University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Data science	Spring semester, 2022. Open / Restricted access
Writer: Martin Sommerli	Martin Sommerli ..... (Writer's signature)
Faculty supervisor: Reggie Davidrajuh, Rituka Jaiswal External supervisor(s):	
Thesis title: Anomaly Detection in Smart Meter Data using BiLSTMGAN and Performance Analysis in Fog Network	
Credits (ECTS): 30	
Key words: Smart Grid, anomaly detection, power consumption, LSTM, GAN, machine learning, Fog network, Fog architecture, performance analysis	Pages: 60 + enclosure: ..... Stavanger, 15/06 - 2022 Date/year

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

Anomaly Detection in Smart Meter Data using BiLSTMGAN and Performance Analysis in Fog Network

Master's Thesis in Computer Science

by

Martin Sommerli

Supervisors

Reggie Davidrajuh

Rituka Jaiswal

June 15, 2022

"Det er i motbakke det går oppover."
"It is in difficult times that progress is made"

Rune Gokstad

Abstract

In recent years, the introduction of Smart Grids has provided us with access to a new layer of energy consumption patterns. This new layer of consumption could be analyzed to boost efficiency, prevent power loss, and generate significant economic and environmental benefits. For the analysis of this data layer, a relatively new paradigm has emerged: fog computing or fog networking. Fog computing seeks to offload a cloud computing architecture by decentralizing data processing from the typical cloud computing platform to edge nodes that have less computing power and are closer to the data source.

In this study, we intend to implement an anomaly detection algorithm on Smart grid data using a model that employs a generative adversarial network and long short term memory to classify anomalies in data from smart grid customers. This algorithm will be evaluated on multiple platforms, including Edge devices and Cloud virtual machines, and run-time metrics will be collected for comparison purposes.

All code used is available at Github.

Acknowledgements

First, I would like to genuinely thank both my supervisors, Prof. Reggie Davidrajuh and Rituka Jaiswal, for the support and help over the past six months. The achievements and process of this paper has benefited greatly from their passion of the field.

In addition, I would like to thank my family and my partner for supporting me throughout the years of studying, giving me the courage and driving force to pursuit and complete a masters degree.

Finally, I want to thank my fellow students that I met over the years, which has provided a lot of help and answers to the different endeavours that has occurred.

Contents

Abbreviations

Acronym	What (it) Stands For
GAN	**G**enerative **A**dverserial **N**etwork
LSTM	**L**ong **S**hort **T**erm **M**emory
IoT	**I**nternet **o**f **T**hings
RNN	**R**ecurrent **N**eural **N**etwork
ANN	**A**rtificial **N**eural **N**etwork
TP	**T**rue **P**ositive
TN	**T**rue **N**egative
FP	**F**alse **P**ositive
FN	**F**alse **N**egative
VM	**V**irtual **M**achine
GC	**G**eneral **C**onsumption
GG	**G**ross **G**eneration
CL	**C**ontrolled **L**oad
RAM	**R**andom **A**ccess **M**emory
CPU	**C**entral **P**rocessing **U**nit
GPU	**G**rapichs **P**rocessing **U**nit

Chapter 1

Introduction

1.1 Background and Motivation

1.1.1 Smart Grid and the need for real-time decision making

Smart grid is an improved electrical network architecture that employs sophisticated monitoring and information transmission technologies to boost the power grid's efficiency, dependability, and safety.[1, 2] Additionally, it permits the capture, transfer, and storage of energy usage data in real-time.[3] The smart grid is essential for economic development, energy structure modification, and adaptation to climate change, all of which can result in energy savings and emission reductions. [4]

In the global power system of the twenty-first century, detecting and preventing power losses have become increasingly important. Each year, hundreds of millions of dollars are lost due to illicit electricity usage by consumers in the power businesses of many nations.[5] Monitoring and forecasting of power consumption is crucial for power providers in terms of power generation, scheduling, and dispatching due to the expansion and popularity of the Internet of Things and the complexity of demand and supply.[1] It benefits energy customers by enabling them to optimize their usage patterns, hence reducing their expenses. In addition, energy suppliers can identify erroneous meter readings resulting from unanticipated meter failures, deliberate meter manipulations, or consumers' atypical usage patterns.

1.1.2 Benefits of anomaly detection

Anomaly detection is the process of detecting patterns in a given data collection that deviate from the behavior that would be expected.[6] Anomaly detection is the process

of identifying events that occur relatively infrequently, which has been widely utilized in a wide range of applications, such as fraud detection in banking, insurance, and health care, intrusion interception in cyber-security, and fault detection for safety-critical systems, among others.[6] Due to the widespread installation of smart meters, smart meter analytics draws a rising amount of study.[7–9] Anomaly detection may be used to evaluate live smart meter data in order to assist energy users in identifying odd behaviors, such as forgetting to switch off the stove after cooking, and assist utilities in identifying abnormal events, such as electricity leaking and theft. Since abnormal consumption could also result from user activities, such as using ineffective electrical items, or over-lighting and working overtime in office spaces, anomalous feedback can be used to warn consumers to lessen energy usage and to help them identify inefficient appliances.[10, 11] Consequently, real-time monitoring and abnormal detection of meter data in the power grid can detect abnormalities or accidents in the power grid in a timely manner and prevent more severe power loss and equipment damage resulting from a failure to respond to failures in a timely manner.[12] Anomaly detection on smart grid data provides significant economic and safety benefits for the smart grid. In data analysis, anomaly detection has long been a popular topic. Traditional anomaly detection approaches are often subdivided into statistically-based, density-based, cluster-based, classification-based, and spectral decomposition-based categories. Nonetheless, the power grid equipment data are time-series data with temporal correlation, therefore the anomaly of the time series must be determined based on the context information; the conventional approach for detecting anomalies is inapplicable. Current anomaly detection algorithms for time series can be categorized as time series decomposition, classification and regression trees, ARIMA, exponential smoothing, and neural networks. [13]

The application of Generative Adversarial Networks(GAN) to describe the complicated and high-dimensional distribution of real-world data has been a success. This property of GANs implies that they can be utilized well for anomaly identification.[14, 15] Anomaly detection using GANs is the problem of modeling normal behavior using adversarial training and finding abnormalities by calculating an anomaly score. The GAN framework learns a generator that creates samples out of a random latent distribution of data. The framework also trains a discriminator which seeks to distinguish between authentic and generated datasets. We present a conditional Long Short Term Memory Generative Adversarial Netowrk(LSTM-GAN) model for composing smart grid data, where a discriminator can aid in ensuring that created data keep the same distribution as actual consumption.

1.1.3 Fog Computing in Smart Grid

When doing anomaly detection on Smart Grid data, it is essential that information travels quickly while protecting user privacy. To address the difficulties of high-bandwidth, geographically scattered, ultra-low latency, and privacy-sensitive applications, a computing paradigm that occurs closer to connected devices is necessary.[16, 17] Industry and academics have proposed fog computing to overcome the aforementioned concerns and satisfy the requirement for a computer paradigm closer to linked devices.[18]

Fog computing creates a link between the cloud and IoT(Internet of Things) devices by providing computation, storage, networking, and data management on IoT-proximate network nodes. As a result, computing, storage, networking, decision making, and data management occur along the path between IoT devices and the cloud as IoT device data is transferred to the cloud. The scientific community has proposed several computing paradigms comparable to fog computing, such as edge computing, mist computing, cloud of things, and cloudlets, to address the aforementioned difficulties. Renewable energy sources can be easily integrated into smart grid infrastructure to power fog technology that communicates directly with smart grids via access networks. Fog Computing offers processing, storage, and networking services between sensor networks and regular Cloud servers. A fog server is comparable to a tiny data center. Due to the local processing of sensitive user data, it also provides security and privacy for the Grid and sensor data. The Grid sensors are globally dispersed, and similarly, Fog devices are geographically dispersed, enabling real-time data analysis from these sensors.

1.2 Problem Definition

This thesis will be divided into two parts, with the first addressing the implementation of a GAN-based algorithm for anomaly detection in smart grid data. The second section will focus on executing the algorithm in different environments and measuring different run-time performance metrics, with time spent as the primary focus.

1.3 Outline

The rest of the thesis is organized as follows. Related works is described in Chapter 2. We introduce the algorithm structure and the different test platforms in Chapter 3. In Chapter 4, we give an introduction to the datasets which will be used for training and making predictions. Also we present the results of training and testing the algorithm on

said data. Then finally present the run times of the algorithm in the different platforms. In Chapter 5 we will discuss the results and further work, to conclude the thesis. All code used is available at Github.

