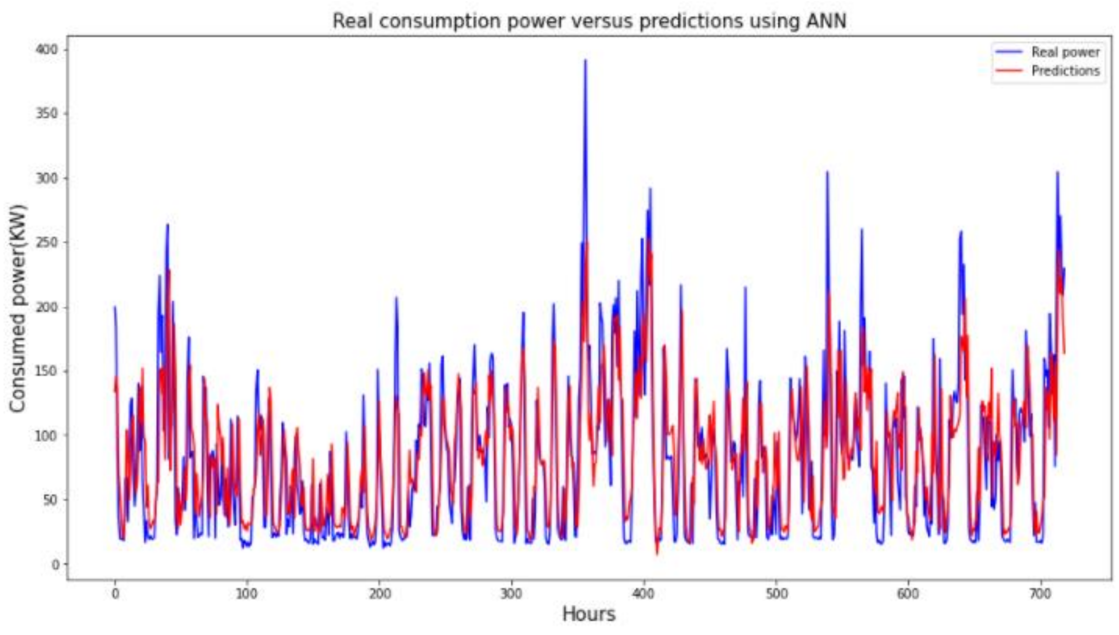


Figure 4.6 Plots of real power consumption and predicted data that were generated using one month test data in *ANN* machine learning model on Fog device.

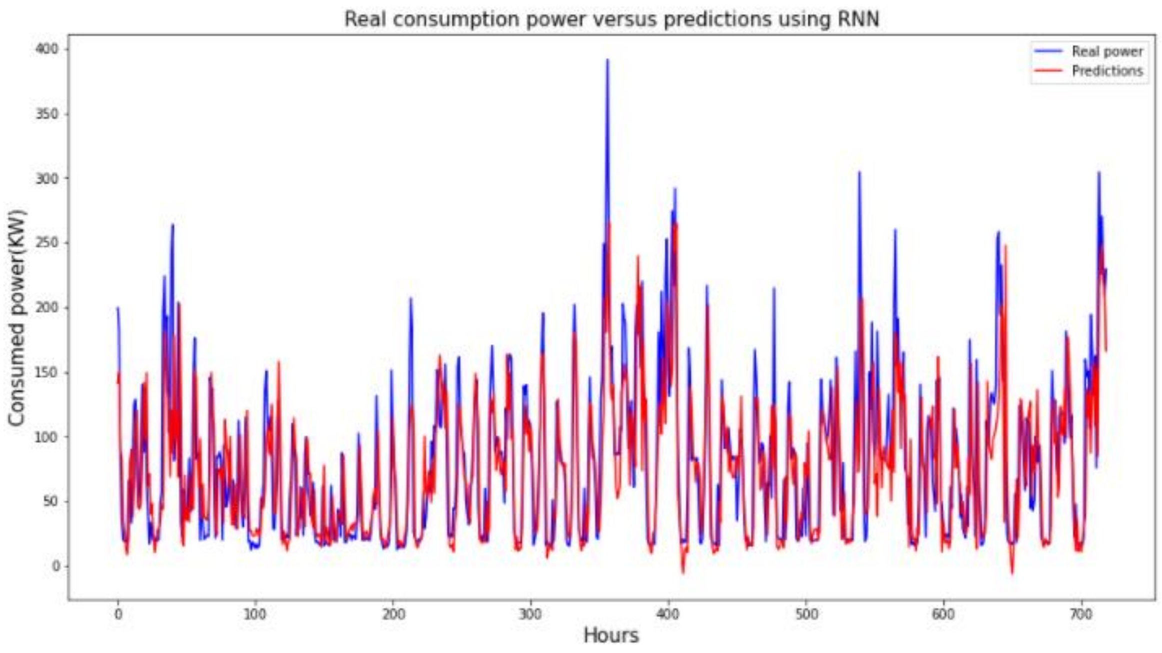


Figure 4.7 Plots of real power consumption and predicted data that were generated using one month test data in *RNN* machine learning model on Fog device.

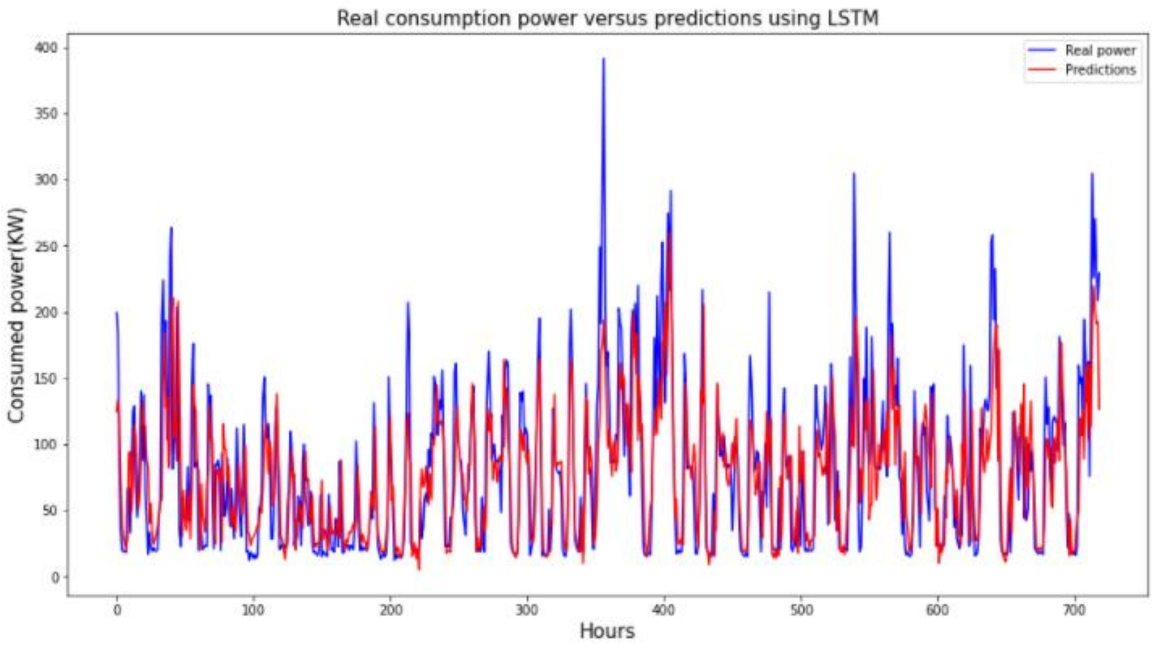


Figure 4.8 Plots of real power consumption and predicted data that were generated using one month test data in *LSTM* machine learning model on Fog device.

RMSE scores were also calculated for each of the models and the results are summarized in the following table.

Table 4.2 RMSE scores in KW of the four models run on Fog device

Model	**RMSE Scores**
SVR	40.89
ANN	39.50
RNN	39.58
LSTM	40.80

Moreover, the deep learning models' training losses are shown in the following table.

Table 4.3 Training loss values of the models used run on Fog device.

Model	**Training Loss Value**
ANN	0.0091
RNN	0.0084
LSTM	0.0079

Resource consumptions of all the computation models were also measured and compared. The following graphs show the results for memory and CPU utilizations by the models that run on a Fog device.

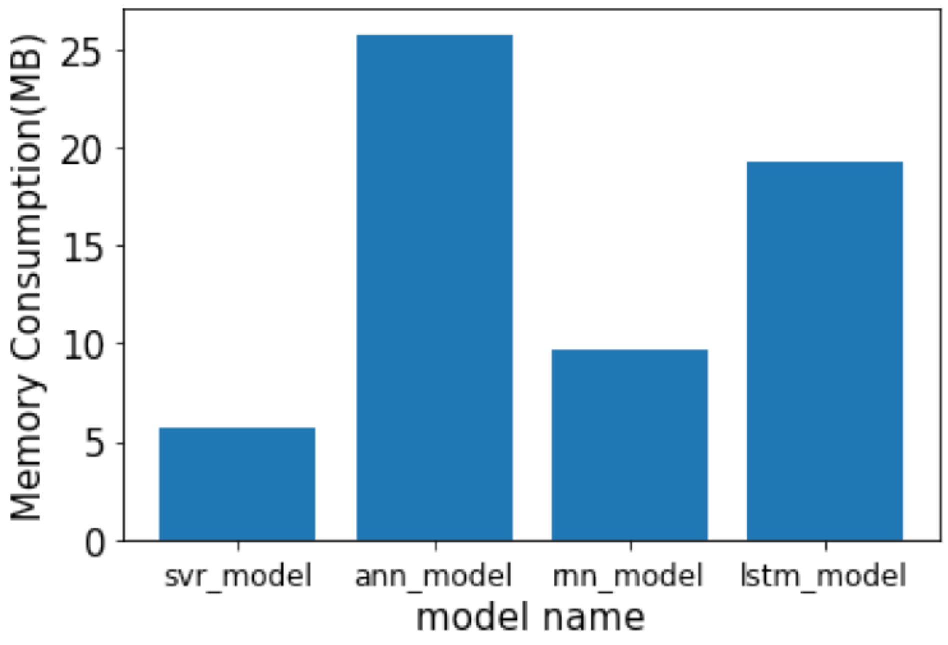


Figure 4.9 Comparison of memory consumption (measured in MB) of each model run on a Fog device.

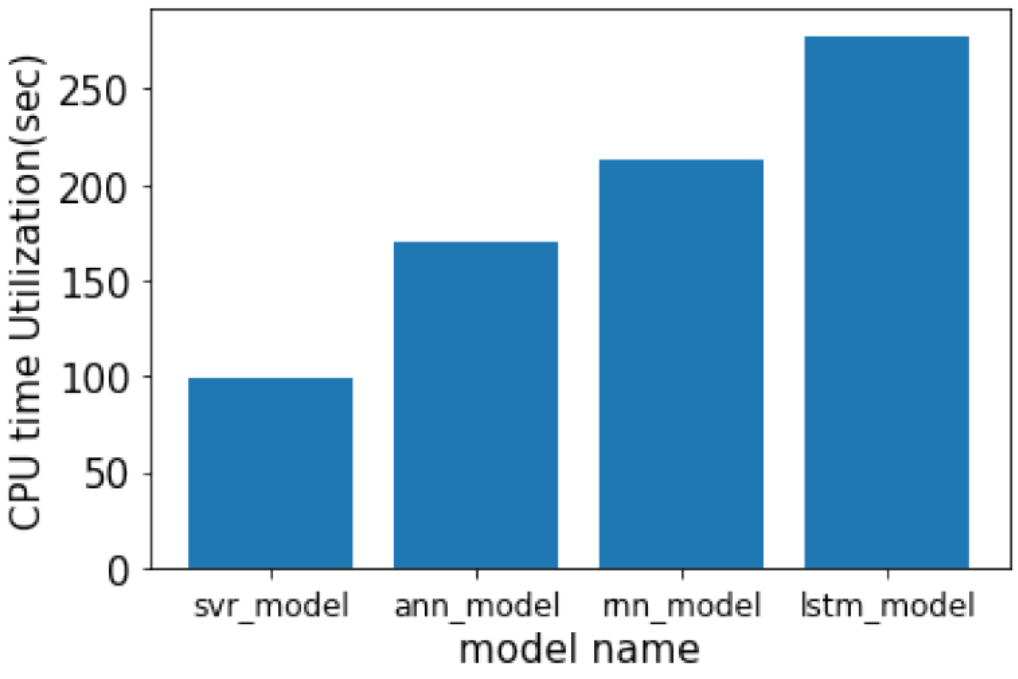


Figure 4.10 Comparison of CPU utilization time (measured in seconds) of each model run on a Fog device.

Similar to all the experimental computations performed on a Fog device, computations were also run on a Cloud device using one month test data.

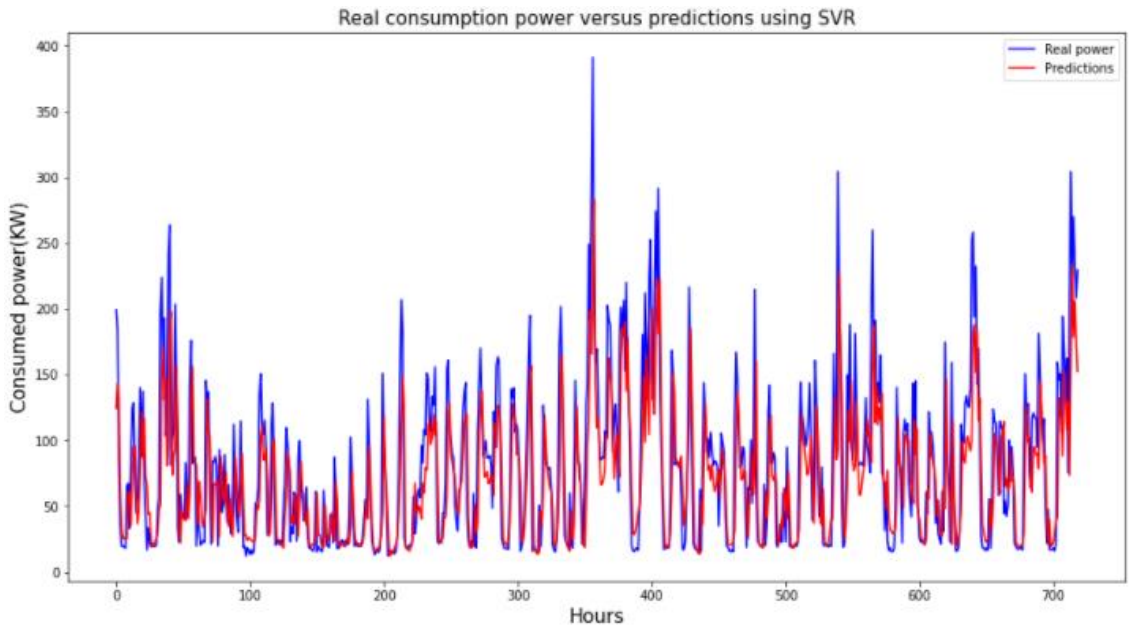


Figure 4.11 Plots of real power consumption and predicted data that were generated using one month test data in *SVR* machine learning model on Cloud device.

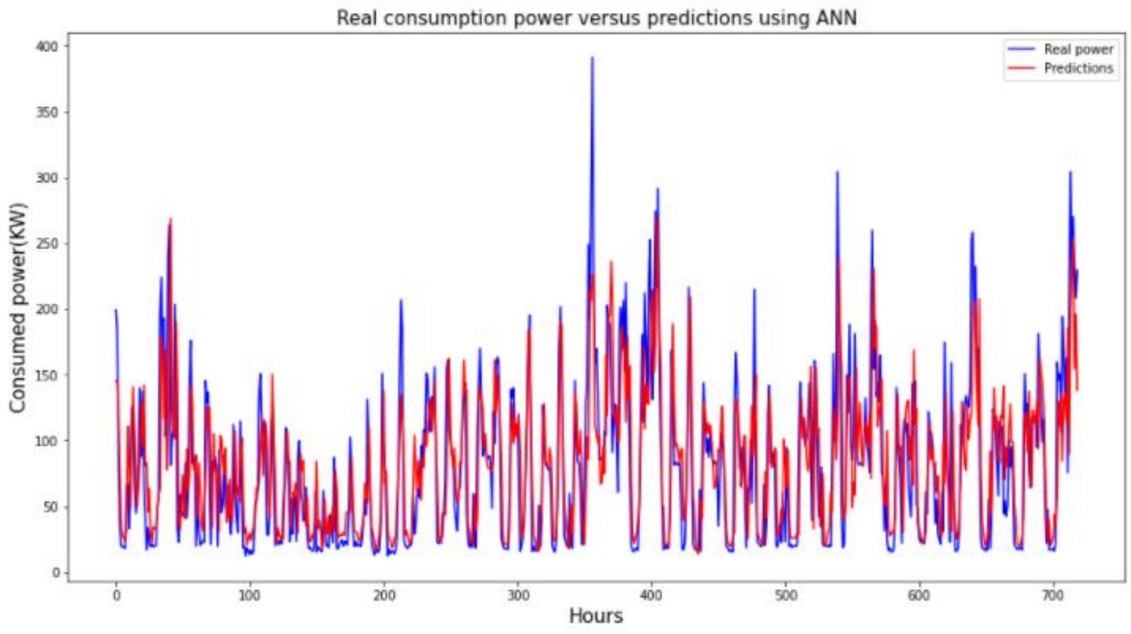


Figure 4.12 Plots of real power consumption and predicted data that were generated using one month test data in *ANN* machine learning model on Cloud device.

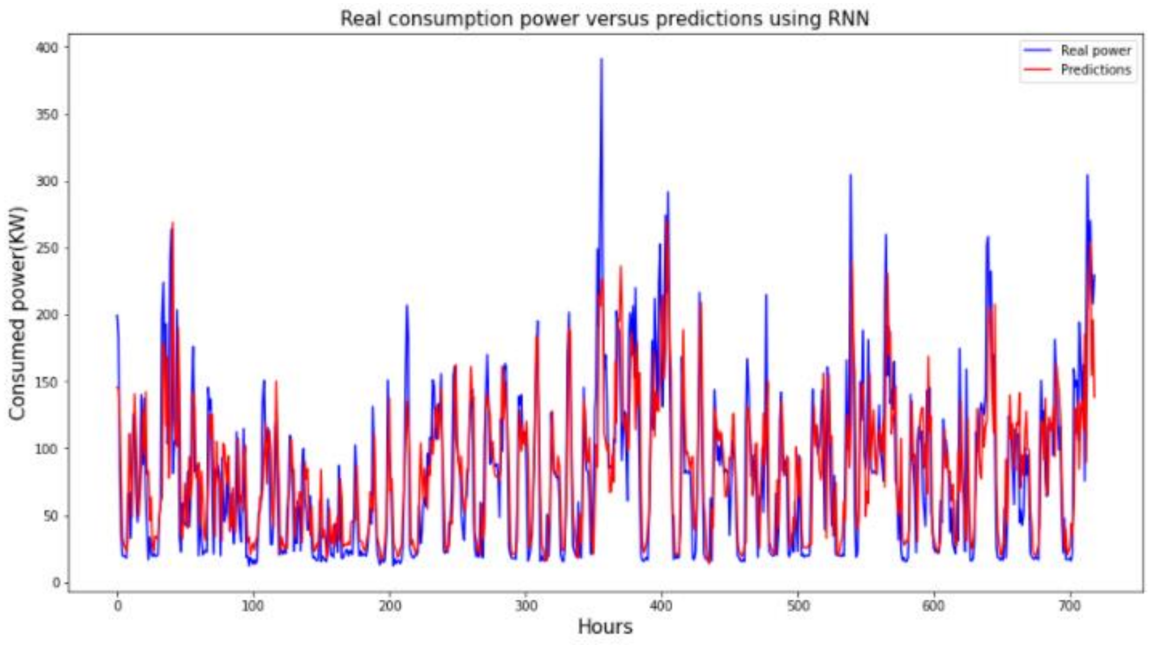


Figure 4.13 Plots of real power consumption and predicted data that were generated using one month test data in *RNN* machine learning model on Cloud device.

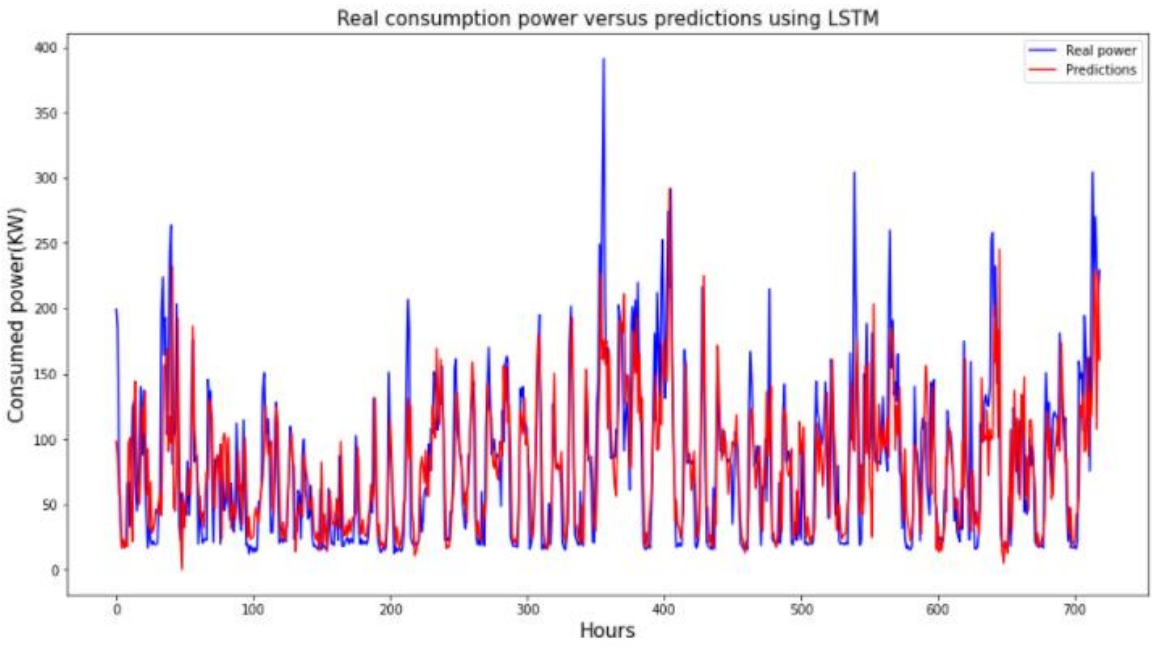


Figure 4.14 Plots of real power consumption and predicted data that were generated using one month test data in LSTM machine learning model on Cloud device.

Once again, similar to the case in Fog computing, RMSE scores were also calculated for each of the models for computations performed on a Cloud device and the results are summarized in the following table.

Table 4.4 RMSE scores in KW of the four models run on Cloud device

Models	RMSE Scores
SVR	40.89
ANN	39.92
RNN	39.75
LSTM	42.15

In the same manner, the deep learning models' training losses for computations made on a cloud device are shown in the following table.

Table 4.5 Training loss values of the models used run on a Cloud device.

Models	Training Loss Values
ANN	0.0090
RNN	0.0083
LSTM	0.0067

For Cloud computations also, resource consumptions of all the models were measured and compared. The following graphs show the results for memory and CPU utilizations by the models.