Chapter 2

Related Work

2.1 Existing Approaches for Anomaly Detection

With the increasing use of artificial intelligence for pattern recognition, a quick search will reveal multiple articles discussing the various approaches and implementations of anomaly detection algorithms for smart grid data. Maatug has proposed multiple models for anomaly detection on a dataset made public by a Australian electricity company named Ausgrid.[9, 19] As the dataset does not contain labeled anomalies, they apply a time series forecasting model named Prophet label the outliers in the data. [20] Then, using four different algorithms, Cost-Sensitive Logistic Regression, DBSCAN, Ensemble Random Forest and One-Class Isolation Forest. From these four algorithms, she gathered the performance of each model from metrics such as recall, precision and F_1-score. The highest scores yielded from the Ensemble Random Forest Model prove a great performance of:

$$\text{Recall: } 0.975 \quad \text{Precision: } 0.985 \quad F_1\text{-score: } 0.978$$

These metrics could be used as a benchmark of the anomaly detection algorithm.

Multiple hypotheses utilizing a variety of statistical models to identify anomalies in smart grid data are discussed in the cited work. Zhang, Wan, and colleagues propose a Gaussian mixture model with linear discriminant analysis, clustering feature learning, and particle swarm optimization support for the Vector Machine. [5] Nielsen and Liu also propose a regression-based model that employs a prediction algorithm based on periodic auto-regression with exogenous variables. [10] These algorithms fail to recognize the seasonality of the data, which may result in undesirable outcomes. In experiments involving anomaly detection on smart grid data, the Prophet algorithm takes into account

a datapoint that is often overlooked. This is the seasonality of the data, as each customer's electricity consumption varies throughout the year's seasons. During the holiday season and winter, energy consumption varies significantly. Maatug mentions that multiple implementations of anomaly detection disregard the seasonality of the data. [9]

We conclude that using a dataset labeled by the Prophet algorithm will improve data quality further, and with higher quality data, we may expect a model with increased accuracy.

2.2 Previous Experiments On Fog Computing performance

Network bandwidth constraints for sending large IoT sensor data to Cloud servers have been addressed in several publications.[21, 22] Additionally, service providers are concerned about the ever-increasing amount of energy consumed by Cloud data center applications [23, 24]. Fog computing infrastructures are created with IoT applications in mind, including their large amounts of sensor data, low latency, and small memory footprint, as well as the prevention of grid-related dangers.

Researchers Jaiswal, Davidrajuh et al. are investigating the performance of a fog network using a real-time forecasting tool. A Fog device may have longer run-times, according to their findings, than a Cloud device while executing a real-time application in real-time. But as a note: the run-times don't account for any potential latency or other delays. We hope to further study the computational difference between running an application on a Fog network, compared to a Cloud solution.[19, 25]

Also Jaiswal, Chakravorty and Rong proposed an Fog framework for real-time anomaly detection using a Distributed Fog Computing architecture. In the research they deploy different machine learning algorithms on the architecture to measure performance of the network. The proposed architecture proved more effective and efficient as compared to Cloud computing architecture because of Big Data, location awareness and low latency requirements.

Chapter 3

Methodology Background and Theoretical Structure

3.1 Introduction

In the framework of Smart Grid, the initiative analyzes sensor data from smart meters. Edge/Fog Computing is employed as a platform for this data analysis since the design of Cloud Computing causes enormous delays in data processing and decision making. This delay may result in power dangers and an imbalance in the power supply. Despite the fact that Cloud Computing is suited for non-time-critical applications owing to its high latency, Fog Computing-based architecture is anticipated to integrate and support AI/ML-based applications, providing local real-time control to make devices adaptable and fast to react. In this chapter, we will provide an introduction to the technology and the academic that is required to grasp the material that will be presented in the subsequent chapters.

3.1.1 Machine Learning

Machine learning is a subfield of artificial intelligence in which statistical approaches are applied to enable computers to discover patterns in massive data sets. Instead of being programmed, we say that the machine "learns."

Learning is also known as model training. To "train," data is required. Typically, the data set is separated into a training set and a test set. The model is trained on the training set and then tested on the test set to determine whether it has learned anything. This is new information for the model, and it will indicate whether it has "learned" what

it is intended to. If the model has just memorized the data, equivalent to memorizing a text without comprehension, it will perform poorly on tests.

In recent years, transfer learning has gained popularity, particularly in computer vision and natural language processing. Here, we employ models that have been pre-trained on big data sets and for which training consumes significant resources. The pre-trained model is trained further to accommodate fresh data sets. Transfer learning has been proved to be effective and typically requires a smaller training set and less processing power than building a model from scratch.[26]

Modern applications of machine learning include self-driving cars, enhanced web search, email filtering, picture recognition, language recognition, enhanced compilers, and a deeper comprehension of the human genome.

Although the phrases are frequently used interchangeably, machine learning is a subset of artificial intelligence. Artificial intelligence embraces all intelligent systems. A contrast can be made between rule-based models and data-driven models. Rule-based models comprehend concepts via rules, which are frequently programmed before to use. This is not artificial intelligence.

The three primary categories of machine learning are supervised learning, unsupervised learning, and improved learning.

In this instance, the machine does not have access to output values for provided input values. Instead, the method attempts to discover the structure of the input values, for instance by clustering them. In this arena, where humans excel, machine learning has a long way to go before it can match human performance.

Generative adversarial network

A generative adversarial network(GAN) is a framework for estimating generative models through the use of an adversarial process. This is accomplished by simultaneously training a generator and a discriminator model. The generator attempts to capture the distribution of arbitrary data, whereas the discriminator attempts to estimate the chance that a given data sample was derived from training data rather than the generator. When the generator can deceive the discriminator approximately fifty percent of the time, we know that it is producing convincing examples that the discriminator cannot distinguish from training data. GANs can be utilized for a variety of applications, such as the development of indistinguishable photo realistic images of people, music, and artwork.

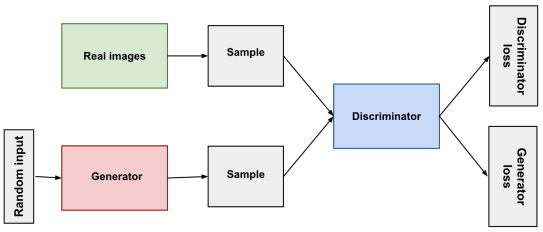


Figure 3.1: Illustration of a simple GAN [27]

Recurrent Neural Network

The Recurrent Neural Network (RNN) is a version of the feedforward neural network that has an internal memory. RNN is recurrent because it performs the same function for each data input while the output of the current input depends on the previous calculation. After generating the output, a copy of it is made and re-sent to the recurrent network. For decision-making, it examines both the current input and the output it has learnt from prior inputs. All inputs in other neural networks are independent from one another. In contrast, RNN's inputs are all interconnected.

The gradient vanishing problem is an inherent drawback of RNNs. This issue happens in ANNs that utilize gradient-based learning approaches and backpropagation. During each training iteration, the weights of the neural network are updated proportionally to partial derivative of the error function in respect to current weight. In some instances, the gradient will be infinitesimally tiny, thereby preventing the weight's value from changing.

Long short-term memory

Long Short-Term Memory (LSTM) networks are a modified kind of recurrent neural networks that make it simpler to retain prior material. Here, the RNN gradient problem with diminishing gradients is overcome. LSTM is ideally suited for classifying, processing, and predicting time series given time delays of uncertain duration. Back-propagation is utilized to train the model.

3.2 Anomaly detection approach/algorithm

The implementation of the algorithm will be done in the programming language Python. We will use the latest version of Python that supports the newest version of Tensorflow 2.8. The program will be greatly influenced by a previously implemented BiLSTMGAN model, created by Vee Upatising. [28]

3.2.1 Initializing Generator and Discriminator

As previously mentioned GANs utilize two models, a discriminator and a generator, by enabling the capability of BiLSTM the models to assess input sequences in both forward and reverse directions. As an adversarial network, the generator and discriminator are always attempting to "win" against one another in what is basically a min-max game. The generator attempts to minimize its loss, whereas the discriminator endeavors to maximize its loss.

The figures 3.2 and 3.3, shows the structure of the layers for the generator and discriminator, respectively.