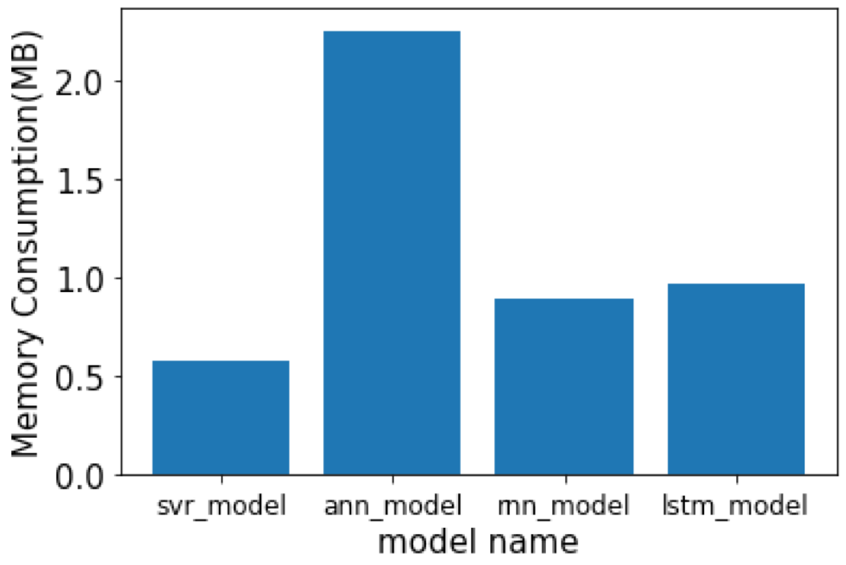


Figure 4.15 Comparison of memory consumption (measured in MB) of each model run on a Cloud device.

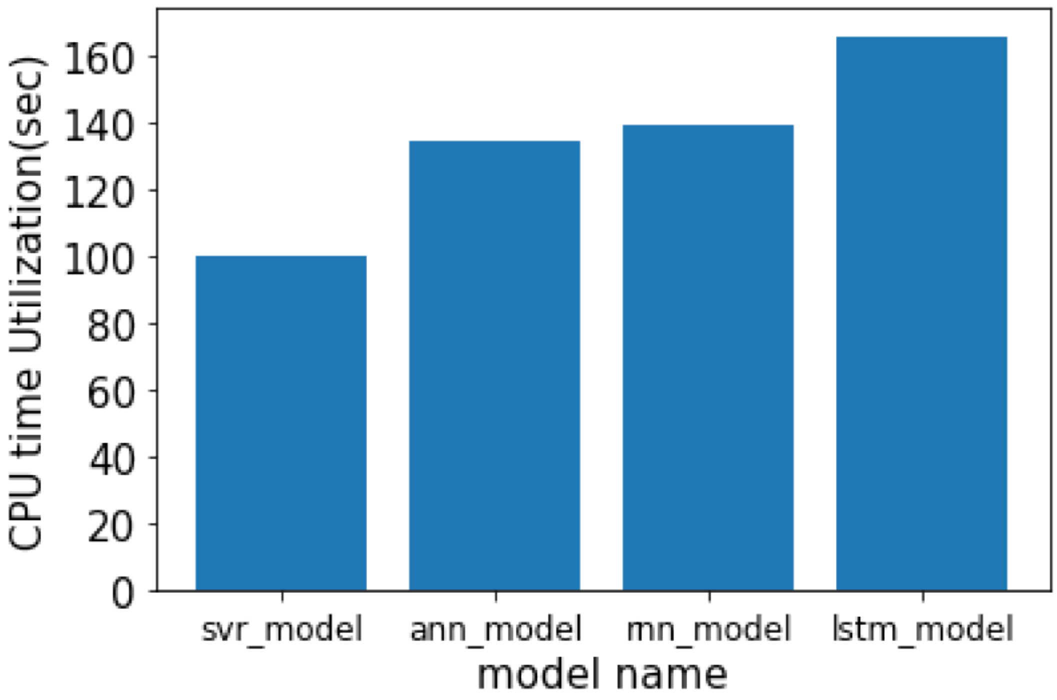


Figure 4.16 Comparison of CPU utilization time (measured in seconds) of each model run on a Cloud device.

5. Discussion and Analysis

As discussed in section 4.1.2.2, the lower the RMSE values are than the threshold value the better are the performances of the models. The RMES scores for all models in both fog and cloud, as presented in Table 4.2 and Table 4.4, show that the RMSE scores for all the models are comparatively similar and smaller than the threshold by small magnitude. This implies that the performances of all the computation models implemented on both Fog and Cloud devices are the similar and in the acceptable range.

Similarly, as discussed in section 4.1.2.1, the MSE loss function value that is closer to 0, indicates higher confidence of in the performance of prediction models. The training loss results presented in Table 4.3 and Table 4.5 for all the three deep-learning models, ANN, RNN and LSTM, run on both Fog and Cloud devices are closer to zero. This shows that all the predictions made with these models have high confidence of performance. Moreover, the results show that among the deep-learning models used in this work, LSTM has the lowest training loss value in both Fog and Cloud devises indicating that its performance is better than those of ANN and RNN.

Memory usage and CPU utilization time results are presented in the figures: Figure 4.9, Figure 4.10, Figure 4.15 and Figure 4.16. The graphs show that CPU utilization time and memory consumption for each model are different between the Fog and Cloud computation devices. For Cloud computations, CPU time for SVR, ANN, RNN and LSTM are 100, 134, 139 and 166 seconds and for computation on a Fog device they are 100, 171, 213 and 278 seconds respectively. Furthermore, for computations made on a Cloud server, memory usage for models SVR, ANN, RNN and LSTM are 0.57, 2.25, 0.89, and 0.97 MB and for that of on Fog device are 5.66, 25.75, 9.66 and 19.27 MB respectively. These results suggest that memory usage and CPU time performances are better in Cloud device than in Fog. However, these rather insignificant improved performances can be attributed to the difference in system specifications of both devices. As detailed in section 4.1.3, the system specifications of the Cloud server are much better than that of the devise used as a Fog device.

Fog devices, despite having low processing capacity, reside near the sensor networks and, therefore, the transmission delay to send the huge sensor Fog device is less. Whereas, the Cloud server is centralized with high processing capacity. The sensor data has to pass from the local area network to the IP network and then to the Cloud in order to be processed. The processing result has to be sent back via the same network for decision making. The networks between sensor and Cloud have limited bandwidth and many sensor networks are connected to the same IP network which usually creates delay in data transmission. Therefore, Fog device processing is highly advantageous as it is fast, secure and localized. To summarize, Fog Computing is a better choice for power consumption prediction applications based on our experimental results.

6. Conclusion and Future Direction

6.1 Conclusion

In this project, by using dataset from UCI machine learning repository, forecasts of one month power consumption were done. The predictions were done using four different machine learning models: SVR, ANN, RNN and LSTM. The forecasts that were performed on both Fog and Cloud devices were found to be in the acceptable error range and so reliable. However, when comparing the performances of the models in predicting power consumption computations on both Fog and Cloud devices, the model LSTM is found to be better. Finally, memory usage and CPU time of all the models was examined and results suggest that Fog computation is better than that of Cloud.

6.2 Future Direction

For future work, measuring of the actual network bandwidth and communication delay from sensors to the Fog device and from Fog device to Cloud server is planned to be looked into. Moreover, the Fog experiment is planned to be performed on an Edge device from Nvidia called Nvidia Nano. In addition, measuring the power consumption of running the IOT application on Fog and Cloud devices is planned.

7. Reference

[1] S. Murugesan and I. Bojanova, “Cloud computing: An overview” in Encyclopedia of cloud computing, IEEE, 2016, PP. 1-14, doi: 10.1002/9781118821930.ch1.

[2] K. Chandrasekaran, “Essentials of Cloud Computing.” United Kingdom, CRC Press, 2014.

[3] Cloud Service Alliance, “The Notorious Nine: Cloud Computing Top Threats in 2013,” Cloud Service Alliance, 2013. [Online]. Available: https://downloads.cloudsecurityalliance.org/initiatives/top_threats/The_Notorious_Nine_Cloud_Computing_Top_Threats_in_2013.pdf. [Accessed 13 04 2021].

[4] Cloud Security Alliance, “Top Threats to Cloud Computing: Deep Dive. A case study analysis for ‘The Treacherous 12: Top Threats to Cloud Computing’ and a relative security industry breach analysis.” Cloud Security Alliance, 2018. [Online]. Available: https://cloudsecurityalliance.org/artifacts/top-threats-to-cloud-computing-deep-dive. [Accessed 13 04 2021].

[5] J. Gubbi, R. Buyya, S. Marusic, M. Palaniswami. “Internet of Things (IoT): A vision, architectural elements, and future directions.” Future Generation Computer Systems, 2013, 29 (7), 1645–1660.

[6] M. R. Montgomery, “The urban transformation of the developing world,” Science, 2008, vol. 319, no. 5864, pp. 761–764.

[7] B. Tang et al., "Incorporating Intelligence in Fog Computing for Big Data Analysis in Smart Cities," in IEEE Transactions on Industrial Informatics, Oct. 2017, vol. 13, no. 5, pp. 2140-2150, doi: 10.1109/TII.2017.2679740.

[8] United Nations, World Urbanization “Prospects: The 2005 Revision Population Database” (Department of Economic and Social Affairs, Population Division, United Nations), NY, 2005.

[9] R. Jaiswal, R. Davidrajuh, C. Rong, “Fog Computing for Realizing Smart Neighborhoods in Smart Grids.” Computers 2020, 9, 76. https://doi.org/10.3390/computers9030076

[10] Z. Zhou, N. Chawla, Y. Jin, and G. Williams, “Big data opportunities and challenges: Discussions from data analytics perspectives,” IEEE Comput. Intell. Mag., Nov. 2014, vol. 9, no. 4, pp. 62–74.

[11] L. Da Xu, W. He, and S. Li, “Internet of things in industries: A survey,” IEEE Trans. Ind. Informat., 2014, vol. 10, no. 4, pp. 2233–2243, Nov.

[12] C. Flügel and V. Gehrmann, “Scientific workshop 4: Intelligent objects for the internet of things: Internet of things-application of sensor networks in logistics,” Commun. Comput. Inf. Sci., 2009, vol. 32, pp. 16–26.

[13] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, “Fog computing and its role in the internet of things,” in Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing, ser. MCC ’12. New York, NY, USA: ACM, 2012, pp. 13–16. [Online]. Available: http://doi.acm.org/10.1145/2342509.2342513

[14] Atlam, H.F.; Walters, R.J.; Wills, G.B. “Fog Computing and the internet of things:” A review. Big Data Cogn. Comput. 2018, 2, 10.

[15] Nature, “Big data,” 2008.
http://www.nature.com/news/specials/bigdata/index.html

[16] Z. Zhou, N. Chawla, Y. Jin, and G. Williams, “Big data opportunities and challenges: Discussions from data analytics perspectives,” *IEEE Comput. Intell. Mag.*, Nov. 2014, vol. 9, no. 4, pp. 62–74.

[17] L. Jiang, L. Da Xu, H. Cai, Z. Jiang, F. Bu, and B. Xu, “An IoT-oriented data storage framework in cloud computing platform,” *IEEE Trans. Ind. Informat.*, May 2014, vol. 10, no. 2, pp. 1443–1451.

[18] N. Mohamed, J. Al-Jaroodi, I. Jawhar, S. Lazarova-Molnar and S. Mahmoud, "SmartCityWare: A Service-Oriented Middleware for Cloud and Fog Enabled Smart City Services," in IEEE Access, 2017, vol. 5, pp. 17576-17588, doi: 10.1109/ACCESS.2017.2731382.

[19] R. Jaiswal, A. Chakravorty and C. Rong, “Distributed Fog Computing Architecture for Real-Time Anomaly Detection in Smart Meter Data,” 2020 IEEE Sixth International Conference on Big Data Computing Service and Applications (BigDataService), 2020, pp. 1-8, doi: 10.1109/BigDataService49289.2020.00009

[20] SmartGrids European Tech. Platform, Strategic Deployment Document for Europe’s Electricity Networks of the Future 6. 2010.

[21] Zhen, Z. “Smart Grid in America and Europe: Similar Desires, Different Approaches.” Public Utilities Fortnightly, 2011, 149, 1.

[22] M. Hussain, M.S. Alam, M. Beg, “Fog Computing in IoT Aided Smart Grid Transition- Requirements, Prospects, Status Quos and Challenges.” ArXiv, 2018 abs/1802.01818.

[23] Y. Zhang, T. Huang, E.F. Bompard, “Big data analytics in smart grids: a review.” Energy Inform 1, 8, 2018. https://doi.org/10.1186/s42162-018-0007-5

[24] Final Report on the 14 August 2003 Blackout in the United States and Canada: Causes and Recommendations; Technical Report; U.S.–Canada Power System Outage Task Force: Washington, DC, USA, 2004.

[25] G. Golding, A. Kumar and K. Mertens, “Cost of Texas’ 2021 Deep Freeze Justifies Weatherization” Federal Reserve Bank of Dallas. April 2021. https://www.dallasfed.org/research/economics/2021/0415.aspx (Accessed May 5, 2021)

[26] Dey, A., “Machine learning algorithms: a review.” IJCSIT, 2016, 7, pp.1174-9.

[27] S.B. Kotsiantis, “Supervised Machine Learning: A Review of Classification Techniques”, Informatica 31, 2007, 249-268

[28] Smola, A.J. and SchÃulkopf, B., “A tutorial on support vector regression.” Statistics and computing, 2004, 14(3), pp.199-222.

[29] Basak, D., Pal, S. and Patranabis, D.C., “Support vector regression.” Neural Information Processing-Letters and Reviews, 2007, 11(10), pp.203-224.

[30] Kleynhans, T., Montanaro, M., Gerace, A. and Kanan, C., “Predicting Top-of-Atmosphere Thermal Radiance Using MERRA-2 Atmospheric Data with Deep Learning.” Remote Sensing, 2017, 9(11), p.1133.

[31] Panchal, F.S. and Panchal, M., “Review on methods of selecting number of hidden nodes in artificial neural network.” International Journal of Computer Science and Mobile Computing, 2014, 3(11), pp.455-464

[32] Yılmaz, I., Yuksek, A.” An Example of Artificial Neural Network (ANN) Application for Indirect Estimation of Rock Parameters.” Rock Mech Rock Eng, 2008, 41, 781–795. https://doi.org/10.1007/s00603-007-0138-7

[33] Kustrin, S. & Beresford, R., “Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research”. Journal of pharmaceutical and biomedical analysis. 2000, 22. 717-27. 10.1016/S0731-7085(99)00272-1.

[34] Salehinejad, H.; Baarbe, J.; Sankar, S.; Barfett, J.; Colak, E.; Valaee, S. Recent “Advances in Recurrent Neural Networks. “arXiv 2017, arXiv:1801.01078.

[35] Jeffrey L. Elman. “Finding structure in time.” Cognitive Science, Mar 1990, 14(2):179 – 211.

[36] Staudemeyer, R. and Eric Rothstein Morris. “Understanding LSTM - a tutorial into Long Short-Term Memory Recurrent Neural Networks.” ArXiv abs/1909.09586, 2019

[37] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural computation, 1997, vol. 9, no. 8, pp. 1735–1780.

[38] Sepp Hochreiter and Jurgen Schmidhuber. “Long short-term memory.” Neural computation, 1997, 9(8):1735–1780

[39] F. A. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: Continual prediction with LSTM,” Neural Computation, 2000, vol. 12, no. 10, pp. 2451–2471.

[40] Center for machine learning and Intelligent Systems. “Individual household electric power consumption Data Set.” Machine Learning Repository, UCI, https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consu mption. (Accessed on: March 2, 2021).

[41] Chai, T. and Draxler, R. R.: “Root mean square error (RMSE) or mean absolute error (MAE)? – Arguments against avoiding RMSE in the literature,” Geosci. Model Dev., 2014, 7, 1247–1250, https://doi.org/10.5194/gmd-7-1247-2014.

Appendix

A. User Manual

In this project Python 3.8.3 version used and the following packages are required to run the implemented applications:

a. For preprocessing the dataframe

- pandas, numpy, math, datetime, time

b. For visualization

- pyplot from matplotlib

c. For normalization (scaling the dataframe)

- MInMaxScaler from sklearn.preprocessing

d. For the Models

- SVR from sklearn.svm
- tensorflow, keras, layers from tensorflow.keras
- Dense, RNN, SimpleRNN, LSTM from layers Sequential

e. For computing CPU and memory usages

- os, psutil

The dataset used in this work is obtained from the UCI machine learning repository.

Accessed from:
https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption

B. Source Code

The source code for this thesis is attached starting on the next page. The codes for both Fog and Cloud applications are similar.

Power_Consumption__Forecast

June 13, 2021

[3]:
```
## Importing required packages
import pandas as pd
import numpy as np
import math
from matplotlib import pyplot as plt
from sklearn.metrics import mean_squared_error
from sklearn.metrics import mean_absolute_error
from sklearn.neighbors import NearestNeighbors
from sklearn.preprocessing import MinMaxScaler
from sklearn.svm import SVR
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

from keras.layers import Dense
from keras.layers import RNN
from keras.layers import LSTM
from keras.models import Sequential
from keras.layers import GRU
from keras.layers import SimpleRNN, Dropout
import datetime
import time
from datetime import timedelta

import os
import psutil
from pandas import concat
```

0.1 Preload data and preprocess

[2]:
```
#loading the original data
data = pd.read_csv('datasets/household_power_consumption.csv', sep=';')
```

```
C:\Users\pc1\anaconda3\lib\site-packages\IPython\core\interactiveshell.py:3071:
DtypeWarning: Columns (2,3,4,5,6,7) have mixed types.Specify dtype option on
import or set low_memory=False.
  has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
[3]: # original data set
     data.head()
```

```
[3]:          Date      Time Global_active_power Global_reactive_power  Voltage  \
     0  16/12/2006  17:24:00               4.216                 0.418  234.840
     1  16/12/2006  17:25:00               5.360                 0.436  233.630
     2  16/12/2006  17:26:00               5.374                 0.498  233.290
     3  16/12/2006  17:27:00               5.388                 0.502  233.740
     4  16/12/2006  17:28:00               3.666                 0.528  235.680

       Global_intensity Sub_metering_1 Sub_metering_2  Sub_metering_3
     0           18.400          0.000          1.000            17.0
     1           23.000          0.000          1.000            16.0
     2           23.000          0.000          2.000            17.0
     3           23.000          0.000          1.000            17.0
     4           15.800          0.000          1.000            17.0
```

```
[4]: # Combining date and time
     data['date_time'] = pd.to_datetime(data['Date'] + ' ' + data['Time'] )
```

```
[5]: # replacing "?" with nan
     data = data.replace(['?'], np.nan)
     # converting 'Global_active_power' to numeric data type
     data['Global_active_power'] = pd.to_numeric(data['Global_active_power'],␣
      ↪errors='coerce')
     # replacing NaN values of Global_active_power with the mean of the column
     data['Global_active_power'].fillna(float(data['Global_active_power'].mean()),␣
      ↪inplace=True)
```

```
[6]: # extract the two columns ['date_time', 'Global_active_power']
     data = data.loc[:, ['date_time', 'Global_active_power']]
     data.sort_values('date_time', inplace=True, ascending=True)
     data = data.reset_index(drop=True)
     #set 'date_time' as an index
     data.set_index('date_time', inplace=True)
```

```
[7]: # resample the data into hours as opposed to minutes
     data = data.resample('H').sum()
```

```
[8]: data.head()
```

```
[8]:                      Global_active_power
     date_time
     2006-12-16 17:00:00              152.024
     2006-12-16 18:00:00              217.932
     2006-12-16 19:00:00              204.014
     2006-12-16 20:00:00              196.114
```

```
2006-12-16 21:00:00            183.388
```

0.1.1 Divide the data in to train and test

```
[9]: # Extract two yerars data for training
     df_train = data.loc['2007-01-01':'2008-12-31',]['Global_active_power']
     data_train = df_train.to_frame()
```

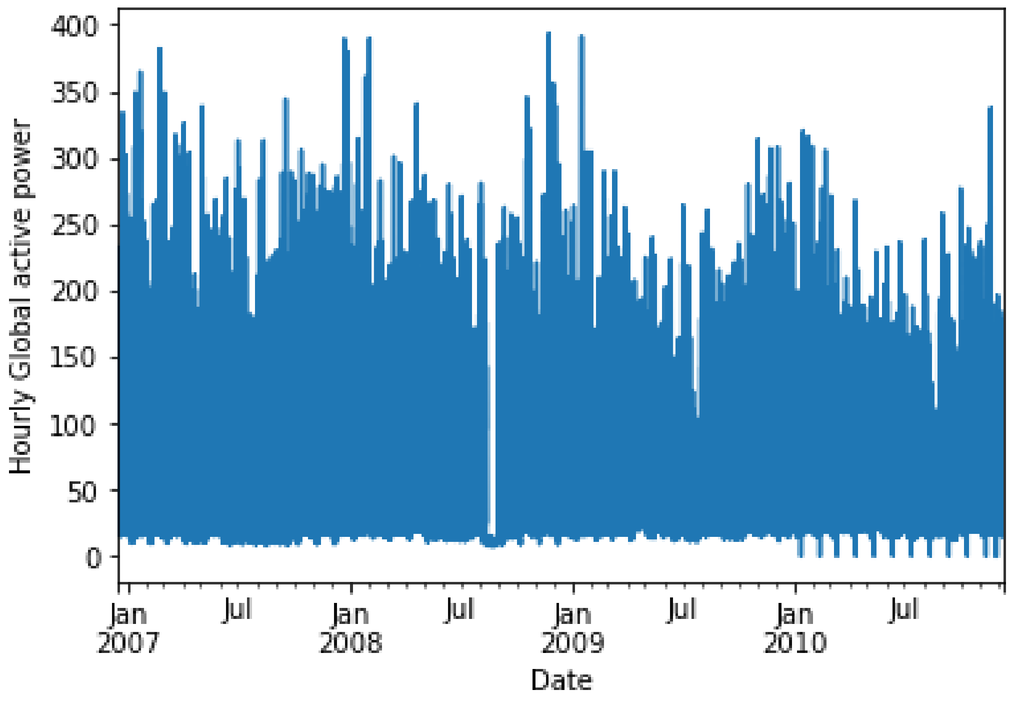
```
[10]: data_train.head()
```

```
[10]:                      Global_active_power
     date_time
     2007-01-01 00:00:00            153.038
     2007-01-01 01:00:00            151.404
     2007-01-01 02:00:00            154.940
     2007-01-01 03:00:00            152.500
     2007-01-01 04:00:00            148.544
```

```
[11]: # visualization of train data
      data['Global_active_power'].plot()

      plt.ylabel('Hourly Global active power')
      plt.xlabel('Date')
```

```
[11]: Text(0.5, 0, 'Date')
```



```
[12]: # write the training data to disk as a csv file for later use
      df_train.to_csv('datasets/cleaned_household_power_consumption_train.csv')
```