Layer (type)	Output Shape	Param #
bidirectional_2 (Bidirectional)	(None, 8, 256)	133120
dropout_6 (Dropout)	(None, 8, 256)	0
bidirectional_3 (Bidirectional)	(None, 256)	394240
dropout_7 (Dropout)	(None, 256)	0
repeat_vector_1 (RepeatVector)	(None, 8, 256)	0
bidirectional_4 (Bidirectional)	(None, 8, 256)	394240
dropout_8 (Dropout)	(None, 8, 256)	0
bidirectional_5 (Bidirectional)	(None, 8, 256)	394240
dropout_9 (Dropout)	(None, 8, 256)	0
time_distributed_4 (TimeDistributed)	(None, 8, 256)	65792
leaky_re_lu_5 (LeakyReLU)	(None, 8, 256)	0
dropout_10 (Dropout)	(None, 8, 256)	0
time_distributed_5 (TimeDistributed)	(None, 8, 1)	257
leaky_re_lu_6 (LeakyReLU)	(None, 8, 1)	0

Total params: 1,381,889
Trainable params: 1,381,889
Non-trainable params: 0

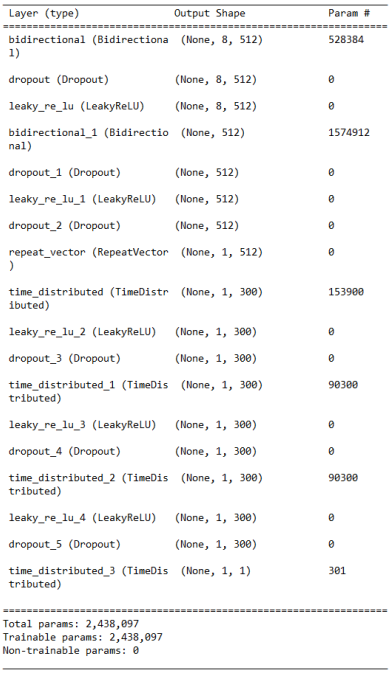
Figure 3.2: The Generator layers

The following code snippet is the code implementation of the Generator:

```
def build_generator():
    model = Sequential()
    # encoder
    model.add(Bidirectional(LSTM(128, return_sequences=True), input_shape=(8, 1)))
    model.add(Dropout(0.2))
    model.add(Bidirectional(LSTM(128)))
    model.add(Dropout(0.2))
    # specifying output to have 8 timesteps
    model.add(RepeatVector(8))
    # decoder
    model.add(Bidirectional(LSTM(128, return_sequences=True)))
    model.add(Dropout(0.2))
    model.add(Bidirectional(LSTM(128, return_sequences=True)))
    model.add(Dropout(0.2))
    model.add(TimeDistributed(Dense(256)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(TimeDistributed(Dense(1)))
    model.add(LeakyReLU(alpha=0.2))
    model.summary()

    noise = Input(shape=(8, 1))
    img = model(noise)

    return Model(noise, img, name='Generator')
```



Layer (type)	Output Shape	Param #
bidirectional (Bidirectiona 1)	(None, 8, 512)	528384
dropout (Dropout)	(None, 8, 512)	0
leaky_re_lu (LeakyReLU)	(None, 8, 512)	0
bidirectional_1 (Bidirectio nal)	(None, 512)	1574912
dropout_1 (Dropout)	(None, 512)	0
leaky_re_lu_1 (LeakyReLU)	(None, 512)	0
dropout_2 (Dropout)	(None, 512)	0
repeat_vector (RepeatVector)	(None, 1, 512)	0
time_distributed (TimeDistr ibuted)	(None, 1, 300)	153900
leaky_re_lu_2 (LeakyReLU)	(None, 1, 300)	0
dropout_3 (Dropout)	(None, 1, 300)	0
time_distributed_1 (TimeDis tributed)	(None, 1, 300)	90300
leaky_re_lu_3 (LeakyReLU)	(None, 1, 300)	0
dropout_4 (Dropout)	(None, 1, 300)	0
time_distributed_2 (TimeDis tributed)	(None, 1, 300)	90300
leaky_re_lu_4 (LeakyReLU)	(None, 1, 300)	0
dropout_5 (Dropout)	(None, 1, 300)	0
time_distributed_3 (TimeDis tributed)	(None, 1, 1)	301

Total params: 2,438,097
Trainable params: 2,438,097
Non-trainable params: 0

Figure 3.3: The Discriminator layers

And for the code snippet initializing the Discriminator:

```
def build_discriminator():
    model = Sequential()

    model.add(Bidirectional(LSTM(256, return_sequences=True), input_shape=(8, 1)))
    model.add(Dropout(0.2))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Bidirectional(LSTM(256)))
    model.add(Dropout(0.2))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(RepeatVector(1))
    model.add(TimeDistributed(Dense(300)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(TimeDistributed(Dense(300)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(TimeDistributed(Dense(300)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(TimeDistributed(Dense(1)))
    model.summary()

    img = Input(shape=(8, 1))
    validity = model(img)

    return Model(img, validity, name='Discriminator')
```

We have now looked at the initialization of the generator and discriminator, and now we will see the implementation as a whole. The model is created as Python class named LSTMGAN. The following code snippet shows the constructor of the class.

```
def __init__(self, learning_rate=0.00001):
        optimizer = Adam(learning_rate)

        # Build and compile the discriminator
        self.discriminator = build_discriminator()
        self.discriminator.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['ac

        self.generator = build_generator()

        # The generator takes noise as input and generates samples
        z = Input(shape=(8, 1))
        sample = self.generator(z)

        # For the combined model we will only train the generator
        self.discriminator.trainable = False

        # The discriminator takes generated samples as input and determines validity
        valid = self.discriminator(sample)

        # The combined model  (stacked generator and discriminator)
        # Trains the generator to fool the discriminator
        self.combined = Model(z, valid)
        self.combined.compile(loss='mean_squared_error', optimizer=optimizer)
```

The constructor initializes the generator and discriminator with an Adam optimizer. The Adam optimizer is a built in class in Keras. [29] This optimizer makes use of the Adam algorithm in place of the classical stochastic gradient descent procedure in order to update the network weights in an iterative fashion depending on training data. The Adam algorithm leverage a combination of two extensions of stochastic gradient descent; Adaptive Gradient Algorithm(AdaGrad) and Root Mean Square Propagation(RMSProp).[30] The optimizer can be configured with multiple parameters, but only the alpha parameter, also called learning rate, is relevant for our usage.

After the initialization of the LSTMGAN-class, the model is now ready to be trained on a labeled dataset.

```
def train(self, dataset, epochs, batch_size=128):
        # Training performance measurements
        generated_data = [[] for _ in range(epochs)]
        training_data = [[] for _ in range(epochs)]
        generated_average = []
        training_average = []
        accuracy = [[] for _ in range(epochs)]

        # Load the dataset
        X_train = dataset
```

```
batch_count = X_train.shape[0] // batch_size

# Adversarial ground truths
valid = np.ones((batch_size, 1, 1))
fake = np.zeros((batch_size, 1, 1))

g_loss_epochs = np.zeros((epochs, 1))
d_loss_epochs = np.zeros((epochs, 1))

for epoch in range(epochs):
    for index in range(batch_count):
        # ---------------------
        #  Train Discriminator
        # ---------------------

        # Select a batch of data
        training_batch = X_train[index * batch_size: (index + 1) * batch_size]
        training_data[epoch].extend(training_batch.flatten())

        # Sample noise and generate a batch of new images
        noise = np.random.normal(0, 1, (batch_size, 8, 1))
        generated_batch = self.generator.predict(noise)
        generated_data[epoch].extend(generated_batch.flatten())

        # Train the discriminator (real classified as ones and generated as zeros)
        d_loss_real = self.discriminator.train_on_batch(training_batch, valid)
        d_loss_fake = self.discriminator.train_on_batch(generated_batch, fake)
        d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

        # ---------------------
        #  Train Generator
        # ---------------------

        # Train the generator (wants discriminator to mistake images as real)
        g_loss = self.combined.train_on_batch(noise, valid)

        # save loss history
        g_loss_epochs[epoch] = g_loss
        d_loss_epochs[epoch] = d_loss[0]

    # Plot the progress
    generated_average.append(pd.Series(generated_data[epoch]).rolling(window=50).mean())
    training_average.append(pd.Series(training_data[epoch]).rolling(window=50).mean())
    timenow = datetime.now()
    print(timenow.strftime("%Y-%m-%d %H:%M:%S"), "%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" %
    accuracy.append(100 * d_loss[1])

return g_loss_epochs, d_loss_epochs, accuracy, \
       [generated_data, training_data, generated_average, training_average]
```

After training the generator and discriminator on a batch of data, anomaly detection can be performed, using the discriminator of the LSTMGAN-class `LSTMGAN.discriminator.predict(dataset)`, this returns a true or false value, where a true value indicates that the given data is

classified as an anomaly. Using the generated discriminator to predict on a given dataset, it is straightforward to determine whether or not the predictions yield anomalies. By comparing the projected anomalies to the genuine anomalies of the dataset, it is feasible to assess the discriminator's performance.

Figure 3.4 displays the architecture of the algorithm.