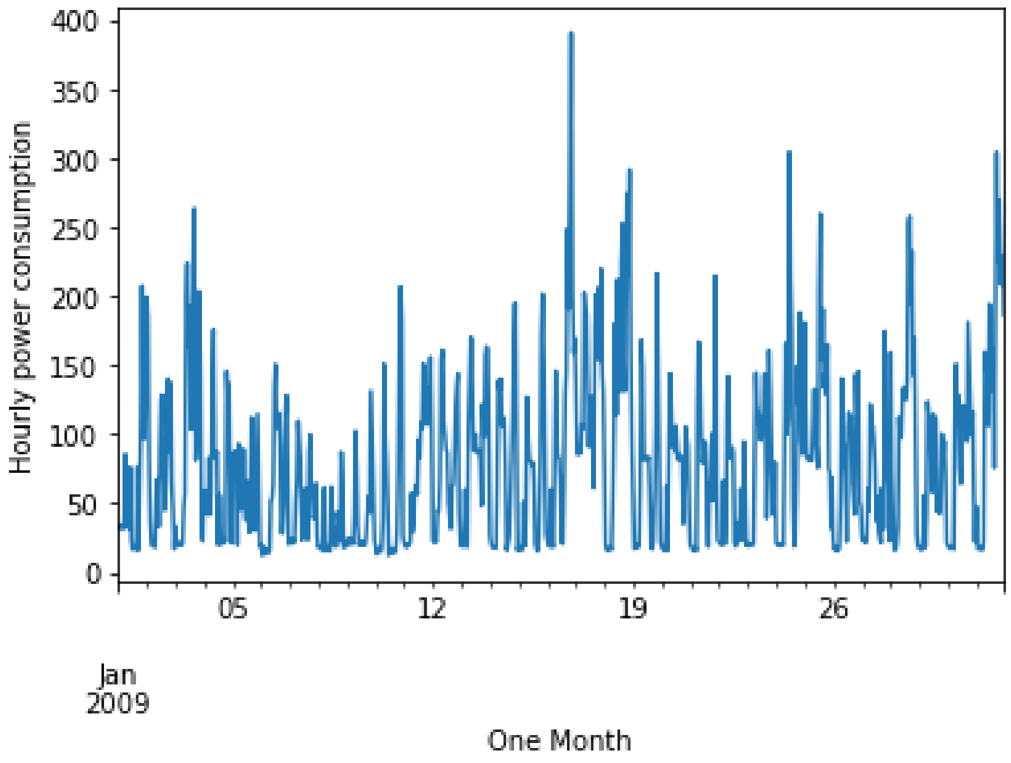
```
[13]: #Extract  one month test data
      data_test_a = data.loc['2009-01-01':'2009-01-31',]['Global_active_power']
      data_test_1 = data_test_a.to_frame()
```

```
[14]: data_test_1.head()
```

```
[14]:                      Global_active_power
      date_time
      2009-01-01 00:00:00               32.096
      2009-01-01 01:00:00               32.428
      2009-01-01 02:00:00               34.522
      2009-01-01 03:00:00               31.590
      2009-01-01 04:00:00               31.314
```

```
[15]: # visualization of 1 month test set
      data_test_1['Global_active_power'].plot()
      plt.ylabel('Hourly power consumption')
      plt.xlabel('One Month')
```

```
[15]: Text(0.5, 0, 'One Month')
```



[16]:
```
# write the one month test data to disk as a csv file for later use
data_test_1.to_csv('datasets/cleaned_household_power_consumption_data_test_1.
 ↪csv')
```

0.2 Memory and CPU Utilization Measurement tool

[4]:
```
mem_dic = {}
cpu_dic = {}

def measureit(func):
    """
    Measures a function's memory usage and running time.
    """
    pid = os.getpid()
    ps = psutil.Process(pid)

    start_mem = ps.memory_info().rss/1024**2
    start_cpu_time = ps.cpu_percent()

    def measure_mem_cpu(*args, **kw):

        result = func(*args, **kw)
        end_cpu_time = ps.cpu_percent()

        mem = ps.memory_info().rss/1024**2 - start_mem
        cpu = end_cpu_time - start_cpu_time


        dic_key = func.__qualname__
        mem_dic.update({dic_key:mem})
        cpu_dic.update({dic_key:cpu})


        print("Memory usage of %s(): %.2f MB." % (func.__qualname__, mem))
        print("Processing cpu time of %s(): %.0f seconds." % (func.
 ↪__qualname__, cpu))
        return result

    return measure_mem_cpu


def prep_cpu_mem_df():

    mem_df = pd.DataFrame(list(zip(mem_dic.keys(), mem_dic.values())),
                  columns =['Model', 'Val'])
    cpu_df = pd.DataFrame(list(zip(cpu_dic.keys(), cpu_dic.values())),
                  columns =['Model', 'Val'])

    return mem_df, cpu_df
```

0.3 Reusable Data and Functions

[5]:
```
df_train = pd.read_csv('datasets/cleaned_household_power_consumption_train.
 ↪csv', usecols=[1], engine='python')
df_train.size
```

[5]:
```
17544
```

[6]:
```
data_test_1 = pd.read_csv('datasets/
 ↪cleaned_household_power_consumption_data_test_1.csv', usecols=[1],␣
 ↪engine='python')
data_test_1.size
```

[6]:
```
744
```

[7]:
```
# appendig test dataframe on training dataframe for use in different models
df1 = df_train.append(data_test_1, ignore_index=True, sort=False)
```

```
[8]: # constants which will be used by all models

     look_back = 30
```

```
[9]: # transform dataframe using scaler
     def normalize_df(data_frame):
         dataset = scaler.fit_transform(data_frame)
         return dataset
```

```
[10]: # divide the dataframe into train and test
      def split_to_train_and_test(dataset):
          train, test = dataset[0:df_train.size:], dataset[df_train.size:len(dataset):
       ↪]
          return train, test
```

```
[11]: # convert an array of values into a dataset matrix
      def create_dataset(dataset):
          dataX, dataY = [], []
          for i in range(len(dataset)-look_back-1):
              a = dataset[i:(i+look_back), 0]
              dataX.append(a)
              dataY.append(dataset[i + look_back, 0])
          #print("Inside create dataset", dataX, dataY)
          return np.array(dataX), np.array(dataY)
```

```
[12]: # reshape input to be [samples, time steps, features]
      def reshape_input(train, test):
          trainX, trainY = create_dataset(train)
          testX, testY = create_dataset(test)
          trainX = np.reshape(trainX, (trainX.shape[0], 1, trainX.shape[1]))
          testX = np.reshape(testX, (testX.shape[0], 1, testX.shape[1]))
          print("inside reshape dataset",trainX.shape, trainY.shape, testX.shape,␣
       ↪testY.shape )
          return trainX, trainY, testX, testY
```

```
[13]: def make_predictions(model, trainX, testX):

          trainPredict = model.predict(trainX)
          testPredict = model.predict(testX)
          return trainPredict, testPredict

```

```
[14]: def invert_predictions(model, trainX, trainY, testX):
          trainPredict, testPredict = make_predictions(model, trainX, testX)

          train_predict = scaler.inverse_transform(trainPredict)
          train_y = scaler.inverse_transform([trainY])
```

```
    test_predict = scaler.inverse_transform(testPredict)

    return train_predict, train_y, test_predict
```

```
[15]: def plot_predictions(dataset, trainPredict, testPredict):
          # shift train predictions for plotting
          trainPredictPlot = np.empty_like(dataset)
          trainPredictPlot[:, :] = np.nan
          trainPredictPlot[look_back:len(trainPredict)+look_back, :] = trainPredict
          # shift test predictions for plotting
          testPredictPlot = np.empty_like(dataset)
          testPredictPlot[:, :] = np.nan
          testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1, :] =␣
       ↪testPredict

          # plot baseline and predictions
          fig= plt.figure(figsize=(15,8))
          power_changes=fig.add_subplot(1,1,1)
          power_changes.set_ylabel('Consumed power')
          power_changes.set_xlabel('Hours')

          power_changes.plot(scaler.inverse_transform(dataset), color='blue',␣
       ↪label='Real power')
          power_changes.plot(trainPredictPlot, color= 'orange', label='Train␣
       ↪Predictions')
          power_changes.plot(testPredictPlot, color= 'Green', label='Test␣
       ↪Predictions')
          power_changes.set_title('Real consumption power versus predictions by SVR␣
       ↪using timeseries data ')
          power_changes.legend(loc='best')
```

```
[16]: #dataset with one month test data
      dataset = df1
```

```
[17]: d = df1['Global_active_power'].values
```

```
[18]: # mean of the dataset
      d.mean()
```

```
[18]: 66.2211870811391
```

```
[19]: # standard devation of the dataset
      import statistics
```

```
print("Standard Deviation of df1 is % s "
      % (statistics.stdev(d)))
```

```
Standard Deviation of df1 is 57.303972419572375
```

0.3.1 SVR

[30]:
```
# split and convert to dataset matrix
train_data, test_data = split_to_train_and_test(dataset)
trainX, trainY = create_dataset(train_data.values)
testX, testY = create_dataset(test_data.values)
print("Train and test data after converting to supervised", trainX.shape,trainY.
 ↪shape, testX.shape, testY.shape )
```

```
Train and test data after converting to supervised (17519, 24) (17519,) (719,
24) (719,)
```

[31]:
```
# Scale the trainX and reshape trainY and after that normalized trainY, only␣
 ↪normalized testX
scaler_svr=MinMaxScaler(feature_range=(0, 1))
X_train_scaled = scaler_svr.fit_transform(trainX)
print("X_train_scaled shape after scaling by minmax scaler", X_train_scaled.
 ↪shape)
print("Y_train shape after scaling by minmax scaler", trainY.shape)
X_test_scaled = scaler_svr.transform(testX)
print("X_test_scaled shape after scaling by minmax scaler", X_test_scaled.shape)
# did not normalized testY
```

```
X_train_scaled shape after scaling by minmax scaler (17519, 24)
Y_train shape after scaling by minmax scaler (17519,)
X_test_scaled shape after scaling by minmax scaler (719, 24)
```

[32]:
```
# convert windows of hourly data into a series of total power
def to_series(data):
    series = np.array(data).flatten()
    return series
```

[33]:
```
@measureit
def svr_model(train_X_scaled, trainY):
    svr_rbf = SVR(kernel='rbf', C=100, gamma=0.01, epsilon=0.1)
    svr_rbf.fit(train_X_scaled, trainY)
    return svr_rbf
```

[34]:
```
# svr model and fit: memory usage and processing cpu time is varying
svr_m = svr_model(X_train_scaled,trainY)
```

```
yhat_sequence=svr_m.predict(X_test_scaled)
```

```
Memory usage of svr_model(): 5.66 MB.
Processing cpu time of svr_model(): 100 seconds.
```

```
[35]: print("predictions on X_test_scaled", yhat_sequence.shape, testY.shape)
```

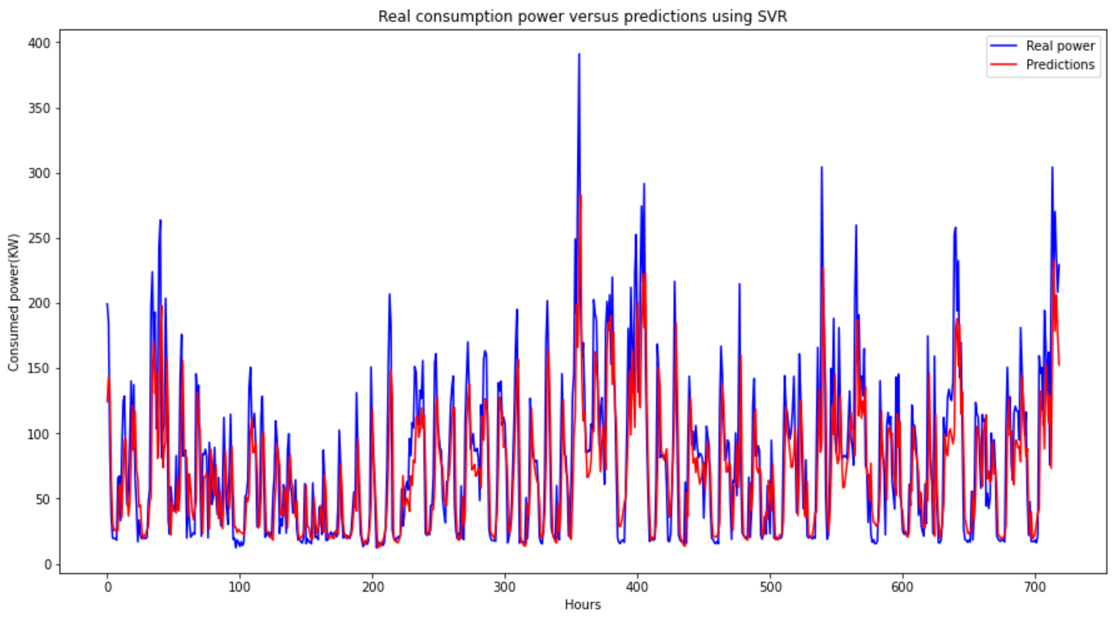
```
predictions on X_test_scaled (719,) (719,)
```

```
[36]: predictions=to_series(yhat_sequence)
      test_data=to_series(testY)
```

```
[37]: #visualization of Real power consumption  versus predictions by SVR using one␣
       ↪month test data
      fig= plt.figure(figsize=(15,8))
      power_changes=fig.add_subplot(1,1,1)
      power_changes.set_ylabel('Consumed power(KW)')
      power_changes.set_xlabel('Hours')

      power_changes.plot(test_data, color='blue', label='Real power')
      power_changes.plot(predictions, color= 'red', label='Predictions')
      power_changes.set_title('Real consumption power versus predictions using SVR')
      power_changes.legend(loc='best')
```

```
[37]: <matplotlib.legend.Legend at 0x1b698dc1ca0>
```



```
[38]: #Calculating RMSE
      def print_scores(y_test,y_pred):
          rmse= np.sqrt(mean_squared_error(y_test,y_pred))
          return rmse
      rmse=print_scores(predictions,test_data)
      print(rmse)
```

```
40.898533648082974
```

0.3.2 ANN

```
[39]: print("Train and test data after converting to supervised", trainX.shape,trainY.
       ↪shape, testX.shape, testY.shape )
```

```
Train and test data after converting to supervised (17519, 24) (17519,) (719,
24) (719,)
```

```
[40]: scaler_ann = MinMaxScaler(feature_range=(0, 1))
      X_train_scaled_ann = scaler_ann.fit_transform(trainX)
      print("X_train_scaled shape after scaling by minmax scaler", X_train_scaled_ann.
       ↪shape)
      trainY=np.reshape(trainY,(trainY.size, 1))
      Y_train_scaled_ann= scaler_ann.fit_transform(trainY)
      print("Y_train_scaled shape after scaling by minmax scaler", Y_train_scaled_ann.
       ↪shape)

      X_test_scaled_ann = scaler_ann.transform(testX)
      print("X_train_scaled shape after scaling by minmax scaler", X_test_scaled_ann.
       ↪shape)
```

```
X_train_scaled shape after scaling by minmax scaler (17519, 24)
Y_train_scaled shape after scaling by minmax scaler (17519, 1)
X_train_scaled shape after scaling by minmax scaler (719, 24)
```

```
[41]: #ANN Model

      @measureit
      def ann_model(num_neurons_1,X, y, verbose=2):
          model_ann = Sequential()
          # define number of input variables and the hidden layer
          model_ann.add(Dense(num_neurons_1, input_dim= X.shape[1],␣
       ↪activation='relu'))
          # add another layer
          model_ann.add(Dense(20, kernel_initializer='normal', activation='relu'))
          # create output layer
          model_ann.add(Dense(1, kernel_initializer='normal'))
          model_ann.compile(loss='mean_squared_error', optimizer='adam')
```

```python
    # fit the model on the training set
    model_ann.fit(X, y, epochs=50, batch_size=32, verbose=verbose)
    return model_ann



#number of neurons in the hidden layer
num_neurons_1= 10
model_ann=ann_model(num_neurons_1,X_train_scaled_ann, Y_train_scaled_ann)
```

```
Epoch 1/50
548/548 - 1s - loss: 0.0145
Epoch 2/50
548/548 - 0s - loss: 0.0099
Epoch 3/50
548/548 - 0s - loss: 0.0098
Epoch 4/50
548/548 - 0s - loss: 0.0096
Epoch 5/50
548/548 - 0s - loss: 0.0096
Epoch 6/50
548/548 - 0s - loss: 0.0096
Epoch 7/50
548/548 - 0s - loss: 0.0095
Epoch 8/50
548/548 - 0s - loss: 0.0095
Epoch 9/50
548/548 - 0s - loss: 0.0095
Epoch 10/50
548/548 - 0s - loss: 0.0095
Epoch 11/50
548/548 - 0s - loss: 0.0095
Epoch 12/50
548/548 - 0s - loss: 0.0095
Epoch 13/50
548/548 - 0s - loss: 0.0095
Epoch 14/50
548/548 - 0s - loss: 0.0094
Epoch 15/50
548/548 - 0s - loss: 0.0094
Epoch 16/50
548/548 - 0s - loss: 0.0094
Epoch 17/50
548/548 - 0s - loss: 0.0094
Epoch 18/50
548/548 - 0s - loss: 0.0094
Epoch 19/50
```

```
548/548 - 0s - loss: 0.0094
Epoch 20/50
548/548 - 0s - loss: 0.0093
Epoch 21/50
548/548 - 0s - loss: 0.0093
Epoch 22/50
548/548 - 0s - loss: 0.0093
Epoch 23/50
548/548 - 0s - loss: 0.0093
Epoch 24/50
548/548 - 0s - loss: 0.0093
Epoch 25/50
548/548 - 0s - loss: 0.0093
Epoch 26/50
548/548 - 0s - loss: 0.0093
Epoch 27/50
548/548 - 0s - loss: 0.0093
Epoch 28/50
548/548 - 0s - loss: 0.0093
Epoch 29/50
548/548 - 0s - loss: 0.0093
Epoch 30/50
548/548 - 0s - loss: 0.0092
Epoch 31/50
548/548 - 0s - loss: 0.0092
Epoch 32/50
548/548 - 0s - loss: 0.0093
Epoch 33/50
548/548 - 0s - loss: 0.0092
Epoch 34/50
548/548 - 0s - loss: 0.0092
Epoch 35/50
548/548 - 0s - loss: 0.0092
Epoch 36/50
548/548 - 0s - loss: 0.0092
Epoch 37/50
548/548 - 0s - loss: 0.0092
Epoch 38/50
548/548 - 0s - loss: 0.0092
Epoch 39/50
548/548 - 0s - loss: 0.0092
Epoch 40/50
548/548 - 0s - loss: 0.0092
Epoch 41/50
548/548 - 0s - loss: 0.0091
Epoch 42/50
548/548 - 0s - loss: 0.0092
Epoch 43/50
```

```
548/548 - 0s - loss: 0.0092
Epoch 44/50
548/548 - 0s - loss: 0.0091
Epoch 45/50
548/548 - 0s - loss: 0.0091
Epoch 46/50
548/548 - 0s - loss: 0.0091
Epoch 47/50
548/548 - 0s - loss: 0.0091
Epoch 48/50
548/548 - 0s - loss: 0.0091
Epoch 49/50
548/548 - 0s - loss: 0.0091
Epoch 50/50
548/548 - 0s - loss: 0.0091
Memory usage of ann_model(): 25.78 MB.
Processing cpu time of ann_model(): 171 seconds.
```

[42]:
```python
testPredict=model_ann.predict(X_test_scaled)
print(testPredict.shape) #printed scaled predicted values
```

```
(719, 1)
```

1 Invert predictions and test dataset

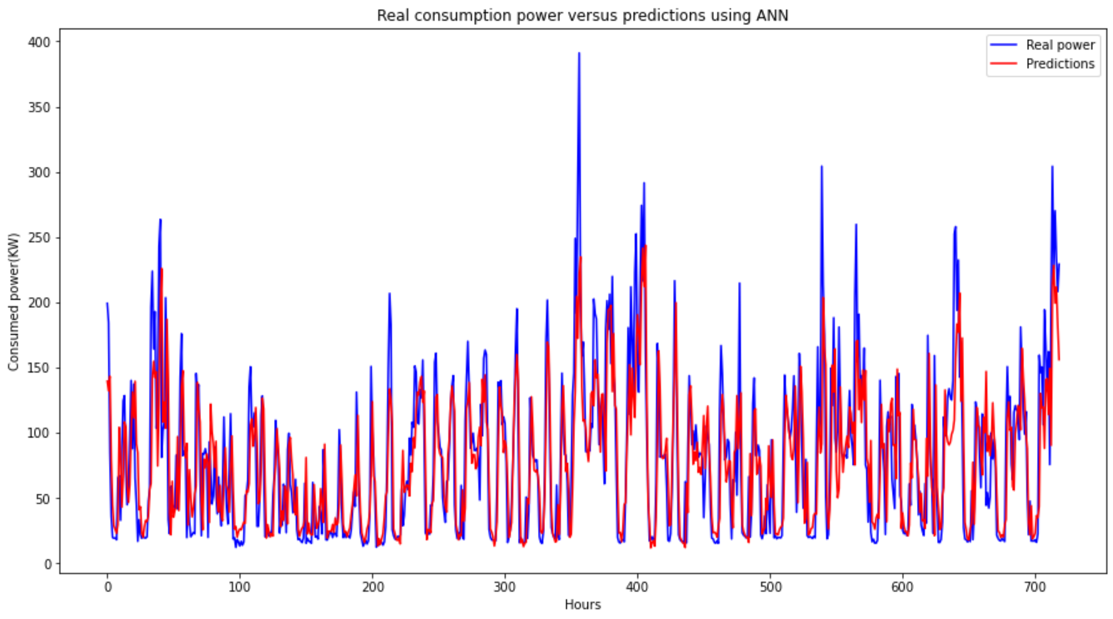
[43]:
```python
inv_yhat = scaler_ann.inverse_transform(testPredict)

predictions=to_series(inv_yhat)
test_data=to_series(testY)
```

[44]:
```python
#visualization of Real power consumption  versus predictions by ANN using one␣
↪month test data
fig= plt.figure(figsize=(15,8))
power_changes=fig.add_subplot(1,1,1)
power_changes.set_ylabel('Consumed power(KW)')
power_changes.set_xlabel('Hours')

power_changes.plot(test_data, color='blue', label='Real power')
power_changes.plot(predictions, color= 'red', label='Predictions')
power_changes.set_title('Real consumption power versus predictions using ANN')
power_changes.legend(loc='best')
```

[44]:
```
<matplotlib.legend.Legend at 0x1b6a8d7d040>
```



[45]:
```
#Calculating RMSE
def print_scores(y_test,y_pred):
    rmse= np.sqrt(mean_squared_error(y_test,y_pred))
    return rmse
rmse=print_scores(predictions,test_data)
print(rmse)
```

```
39.502684241815885
```

1.0.1 RNN

[83]:
```
scaler_rnn = MinMaxScaler(feature_range=(0, 1))
X_train_scaled_rnn = scaler_rnn.fit_transform(trainX)
print("X_train_scaled shape after scaling by minmax scaler", X_train_scaled_rnn.
 ↪shape)

Y_train_scaled_rnn= scaler_rnn.fit_transform(trainY)
print("Y_train_scaled shape after scaling by minmax scaler", Y_train_scaled_rnn.
 ↪shape)

X_test_scaled_rnn = scaler_rnn.transform(testX)
print("X_train_scaled shape after scaling by minmax scaler", X_test_scaled_rnn.
 ↪shape)

trainX_re = np.reshape(X_train_scaled_rnn, (X_train_scaled_rnn.shape[0], 1,␣
 ↪X_train_scaled_rnn.shape[1]))
```

```
testX_re = np.reshape(X_test_scaled_rnn, (X_test_scaled_rnn.shape[0], 1,␣
↪X_test_scaled_rnn.shape[1]))

print(trainX_re.shape, testX_re.shape)
```

```
X_train_scaled shape after scaling by minmax scaler (17519, 24)
Y_train_scaled shape after scaling by minmax scaler (17519, 1)
X_train_scaled shape after scaling by minmax scaler (719, 24)
(17519, 1, 24) (719, 1, 24)
```

[84]:
```
# SimpleRNN model
@measureit
def rnn_model():
    model = Sequential()
    model.add(SimpleRNN(units=32, input_shape=(1,look_back), activation="relu"))
    model.add(Dense(8, activation="relu"))
    model.add(Dense(1))
    model.compile(loss='mean_squared_error', optimizer='Adam')

    model.fit(trainX_re,Y_train_scaled_rnn, epochs=100, batch_size=32,␣
↪verbose=2)
    return model
```