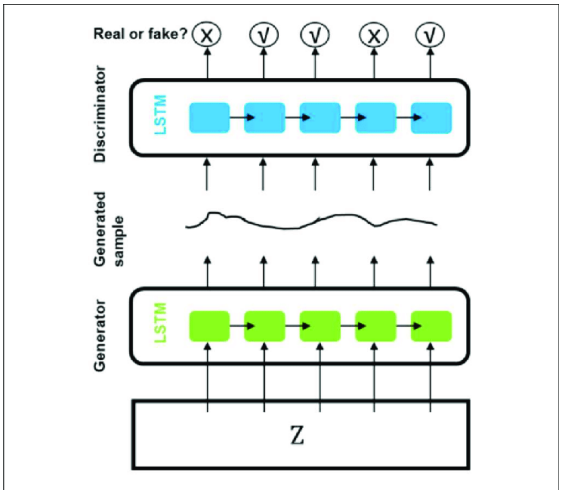


Figure 3.4: Architecture of Algorithm

3.3 Fog/Edge computing

Fog computing expands the concept of Cloud computing by offering processing, storage, and networking capabilities between end devices and the standard Cloud server. Typically, Fog Computing nodes are positioned at the network's edge, distant from the primary Cloud Data Centres.[21, 31] The fog server is comparable to a cloud server in that it receives information from different fog networks that are connected to it, but with less computational and storage capability. The fog server is a virtualized environment in which numerous virtual machines are designed to execute fog applications. Additionally, the fog server transmits pertinent data to the main cloud server for large-scale data processing and service provisioning. It is essential to understand that Fog Computing does not replace Cloud Computing; rather, it will serve as a critical adjunct to Cloud Computing.

NVIDIA® Jetson Nano™

The NVIDIA Jetson Nano Developer Kit is a compact and powerful computer that enables the concurrent execution of numerous neural networks for applications such as

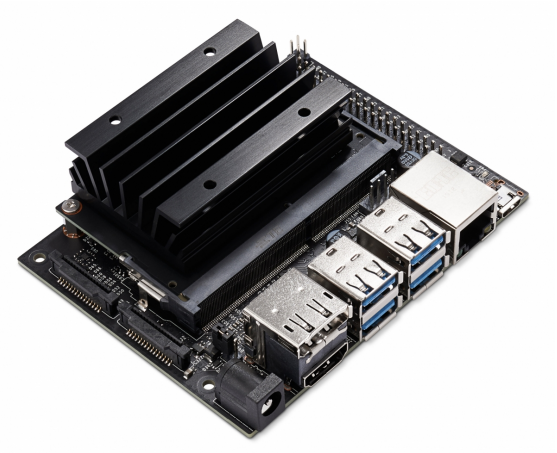


Figure 3.5: A NVIDIA Jetson Nano developer board

image classification, object recognition, segmentation, and audio processing. All on a platform that operates with as low as 5 watts.

The board is designed to deliver strong GPUs for edge computing robotics and AI applications. The developer kits are generally used for software and hardware implementation prototyping and testing. The integrated GPU in the Jetson line-up makes it suited for this application. The Jetson Lineup utilizes Ubuntu 18.04 with OpenCV and CUDA preloaded for GPU computation. In production, the development board should be discarded and replaced by the NVIDIA Jetson Module. The available I/Os on development kits are the primary distinction between developer kits and modules. Modules are only available as computer boards devoid of I/Os.

The Jetson Nano has the following specifications:

OS	Ubuntu 18.04 - aarch64
GPU	128-core Maxwell
CPU	Quad-core ARM A57 @ 1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s

Local Computer

Over the course of the project, the algorithm has been developed and tested on a local computer. The computer has the following specifications:

OS	Windows 10 64bit
GPU	NVIDIA GeForce RTX 2080
CPU	AMD Ryzen 5 5600X 6-Core Processor 3.70 GHz
Memory	32 GB 64-bit DDR4 21.3 GB/s

3.3.1 Cloud computing

Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. Cloud computing is comprised of three service models: Software as a Service, Platform as a Service, and Infrastructure as a Service. The service models are offered by providers, which can be public, private, or community providers, as well as hybrids. Regardless of its deployment or service model, cloud computing services are powered by large data centers comprised of numerous virtualized server instances and high-bandwidth networks, as well as of supporting systems such as cooling and power supplies.

Google Cloud Platform

Google Cloud Platform is Google's cloud service and one of the world's largest and most popular cloud services. It was initially introduced in April 2008, and its primary purpose was to host web apps on Google's infrastructure. Since then, Google has continued to expand the service and added more cloud-based offerings. In this project, we will use Vertex AI, an AI tool from Google Cloud Platforms. Vertex AI's API, client library, and user interface allow you to build, deploy, and scale machine learning models. In our example, Vertex AI will host a Jupyter Notebook environment, which is where we will deploy our anomaly detection algorithm script. Then we will measure the performance with different machine types.

The machine types used are: n1-standard-4 - N1 machine is a general purpose machine, which is the default machine type when starting a Vertex AI service. The general purpose machines are marketed as the best price-performance ratio over a variety of workloads.

OS	Linux with debian 64bit
GPU	-
CPU	4 x Intel Xeon - multiple types 2.0 - 2.6 GHz
Memory	15 GB
Price	0.133$ per hour

n1-highcpu-16 - Also a N1 machine, but configured to handle tasks that are compute heavy, relative to RAM necessary for a task.

OS	Linux with debian 64bit
GPU	-
CPU	16 x Intel Xeon - multiple types 2.0 - 2.6 GHz
Memory	15 GB
Price	0.397$ per hour

Chapter 4

Experimental Evaluation

This study's primary objective is to identify anomalies in power consumption data and to identify the best model for classifying future events. Then, we will test the model on various platforms in order to analyze its run-metrics. This chapter outlines the steps required to achieve this objective.

4.1 Datasets

The project used mainly two datasets which will be further described in this subchapter. The datasets are of equal origin, whereas one contains a time series of anomaly labeled data, and the other is just the raw data, without any additions or prepossessing.

4.1.1 Ausgrid Dataset

The original dataset is supplied by an Australian electricity company named Ausgrid. The company describes the data:

> "The data has been sourced from 300 randomly selected solar customers in Ausgrid's electricity network area that were billed on a domestic tariff and had a gross metered solar system installed for the whole of the period from 1 July 2010 to 30 June 2013. The customers chosen had a full set of actual data for the period from 1 July 2010 to 30 June 2011, gathered through our meter reading processes. We also undertook some data quality checking and excluded customers on the high and low ends of household consumption and solar generation performance during the first year. ... The customers in this dataset may not represent a statistically relevant sample of residential

customers in the Ausgrid network area, and have not been surveyed to collect household characteristics. Typically, households that install solar systems own their home and live in separate houses with the available roof space needed to install a solar power system." Ausgrid, 22. August 2014 [32]

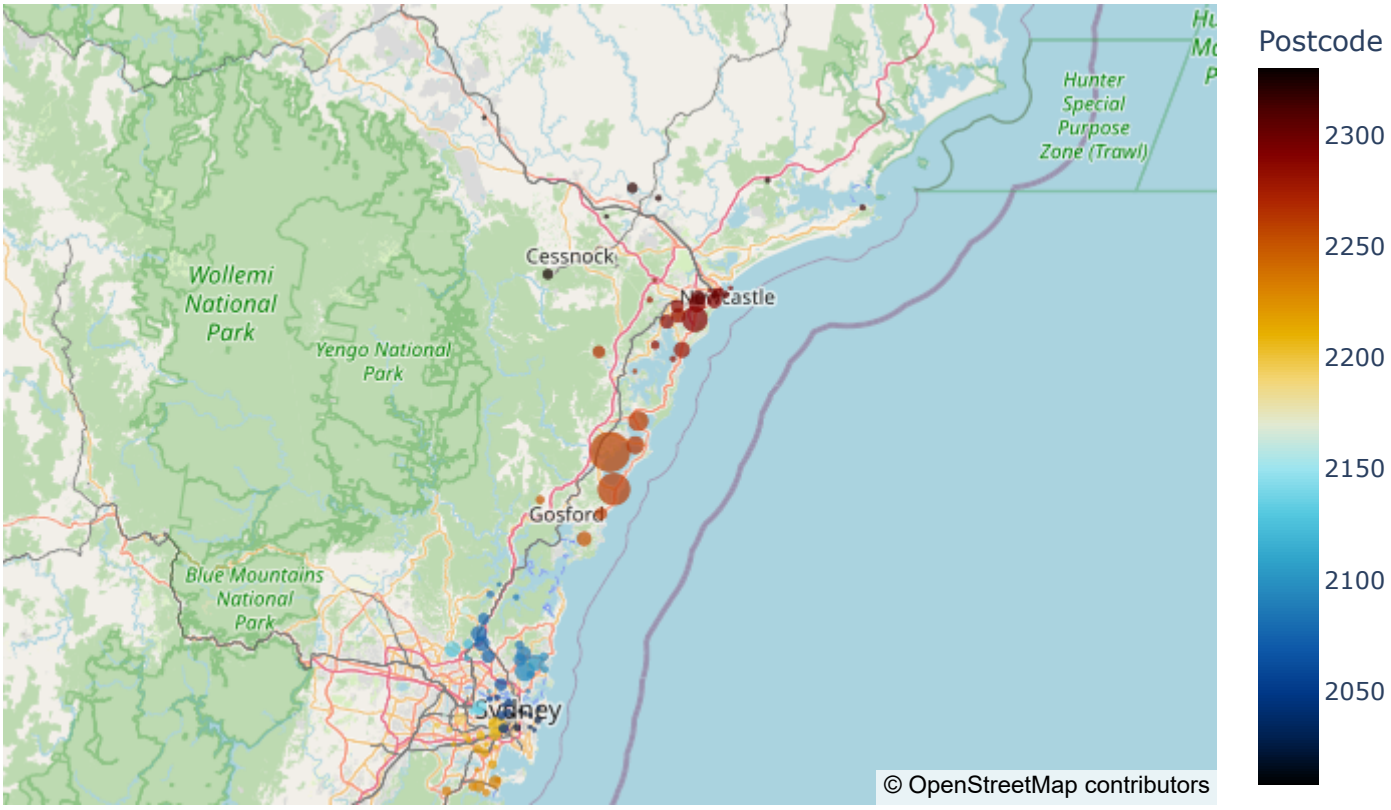


Figure 4.1: Map displaying density of customer locations. Color palette describes which postcode the customer belongs in.

Ausgrid dataset notes document

With the Ausgrid data, there follows a document with descriptions of the data structure.[33] The most interesting information is included below. Each row of data has the following format:

Column	Field	Description
1	Customer	Customer ID from 1 to 300
2	Postcode	Postcode location of customer
3	Generator Capacity	Solar panel capacity recorded on the application for connection for each customer. Units are Kilowatt Peak (kWp), which is the solar panels peak power under full solar radiation and tested under standard conditions.
4	Consumption Category	Two letter code each meaning the following: GC = General Consumption for electricity supplied all the time (primary tariff, either inclining block or time of use rates), excluding solar generation and controlled load supply CL = Controlled Load Consumption (Off peak 1 or 2 tariffs) GG = Gross Generation for electricity generated by the solar system with a gross metering configuration, measured separately to household loads
5	Date	Date in DDMMMYYYY format.
6	0:30	Kilowatt hours (kWh) of electrical energy consumed or generated in the half hour ending at 0:30 (eg. between 0:00 and 0:30). The value is positive regardless of whether it is consumption or generation.
7 to 53	1:00. . . . . 0:00	As above, covering every half hour of the day up until the last half hour of the day at 0:00, (eg. between 23:30 to 0:00).

4.1.2 Labeled dataset

To train the GAN model, we use a labeled dataset, generated from a previous implementation of anomaly detection using the Prophet method.[9] This dataset is an preprossesed and labeled version of the previously mentioned Ausgrid dataset.

4.2 Anomaly Detection Setup - hyperparameters, prossesing data

Results from experimentation have been split into results from running tests on a single day of data capture, and from running tests on an entire month. Everything shown is from a single household from the Ausgrid dataset.

Hyper-parameters

To find the best hyper-parameters one would usually use a method called Hyper-parameter Optimization. Hyper-parameter Optimization or Hyper-parameter Tuning can be defined as choosing the right set of values in building a machine learning model, in our case the machine learning model is GAN.

In this model the GAN hyper-parameters can be summarized to:

1. Learning Rate
    - The Learning Rate is what determines how the model will change in response to new weight updates that are determined by the amount of error produced by the output. It determines how quickly a model can adjust to new information about a situation. LR is particularly significant since it can decide whether or not weight updates become "stuck" as a result of tiny rates of change, or whether or not they become excessively quick and unstable as a result of high learning rates. To be more specific, it refers to the rate at which the stochastic gradient descent oscillates in order to find the value that corresponds to the lowest feasible cost.
2. Batch Size
    - Batch size determines how many datapoints/examples to include in each iteration of model training. After a iteration, the model makes a single update to the model's weights.
3. Number of Epochs
    - An epoch is one complete pass over of the dataset. This means that each example or datapoint is seen once. An epoch will be split into n-examples/batch-size segments.

To optimize hyper-parameters, a variety of performance metrics are considered. Using the Python module sklearn, we have access to a function called precision_recall_fscore_support. This function computes each class's precision, recall, F-measure, and support. Before examining these metrics, let's familiarize ourselves with the classification terms True versus False and Positive versus Negative.

Described easily in Googles Machine Learning Crash Course: "A true positive is an outcome where the model correctly predicts the positive class. Similarly, a true negative is an outcome where the model correctly predicts the negative class. A false positive is an outcome where the model incorrectly predicts the positive class. And a false negative is an outcome where the model incorrectly predicts the negative class."

Precision is the capacity of the classifier to avoid mislabeling negative samples as positive.

$$tp/(tp + fn)$$

Recall is the classifier's ability to locate all positive samples.

$$tp/(tp + fn)$$

The F-measure (measures F_β and F_1) can be understood as the weighted harmonic mean of the precision and recall. A F_β measure reaches its maximum value of 1 and its minimum value of 0 at 1. When $\beta = 1$, F_β and F_1 are equivalent,and both recall and precision are equally important.

Support is the number of real instances of a class in a given dataset. The requirement for stratified sampling or rebalancing may be indicated by unbalanced support in the training data, which may imply fundamental problems in the reported scores of the classifier. As opposed to differing between models, support diagnoses the evaluation procedure.

Following a comprehensive analysis of a variety of potential hyper-parameters, we are left with the following optimal setup.

Number of Epochs: 100 Learning rate: 0.00001 Batch size: 512

Sample initialization of the LSTM-class, using the specified hyperparameters for training:

```
lstmgan = LSTMGAN(learning_rate = 0.00001)
lstmgan.train(dataset = prepossessed_dataset, epochs = 100, batch_size=512)
```

4.3 Anomaly Detection Results

The findings of the anomaly detection model are next on our agenda to deliver.

When the training of the GAN has been completed using the given parameters, we can plot the resultant loss of the model. We compare the loss of the discriminator to the loss of the generator, we can see that they are gradually approaching an optimal state. One thing to note, is that while usually the discriminator and generator should converge, in this case they fail to do so. This will be further looked into in the discussion chapter.

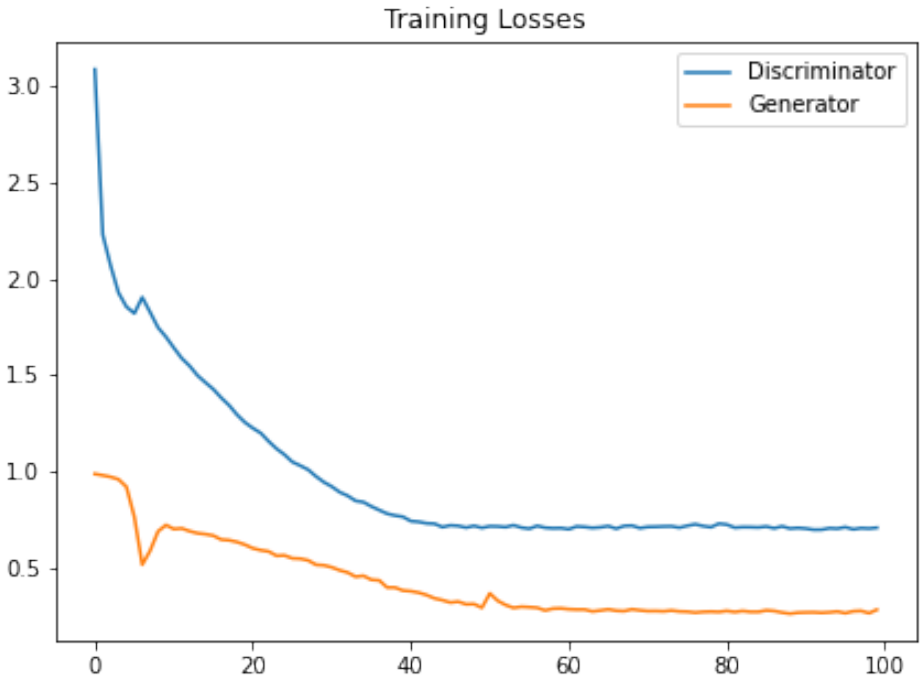


Figure 4.2: Training loss for Generator and Discriminator

To see the progress of the generator we keep track of the generated examples for each epoch. In the following figures we will see the generated data compared to actual data. The goal is to see an clear overlap of the two, as the generators generated datapoints should match the distribution of the actual data.