[85]:
```
rnn_mod = rnn_model()
```

```
Epoch 1/100
548/548 - 1s - loss: 0.0176
Epoch 2/100
548/548 - 0s - loss: 0.0100
Epoch 3/100
548/548 - 0s - loss: 0.0097
Epoch 4/100
548/548 - 0s - loss: 0.0096
Epoch 5/100
548/548 - 0s - loss: 0.0095
Epoch 6/100
548/548 - 0s - loss: 0.0094
Epoch 7/100
548/548 - 0s - loss: 0.0094
Epoch 8/100
548/548 - 0s - loss: 0.0094
Epoch 9/100
548/548 - 0s - loss: 0.0093
Epoch 10/100
548/548 - 1s - loss: 0.0093
Epoch 11/100
548/548 - 0s - loss: 0.0093
```

```
Epoch 12/100
548/548 - 0s - loss: 0.0092
Epoch 13/100
548/548 - 0s - loss: 0.0092
Epoch 14/100
548/548 - 0s - loss: 0.0092
Epoch 15/100
548/548 - 0s - loss: 0.0091
Epoch 16/100
548/548 - 0s - loss: 0.0091
Epoch 17/100
548/548 - 0s - loss: 0.0091
Epoch 18/100
548/548 - 0s - loss: 0.0091
Epoch 19/100
548/548 - 0s - loss: 0.0091
Epoch 20/100
548/548 - 0s - loss: 0.0091
Epoch 21/100
548/548 - 0s - loss: 0.0090
Epoch 22/100
548/548 - 0s - loss: 0.0090
Epoch 23/100
548/548 - 0s - loss: 0.0090
Epoch 24/100
548/548 - 0s - loss: 0.0090
Epoch 25/100
548/548 - 0s - loss: 0.0089
Epoch 26/100
548/548 - 0s - loss: 0.0090
Epoch 27/100
548/548 - 0s - loss: 0.0090
Epoch 28/100
548/548 - 0s - loss: 0.0089
Epoch 29/100
548/548 - 0s - loss: 0.0089
Epoch 30/100
548/548 - 0s - loss: 0.0089
Epoch 31/100
548/548 - 0s - loss: 0.0089
Epoch 32/100
548/548 - 0s - loss: 0.0089
Epoch 33/100
548/548 - 0s - loss: 0.0089
Epoch 34/100
548/548 - 0s - loss: 0.0088
Epoch 35/100
548/548 - 0s - loss: 0.0089
```

```
Epoch 36/100
548/548 - 0s - loss: 0.0089
Epoch 37/100
548/548 - 0s - loss: 0.0088
Epoch 38/100
548/548 - 0s - loss: 0.0088
Epoch 39/100
548/548 - 0s - loss: 0.0088
Epoch 40/100
548/548 - 0s - loss: 0.0088
Epoch 41/100
548/548 - 0s - loss: 0.0088
Epoch 42/100
548/548 - 0s - loss: 0.0088
Epoch 43/100
548/548 - 0s - loss: 0.0087
Epoch 44/100
548/548 - 0s - loss: 0.0088
Epoch 45/100
548/548 - 0s - loss: 0.0087
Epoch 46/100
548/548 - 0s - loss: 0.0087
Epoch 47/100
548/548 - 0s - loss: 0.0087
Epoch 48/100
548/548 - 0s - loss: 0.0087
Epoch 49/100
548/548 - 0s - loss: 0.0087
Epoch 50/100
548/548 - 0s - loss: 0.0087
Epoch 51/100
548/548 - 0s - loss: 0.0087
Epoch 52/100
548/548 - 0s - loss: 0.0087
Epoch 53/100
548/548 - 0s - loss: 0.0087
Epoch 54/100
548/548 - 0s - loss: 0.0087
Epoch 55/100
548/548 - 0s - loss: 0.0087
Epoch 56/100
548/548 - 0s - loss: 0.0086
Epoch 57/100
548/548 - 1s - loss: 0.0087
Epoch 58/100
548/548 - 0s - loss: 0.0086
Epoch 59/100
548/548 - 0s - loss: 0.0086
```

```
Epoch 60/100
548/548 - 0s - loss: 0.0086
Epoch 61/100
548/548 - 0s - loss: 0.0087
Epoch 62/100
548/548 - 0s - loss: 0.0086
Epoch 63/100
548/548 - 0s - loss: 0.0086
Epoch 64/100
548/548 - 0s - loss: 0.0086
Epoch 65/100
548/548 - 0s - loss: 0.0086
Epoch 66/100
548/548 - 0s - loss: 0.0086
Epoch 67/100
548/548 - 0s - loss: 0.0086
Epoch 68/100
548/548 - 0s - loss: 0.0086
Epoch 69/100
548/548 - 0s - loss: 0.0086
Epoch 70/100
548/548 - 0s - loss: 0.0086
Epoch 71/100
548/548 - 0s - loss: 0.0086
Epoch 72/100
548/548 - 0s - loss: 0.0086
Epoch 73/100
548/548 - 0s - loss: 0.0085
Epoch 74/100
548/548 - 0s - loss: 0.0086
Epoch 75/100
548/548 - 0s - loss: 0.0086
Epoch 76/100
548/548 - 0s - loss: 0.0086
Epoch 77/100
548/548 - 0s - loss: 0.0085
Epoch 78/100
548/548 - 0s - loss: 0.0085
Epoch 79/100
548/548 - 0s - loss: 0.0086
Epoch 80/100
548/548 - 0s - loss: 0.0085
Epoch 81/100
548/548 - 0s - loss: 0.0085
Epoch 82/100
548/548 - 0s - loss: 0.0085
Epoch 83/100
548/548 - 0s - loss: 0.0085
```

```
Epoch 84/100
548/548 - 0s - loss: 0.0085
Epoch 85/100
548/548 - 0s - loss: 0.0085
Epoch 86/100
548/548 - 0s - loss: 0.0085
Epoch 87/100
548/548 - 0s - loss: 0.0085
Epoch 88/100
548/548 - 0s - loss: 0.0085
Epoch 89/100
548/548 - 0s - loss: 0.0085
Epoch 90/100
548/548 - 0s - loss: 0.0085
Epoch 91/100
548/548 - 1s - loss: 0.0085
Epoch 92/100
548/548 - 1s - loss: 0.0085
Epoch 93/100
548/548 - 0s - loss: 0.0085
Epoch 94/100
548/548 - 0s - loss: 0.0085
Epoch 95/100
548/548 - 0s - loss: 0.0085
Epoch 96/100
548/548 - 0s - loss: 0.0084
Epoch 97/100
548/548 - 0s - loss: 0.0084
Epoch 98/100
548/548 - 0s - loss: 0.0085
Epoch 99/100
548/548 - 0s - loss: 0.0085
Epoch 100/100
548/548 - 0s - loss: 0.0084
Memory usage of rnn_model(): 9.66 MB.
Processing cpu time of rnn_model(): 213 seconds.
```

```
[49]: testPredict=rnn_mod.predict(testX_re)
      print(testPredict.shape) #printed scaled predicted values
```

```
(719, 1)
```

```
[50]: trainScore = rnn_mod.evaluate(trainX_re, Y_train_scaled_rnn, verbose=0)
      print(trainScore)
```

```
0.0080118328332901
```

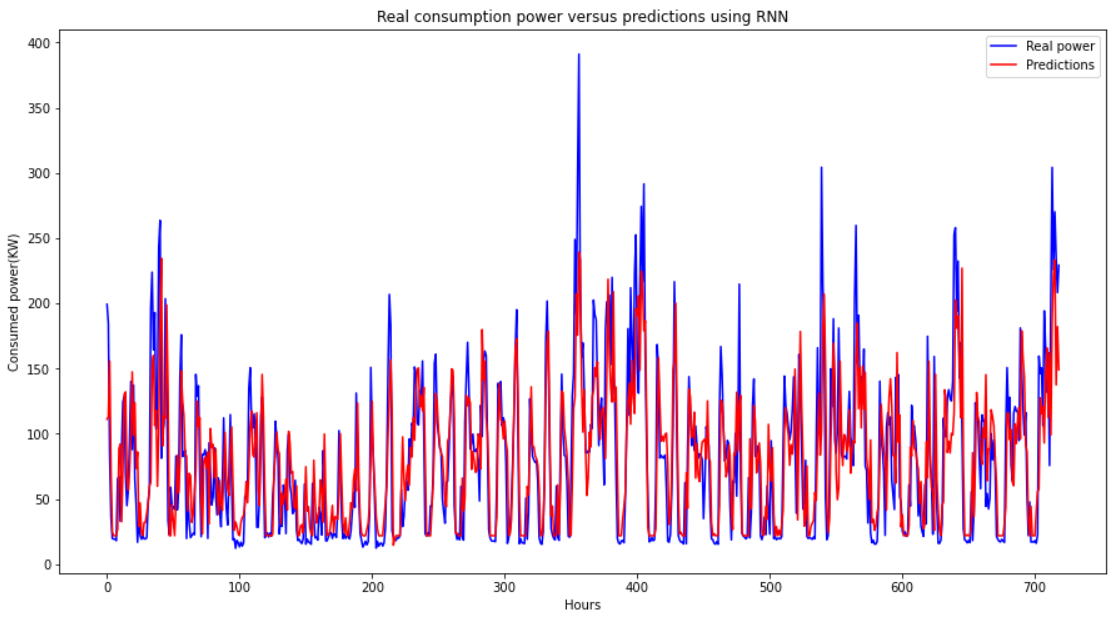
```
[51]: inv_yhat = scaler_rnn.inverse_transform(testPredict)

      predictions_rnn=to_series(inv_yhat)
      test_data_rnn=to_series(testY)
```

```
[52]: #visualization of Real power consumption  versus predictions by RNN using one␣
       ↪month test data
      fig= plt.figure(figsize=(15,8))
      power_changes=fig.add_subplot(1,1,1)
      power_changes.set_ylabel('Consumed power(KW)')
      power_changes.set_xlabel('Hours')

      power_changes.plot(test_data_rnn, color='blue', label='Real power')
      power_changes.plot(predictions_rnn, color= 'red', label='Predictions')
      power_changes.set_title('Real consumption power versus predictions using RNN')
      power_changes.legend(loc='best')
```

```
[52]: <matplotlib.legend.Legend at 0x1b6a9d869a0>
```



```
[53]: #Calculating RMSE
      def print_scores(y_test,y_pred):
          rmse= np.sqrt(mean_squared_error(y_test,y_pred))
          return rmse
      rmse=print_scores(predictions_rnn,test_data_rnn)
      print(rmse)
```

```
39.5803241095926
```

1.0.2 LSTM

[75]:
```
scaler_lstm = MinMaxScaler(feature_range=(0, 1))
X_train_scaled_lstm = scaler_lstm.fit_transform(trainX)
print("X_train_scaled shape after scaling by minmax scaler",␣
 ↪X_train_scaled_lstm.shape)

Y_train_scaled_lstm= scaler_lstm.fit_transform(trainY)
print("Y_train_scaled shape after scaling by minmax scaler",␣
 ↪Y_train_scaled_lstm.shape)

X_test_scaled_lstm = scaler_lstm.transform(testX)
print("X_train_scaled shape after scaling by minmax scaler", X_test_scaled_lstm.
 ↪shape)

trainX_re_lstm = np.reshape(X_train_scaled_lstm, (X_train_scaled_lstm.shape[0],␣
 ↪1, X_train_scaled_lstm.shape[1]))
testX_re_lstm = np.reshape(X_test_scaled_lstm, (X_test_scaled_lstm.shape[0], 1,␣
 ↪X_test_scaled_lstm.shape[1]))

print(trainX_re_lstm.shape, testX_re_lstm.shape)
```

```
X_train_scaled shape after scaling by minmax scaler (17519, 24)
Y_train_scaled shape after scaling by minmax scaler (17519, 1)
X_train_scaled shape after scaling by minmax scaler (719, 24)
(17519, 1, 24) (719, 1, 24)
```

[76]:
```
@measureit
def lstm_model(X, Y, verbose=2):
    batch_size= 32
    model = Sequential()
    model.add(LSTM(216, activation='relu', input_shape=(1, look_back )))
    model.add(Dense(1))
    optimizer = keras.optimizers.Adam(lr=0.001, clipvalue=0.5)
    model.compile(loss="mse", optimizer=optimizer)
    model.fit(X, Y, epochs= 50, batch_size= batch_size, verbose=verbose)
    return model
```