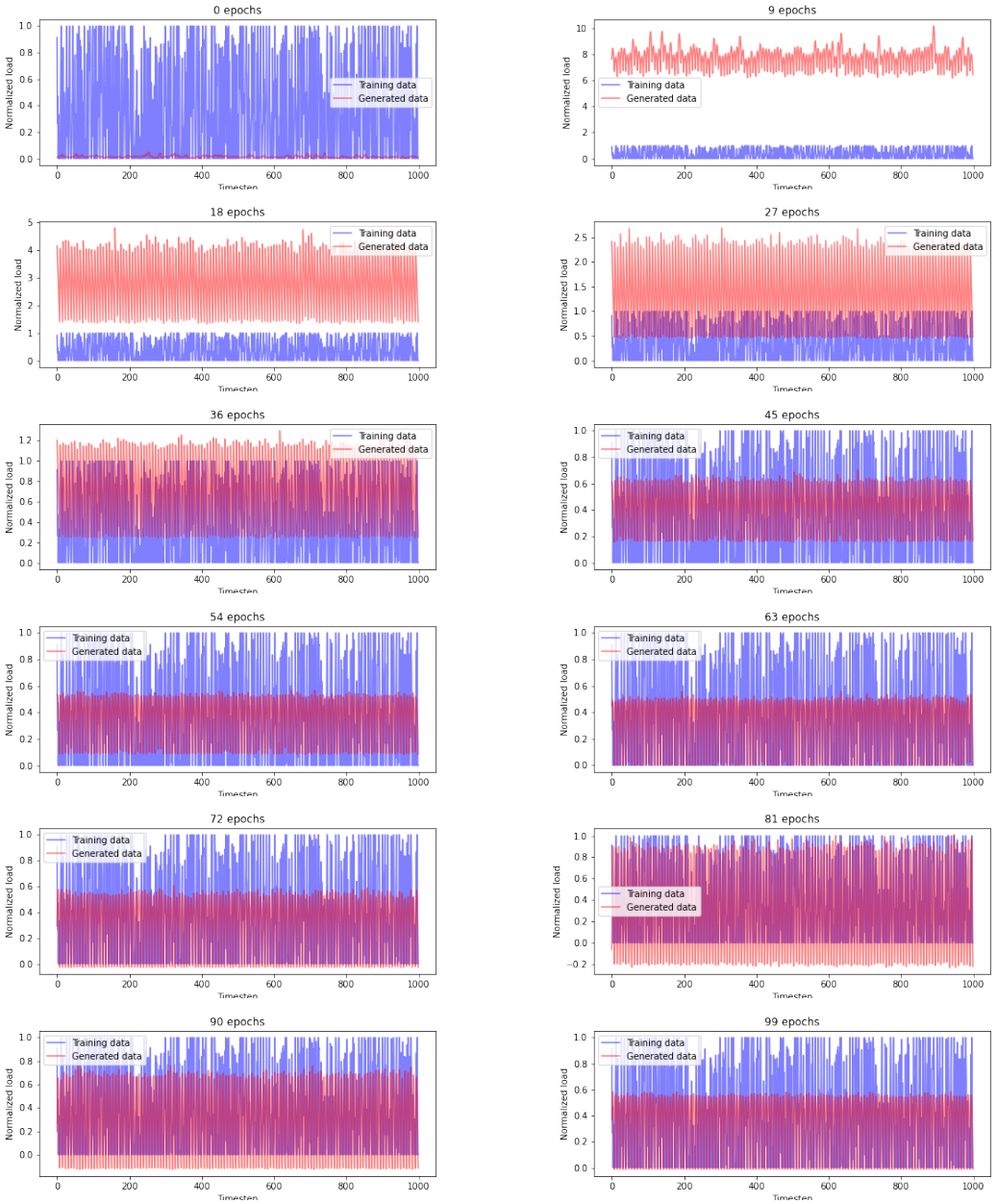


Figure 4.3: Generated data(red) compared to actual data(blue), over 100 epochs

What we gather from Figure 4.3 is the overall training of the generator. Looking at the different examples for each epoch, there is a clear improvement between the earlier epochs compared to the later. But, when looking at epoch 90 -> 99, its not obvious if the last 9 epochs are necessarily an improvement for the generator.

As mentioned, we want the generator to create datasets with equal distribution of the original data. To further look at the performance of the generator, we therefore will take a look at the same epochs, but with the average generated values. The goal for the generator in these plots, is to converge onto the original data.

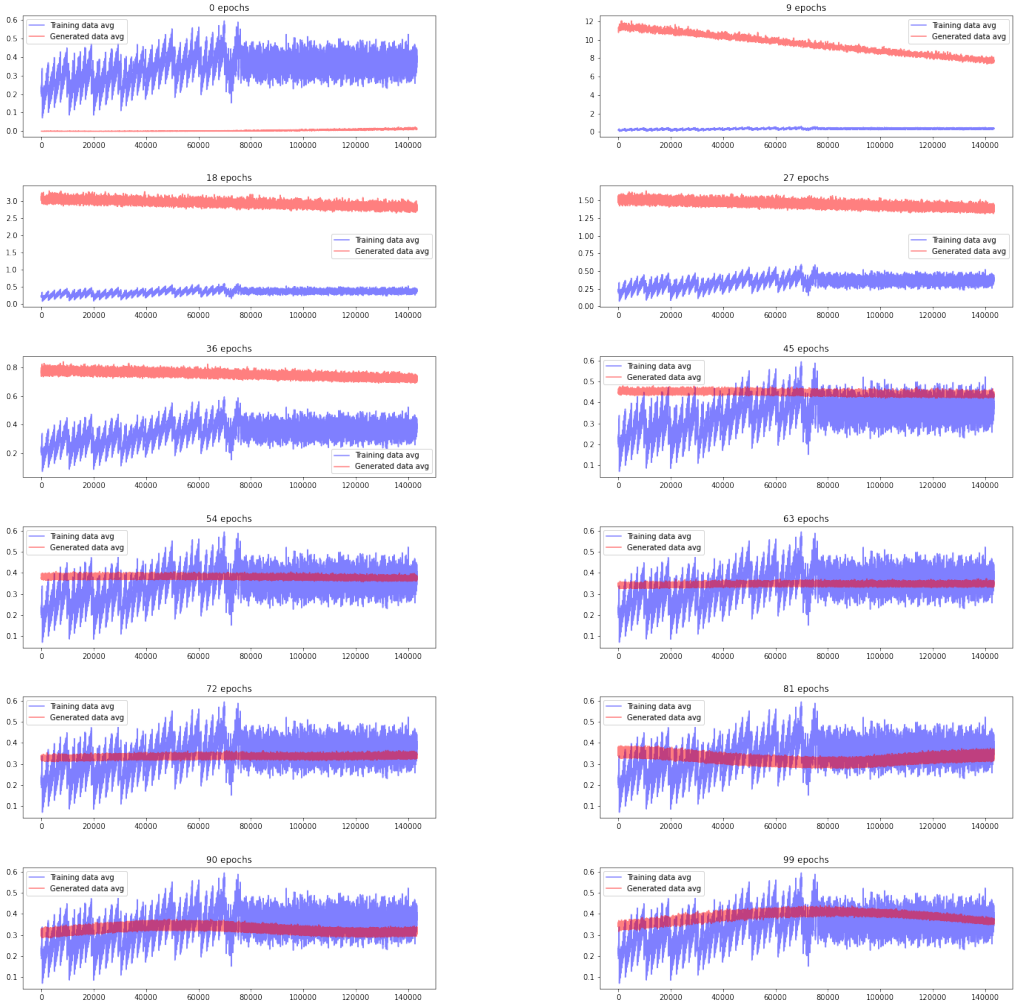


Figure 4.4: Generated data(red) compared to actual data(blue), over 100 epochs

The average value, shown in Figure 4.4, also shows a clear indication of improvement between the earlier epochs to the later. but no clear indication of improvement in the later epochs. Further, we keep track of the discriminator accuracy for each epoch, which should be the final evaluation of the generator. Should the discriminator have a correct selection accuracy of 50% or below, we know the generator is able to fool the discriminator sufficiently. We therefore keep track of the accuracy for each epoch. See Figure 4.5 for discriminator accuracy.

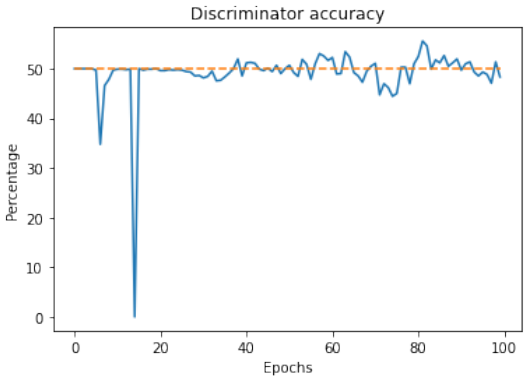


Figure 4.5: Discriminator accuracy over the training epochs

After reviewing the models training, the discriminator will now be able to label anomalies.

We run the anomaly detection using the trained discriminator and get a resulting set of predicted y-values, i.e. the predicted anomalies. This is a list of boolean values where a 0 represent good measurements and 1 represent anomalies. By evaluating the predicted y-values compared to the true y-values we get an accuracy of 90.3 percent. To further visualize the results of the anomaly detection we can plot the time-series data and overlay the true anomalies on one plot and the predicted anomalies on another. By doing this it becomes simple to see if the predicted results are satisfying.