[77]:
```
model_lstm=lstm_model(trainX_re_lstm, Y_train_scaled_lstm)
```

```
Epoch 1/50
548/548 - 2s - loss: 0.0114
Epoch 2/50
548/548 - 1s - loss: 0.0097
Epoch 3/50
548/548 - 1s - loss: 0.0094
Epoch 4/50
548/548 - 1s - loss: 0.0093
```

```
Epoch 5/50
548/548 - 1s - loss: 0.0093
Epoch 6/50
548/548 - 1s - loss: 0.0093
Epoch 7/50
548/548 - 1s - loss: 0.0092
Epoch 8/50
548/548 - 1s - loss: 0.0092
Epoch 9/50
548/548 - 1s - loss: 0.0092
Epoch 10/50
548/548 - 1s - loss: 0.0091
Epoch 11/50
548/548 - 1s - loss: 0.0091
Epoch 12/50
548/548 - 1s - loss: 0.0091
Epoch 13/50
548/548 - 1s - loss: 0.0090
Epoch 14/50
548/548 - 1s - loss: 0.0090
Epoch 15/50
548/548 - 1s - loss: 0.0090
Epoch 16/50
548/548 - 1s - loss: 0.0090
Epoch 17/50
548/548 - 1s - loss: 0.0089
Epoch 18/50
548/548 - 1s - loss: 0.0089
Epoch 19/50
548/548 - 1s - loss: 0.0089
Epoch 20/50
548/548 - 1s - loss: 0.0088
Epoch 21/50
548/548 - 1s - loss: 0.0088
Epoch 22/50
548/548 - 1s - loss: 0.0088
Epoch 23/50
548/548 - 1s - loss: 0.0087
Epoch 24/50
548/548 - 1s - loss: 0.0088
Epoch 25/50
548/548 - 1s - loss: 0.0087
Epoch 26/50
548/548 - 1s - loss: 0.0087
Epoch 27/50
548/548 - 1s - loss: 0.0087
Epoch 28/50
548/548 - 1s - loss: 0.0086
```

```
Epoch 29/50
548/548 - 1s - loss: 0.0086
Epoch 30/50
548/548 - 1s - loss: 0.0085
Epoch 31/50
548/548 - 1s - loss: 0.0085
Epoch 32/50
548/548 - 1s - loss: 0.0085
Epoch 33/50
548/548 - 1s - loss: 0.0085
Epoch 34/50
548/548 - 1s - loss: 0.0084
Epoch 35/50
548/548 - 1s - loss: 0.0084
Epoch 36/50
548/548 - 1s - loss: 0.0084
Epoch 37/50
548/548 - 1s - loss: 0.0083
Epoch 38/50
548/548 - 1s - loss: 0.0082
Epoch 39/50
548/548 - 1s - loss: 0.0082
Epoch 40/50
548/548 - 1s - loss: 0.0082
Epoch 41/50
548/548 - 1s - loss: 0.0082
Epoch 42/50
548/548 - 1s - loss: 0.0081
Epoch 43/50
548/548 - 1s - loss: 0.0081
Epoch 44/50
548/548 - 1s - loss: 0.0081
Epoch 45/50
548/548 - 1s - loss: 0.0080
Epoch 46/50
548/548 - 1s - loss: 0.0080
Epoch 47/50
548/548 - 1s - loss: 0.0080
Epoch 48/50
548/548 - 1s - loss: 0.0079
Epoch 49/50
548/548 - 1s - loss: 0.0079
Epoch 50/50
548/548 - 1s - loss: 0.0079
Memory usage of lstm_model(): 19.27 MB.
Processing cpu time of lstm_model(): 278 seconds.
```

[57]:
```
testPredict_lstm= model_lstm.predict(testX_re_lstm)
```

[58]:
```
inv_yhat = scaler_lstm.inverse_transform(testPredict_lstm)

predictions_lstm=to_series(inv_yhat)
test_data_lstm=to_series(testY)
```

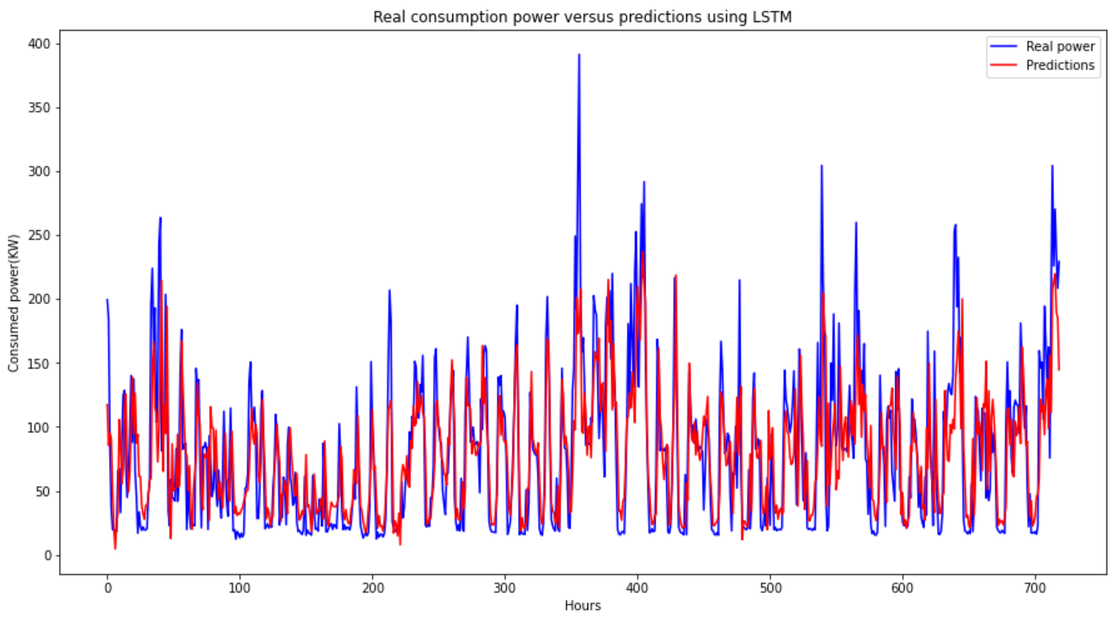
[59]:
```
#Calculating RMSE
def print_scores(y_test,y_pred):
    rmse= np.sqrt(mean_squared_error(y_test,y_pred))
    return rmse
rmse=print_scores(predictions_lstm,test_data_lstm)
print(rmse)
```

```
40.80439774947916
```

[60]:
```
#visualization of Real power consumption  versus predictions by LSTM using one␣
 ↪month test data
fig= plt.figure(figsize=(15,8))
power_changes=fig.add_subplot(1,1,1)
power_changes.set_ylabel('Consumed power(KW)')
power_changes.set_xlabel('Hours')

power_changes.plot(test_data_lstm, color='blue', label='Real power')
power_changes.plot(predictions_lstm, color= 'red', label='Predictions')
power_changes.set_title('Real consumption power versus predictions using LSTM')
power_changes.legend(loc='best')
```

[60]:
```
<matplotlib.legend.Legend at 0x1b6ab24cfa0>
```

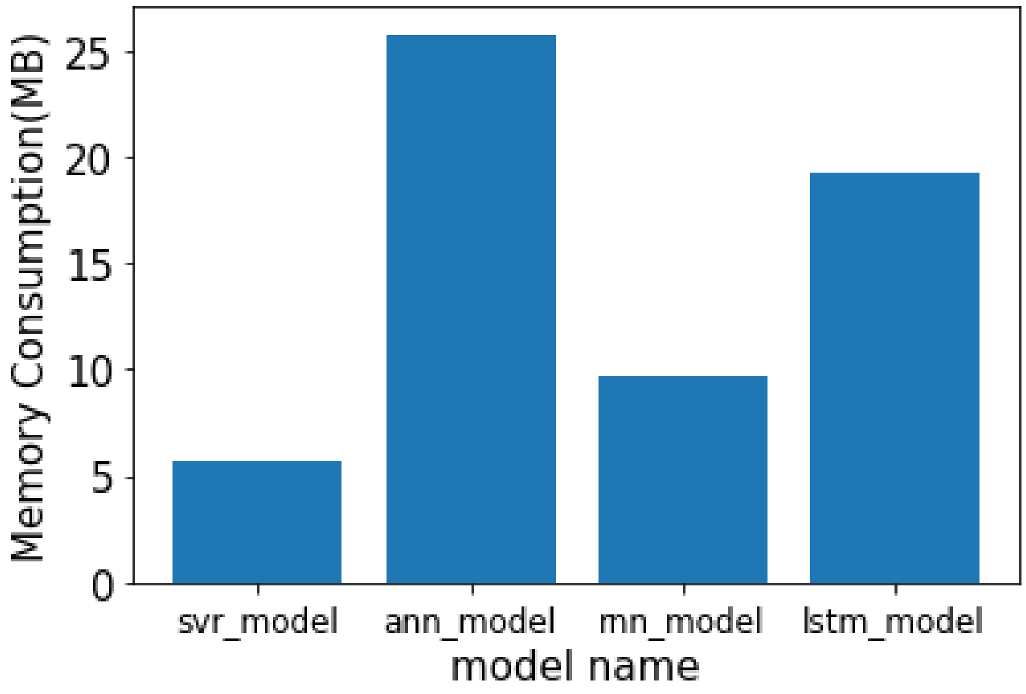


1.1 Memory and CPU utilization visualization

```
[86]: mem_df, cpu_df = prep_cpu_mem_df()
```

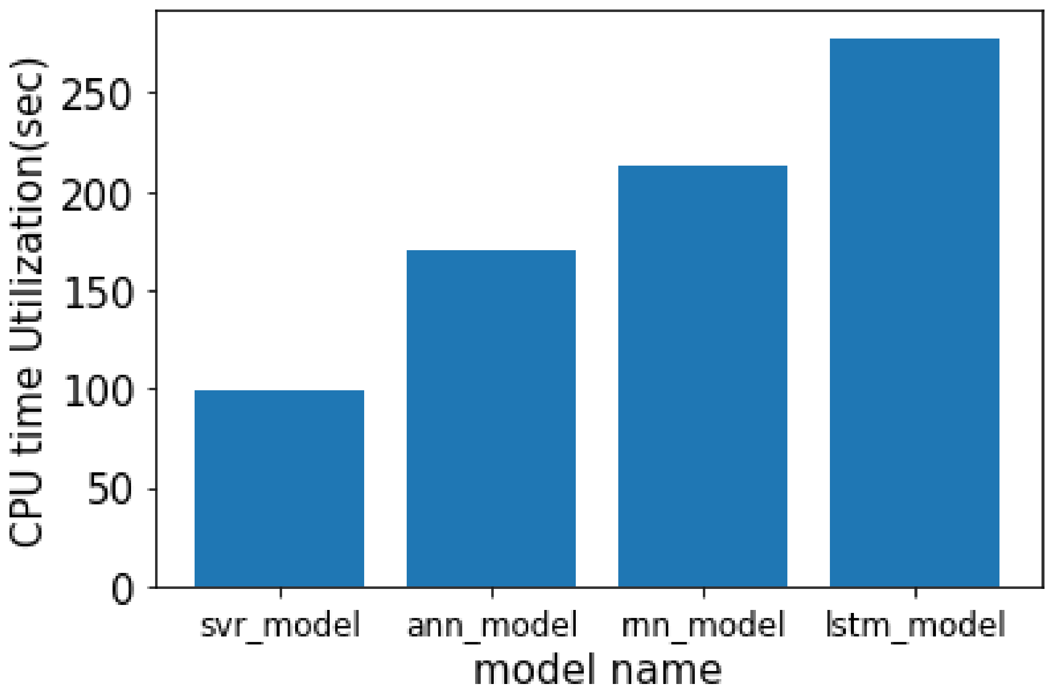
```
[87]: # memory usage visualization of a model
      plt.ylabel('Memory Consumption(MB)',fontsize=15)
      plt.xlabel('model name',fontsize=15)
      plt.xticks(fontsize=12)
      plt.yticks(fontsize=15)
      plt.bar(mem_df.Model, mem_df.Val)
```

```
[87]: <BarContainer object of 4 artists>
```



```
[88]: # cpu utilization visualization of a model
      plt.ylabel('CPU time Utilization(sec)',fontsize=15)
      plt.xlabel('model name',fontsize=15)
      plt.xticks(fontsize=12)
      plt.yticks(fontsize=15)
      plt.bar(cpu_df.Model, cpu_df.Val)
```

```
[88]: <BarContainer object of 4 artists>
```


CPU time Utilization(sec)
250
200
150
100
50
0
svr_model
ann_model
rnn_model
lstm_model
model name