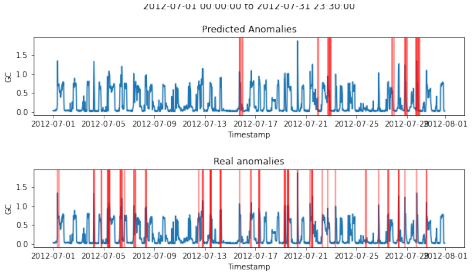


Figure 4.6: Customer 202 Predicted Anomalies and Real Anomalies between 2012-07-01 to 2012-07-31(One month duration)

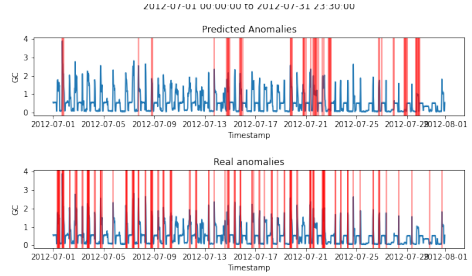


Figure 4.7: Customer 206 Predicted Anomalies and Real Anomalies between 2012-07-01 to 2012-07-31(One month duration)

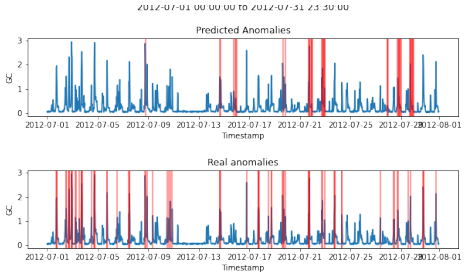


Figure 4.8: Customer 215 Predicted Anomalies and Real Anomalies between 2012-07-01 to 2012-07-31(One month duration)

We can gather from the plots, that while the accuracy previously specified under training yielded, a result of 90.3 percent accuracy, this does not seem to be the case when applying the algorithm to test data. We see multiple anomalies are left undetected, while also having mislabeled anomalies. The cause of the bad performance will be further discussed in the next chapter.

4.4 Running in Fog and cloud results

This section outlines the results from running the anomaly detection algorithm on different platforms with varying specifications. All experiments are conducted by running a script, which is implemented to complete all steps of for a complete anomaly detection. This includes prepossessing, training the model and finally a prediction. As the most time-consuming actions are training the model, and making predictions, this are where we will keep track of time spent. The timing is done by using the built in module named time in Python, and we expect the time spent will range from a few seconds to hours.

The script is implemented using Python 3.10, with Tensorflow 2.8.0 and Keras as a key modules for building the GAN. From these modules we make several functions and classes available at our disposal, to help implement the model. Both the Generator and the Discriminator models are derived from the Sequential class, then supplied with the specified layers from figure 3.2 and 3.3. We also gain access to the Adam optimizer, which was previously mentioned.

4.4.1 Computing time results

The following table summarize the time spent training the model on each test-platform(train time given in HH:mm:ss):

Local Computer	1:31:31.05
NVIDIA Jetson Nano	20:45:51.65
n1-standard-4	3:40:54.59
n1-highcpu-16	2:16:51.91

And then the time spent to make predictions for customer 202, 206 and 215(time given in seconds):

Local Computer	4.50
NVIDIA Jetson Nano	34.87
n1-standard-4	6.88
n1-highcpu-16	6.28

The time spent on the different actions differ drastically from each test platform. There is a clear distinction to be made between the more expensive solutions to the more affordable options. Further we will discuss the results and keypoints in the next chapter.

Chapter 5

Discussion

As the results now have been presented, we are now ready to discuss the outcomes of our experiments. First, we will discuss the anomaly detection algorithm, mainly focusing on its performance, and what variables could have changed the outcome of our results.

5.1 Anomaly Detection Discussion

5.1.1 Dataset

Although the Prophet forecasting algorithm's labeled dataset appears promising in its ability to label outliers as possible anomalies, it cannot be expected to correctly classify all anomalies. [9] As the number of True Negatives and False Positives (incorrect labels) is unknown, there will be a degree of uncertainty for each datapoint. Our algorithm will not be able to correctly identify actual anomalies if the dataset has been incorrectly labeled, so we must assess this possibility. In this instance, the only solution would be to train the algorithm on a real labeled dataset, as opposed to the open-source Ausgrid dataset, which lacks labeled anomalies.

5.1.2 Algorithm implementation

If we disregard the variance of dataset accuracy and only concentrate on the implementation of the algorithm, there are only a few variables that can be adjusted to improve the classifier. In addition to discussing the hyperparameters (learning rate, batch size, and epochs), we outlined the layers for the generator and discriminator. These are crucial variables for the performance of the model, and further examination reveals that

section 4's results are inferior to those of the Ensemble Random Forest Model, which had excellent performance.

Experiment with the layer composition of the generator and discriminator in order to implement a faster convergence of the generator and discriminator loss. After the composition of the layers has been proposed, a new hyperparameter-tuning session should be conducted to identify the parameters that provide the best performance with the new layers.

5.2 Run times in platforms

We will now discuss the results of a complete run-trough of the algorithm on the different platforms.

5.2.1 Tensorflow GPU and CPU

Tensorflow was previously mentioned as a package used to assist in the implementation of the algorithm. Tensorflow is an API for Python that enables the use of GPU for processing compute-intensive tasks, rather than the default behavior of using the CPU. Datamadness compared the training of neural networks on a state-of-the-art GPU and CPU and found that this significantly reduces training time. [34] Using the GPU as opposed to the CPU, the authors of this article are able to reduce the training time of neural networks by 85 percent.

Unfortunately, based on our testing across multiple platforms, we were only enabling the CPU for computation when training models. Due to the configurations of the Jetson Nano board, there was no possibility of Anaconda environment management. This is an issue that must be addressed in future work, as the results could be vastly different from ours.

5.2.2 Results

As previously stated, the computing time results for each platform vary significantly. The Local test computer has the best performance, but because it is a relatively new personal computer with limited scalability, we cannot consider it for a long-term solution.

We will instead focus on the slow processing speed of the Jetson Nano board, as the time difference between it and cloud-based virtual machines is substantial. On Jetson, training a model took over nine times longer than on the n1-highcpu VM and over five

and a half times longer than on the n1-standard VM. Creating predictions for customers 202, 206, and 215 on the Jetson board took over five times longer than on the cloud VMs. This demonstrates conclusively that either the NVIDIA Jetson Nano board is unsuitable for running an anomaly detection algorithm using BiLSTMGAN for classifications, or that the board has been misutilized due to improper installation or code structure.

5.3 Summary and Conclusion

In conclusion, we proposed in this thesis an algorithm for anomaly detection of smart meter data employing an LSTM GAN classifier on historical power consumption data. The run-times of the algorithm were then measured on cloud and fog platforms. We analyzed previously implemented algorithms to establish a standard for measuring the algorithm's accuracy, as well as similar projects experimenting with Fog networks and their execution time for various applications. Test results indicate that it is possible to make predictions on power consumption data with a certain degree of accuracy, but there are no significant improvements relative to other statistical algorithms. The results also indicate that improper configuration of Tensorflow applications may result in significant run-time delays. Consider the fact that the initial dataset does not contain any labeled anomalies; consequently, when such a dataset becomes available, much of the previous work should be backtested.

List of Figures

Appendix A

Instructions to Compile and Run System

Required versions: Python 3.10, Tensorflow 2.8.0

1. Unzip attached files.
2. To install all packages necessary run the following command where the requirements.txt file is located:

   ```
   pip install -r requirements.txt
   ```

3. To run the algorithm from start to finish run the following command:

   ```
   python LSTMGAN-completerun.py
   ```

4. To experiment with different numbers of epochs, batch size and learning rate, simply edit the scripts line 266-269.

LSTMGAN-completerun.py

```
from __future__ import print_function, division

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_recall_fscore_support
from sklearn.metrics import roc_curve, auc
import pickle

from keras import Sequential, Input, Model
from keras.layers import Bidirectional, Dropout, LSTM, RepeatVector, TimeDistributed, LeakyReLU,
from sklearn.metrics import confusion_matrix
```

```
from tensorflow.keras.optimizers import Adam
from keras.models import load_model

import datetime
import time

from sklearn.model_selection import train_test_split
from tqdm import tqdm

data_path = './datasets/'
model_path = './models/'

def anomaly_detection(x_test, y_test, batch_size, discriminator):
    nr_batches_test = np.ceil(x_test.shape[0] // batch_size).astype(np.int32)

    results = []

    for t in range(nr_batches_test + 1):
        ran_from = t * batch_size
        ran_to = (t + 1) * batch_size
        image_batch = x_test[ran_from:ran_to]
        tmp_rslt = discriminator.predict(
            x=image_batch, batch_size=128, verbose=0)
        results = np.append(results, tmp_rslt)

    pd.options.display.float_format = '{:20,.7f}'.format
    results_df = pd.concat(
        [pd.DataFrame(results), pd.DataFrame(y_test)], axis=1)
    results_df.columns = ['results', 'y_test']
    # print ('Mean score for normal packets :', results_df.loc[results_df['y_test'] == 0, 'results'].me
    # print ('Mean score for anomalous packets :', results_df.loc[results_df['y_test'] == 1, 'results']

    # Obtaining the lowest 3% score
    per = np.percentile(results, 3)
    y_pred = results.copy()
    y_pred = np.array(y_pred)

    # Thresholding based on the score
    inds = (y_pred > per)
    inds_comp = (y_pred <= per)
    y_pred[inds] = 0
    y_pred[inds_comp] = 1

    return y_pred, results_df

def load_data():
    with open(data_path + 'dataset_one_month_not_shuffled.pkl',
              'rb') as f:  # 'dataset_one_week_not_shuffled.pkl', 'rb') as f:
        dataset = pickle.load(f)
    x_train, y_train, x_test, y_test = dataset['x_train'], dataset['y_train'], dataset['x_test'], datas
    return x_train, y_train, x_test, y_test

def plot_confusion_matrix(cm, title='Confusion matrix', cmap=plt.cm.Blues):
    target_names = ['normal', 'anomaly']
    # plt.figure(figsize=(10,10),)
```

```
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(target_names))
    plt.xticks(tick_marks, target_names, rotation=45)
    plt.yticks(tick_marks, target_names)
    plt.tight_layout()

    width, height = cm.shape

    for x in range(width):
        for y in range(height):
            plt.annotate(str(cm[x][y]), xy=(y, x),
                         horizontalalignment='center',
                         verticalalignment='center')
    plt.ylabel('True label')
    plt.xlabel('Predicted label')


def plot_roc_curve(y_test, y_pred):
    fpr_keras, tpr_keras, thresholds_keras = roc_curve(y_test, y_pred)
    auc_keras = auc(fpr_keras, tpr_keras)
    # plt.figure(1)
    plt.plot([0, 1], [0, 1], 'k--')
    plt.plot(fpr_keras, tpr_keras,
             label='Keras (area = {:.2f})'.format(auc_keras))

    plt.xlabel('False positive rate')
    plt.ylabel('True positive rate')
    plt.title('ROC curve')
    plt.legend(loc='best')
    plt.show()

def run_and_plot(dataset, learning_rate, batch_size, epochs, save_interval=50):
    x_train, y_train, x_test, y_test = dataset['x_train'], dataset[
        'y_train'], dataset['x_test'], dataset['y_test']
    gan = LSTMGAN(epochs, learning_rate)
    discriminator_loss, gan_loss = gan.train(epochs, batch_size, save_interval)  # x_train, batc
    y_pred = anomaly_detection(x_test, y_test, batch_size, gan.discriminator)
    print(len(y_pred), len(y_test))
    # acc_score, precision, recall, f1 = evaluation(y_test, y_pred)

    plt.figure(figsize=(20, 20))

    plt.subplot(2, 1, 1)
    plt.plot(discriminator_loss, label='Discriminator')
    plt.plot(gan_loss, label='Generator')
    plt.title("Training Losses")
    plt.legend()

    plt.subplot(2, 2, 3)
    cm = confusion_matrix(y_test, y_pred)
    plot_confusion_matrix(cm)

    plt.subplot(2, 2, 4)
```

```
    plot_roc_curve(y_test, y_pred)

    plt.show()

    return y_pred, gan


def build_generator():
    model = Sequential()
    # encoder
    model.add(Bidirectional(LSTM(128, return_sequences=True), input_shape=(8, 1)))
    model.add(Dropout(0.2))
    model.add(Bidirectional(LSTM(128)))
    model.add(Dropout(0.2))
    # specifying output to have 8 timesteps
    model.add(RepeatVector(8))
    # decoder
    model.add(Bidirectional(LSTM(128, return_sequences=True)))
    model.add(Dropout(0.2))
    model.add(Bidirectional(LSTM(128, return_sequences=True)))
    model.add(Dropout(0.2))
    model.add(TimeDistributed(Dense(256)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(TimeDistributed(Dense(1)))
    model.add(LeakyReLU(alpha=0.2))
    # model.summary()

    noise = Input(shape=(8, 1))
    img = model(noise)

    return Model(noise, img)


def build_discriminator():
    model = Sequential()

    model.add(Bidirectional(LSTM(256, return_sequences=True), input_shape=(8, 1)))
    model.add(Dropout(0.2))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Bidirectional(LSTM(256)))
    model.add(Dropout(0.2))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(RepeatVector(1))
    model.add(TimeDistributed(Dense(300)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(TimeDistributed(Dense(300)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(TimeDistributed(Dense(300)))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(TimeDistributed(Dense(1)))
```

```
    # model.summary()

    img = Input(shape=(8, 1))
    validity = model(img)

    return Model(img, validity)

class LSTMGAN:
    def __init__(self, learning_rate=0.00001):

        optimizer = Adam(learning_rate)  # , 0.4)

        # Build and compile the discriminator
        self.discriminator = build_discriminator()
        self.discriminator.compile(loss='binary_crossentropy',
                                   optimizer=optimizer,
                                   metrics=['accuracy'])

        # Build or load the generator
        self.generator = build_generator()

        # The generator takes noise as input and generates imgs
        z = Input(shape=(8, 1))
        img = self.generator(z)

        # For the combined model we will only train the generator
        self.discriminator.trainable = False

        # The discriminator takes generated images as input and determines validity
        valid = self.discriminator(img)

        # The combined model  (stacked generator and discriminator)
        # Trains the generator to fool the discriminator
        self.combined = Model(z, valid)
        self.combined.compile(loss='mean_squared_error', optimizer=optimizer)

    def train(self, epochs, batch_size=128, save_interval=10):
        # Load the dataset
        X_train = load_data()[0]

        # Rescale -1 to 1
        # X_train = X_train / 127

        batch_count = X_train.shape[0] // batch_size

        # Adversarial ground truths
        valid = np.ones((batch_size, 1, 1))
        fake = np.zeros((batch_size, 1, 1))

        g_loss_epochs = np.zeros((epochs, 1))
        d_loss_epochs = np.zeros((epochs, 1))

        for epoch in range(epochs):
            for index in range(batch_count):
                # ---------------------
```

```
                #  Train Discriminator
                # ----------------------

                # Select a batch of data
                training_batch = X_train[index * batch_size: (index + 1) * batch_size]

                # Sample noise and generate a batch of new images
                noise = np.random.normal(0, 1, (batch_size, 8, 1))
                generated_batch = self.generator.predict(noise)

                # Train the discriminator (real classified as ones and generated as zeros)
                d_loss_real = self.discriminator.train_on_batch(training_batch, valid)
                d_loss_fake = self.discriminator.train_on_batch(generated_batch, fake)
                d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

                # ----------------------
                #  Train Generator
                # ----------------------

                # Train the generator (wants discriminator to mistake images as real)
                g_loss = self.combined.train_on_batch(noise, valid)

                # save loss history
                g_loss_epochs[epoch] = g_loss
                d_loss_epochs[epoch] = d_loss[0]

            # Plot the progress
            print("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch, d_loss[0], 100 * d_loss[1], g_

        return g_loss_epochs, d_loss_epochs


learning_rate = 0.00001
epochs = 10
batch_size = 512
lstmgan = LSTMGAN(learning_rate)

start_time = time.perf_counter_ns()
g_loss, d_loss = lstmgan.train(epochs=epochs, batch_size=batch_size, save_interval=10)
end_time = time.perf_counter_ns()
time_formatted = str(datetime.timedelta(seconds=(end_time - start_time) * 10**(-9)))
# gives time in hh:mm:ss

print(f"Trained the model in {time_formatted}")

for i in range(0, lstmgan.epochs, 9):
    y_gen = lstmgan.generated_data[i][-1000:]
    x_gen = range(len(y_gen))

    y = lstmgan.training_data[i][-1000:]
    x = range(len(y))

    fig = plt.figure(figsize=(8,3))
    plt.plot(x, y, 'b', label='Training data', alpha=0.5)
    plt.plot(x_gen, y_gen, 'r', label='Generated data', alpha=0.5)
```

```
    plt.xlabel('Timestep')
    plt.ylabel('Normalized load')
    plt.title(f'{i} epochs')
    plt.legend()
    plt.show()
    fig.savefig(f'{i}_epochs_train_vs_gen.png')

for i in range(0, lstmgan.epochs, 9):

    y_gen = lstmgan.generated_average[i]
    x_gen = range(len(y_gen))

    y = lstmgan.training_average[i]
    x = range(len(y))

    fig = plt.figure(figsize=(10,3))
    plt.plot(x, y, 'b', label='Training data avg', alpha=0.5)
    plt.plot(x_gen, y_gen, 'r', label='Generated data avg', alpha=0.5)
    plt.title(f'{i} epochs')
    plt.legend()
    plt.show()
    fig.savefig(f'{i}_epochs_train_vs_gen_averages.png')

_, y_train, x_test, y_test = load_data()
start_time = time.perf_counter_ns()
y_pred, result_df = anomaly_detection(x_test, y_test, lstmgan.batch_size, lstmgan.discriminator)
end_time = time.perf_counter_ns()
time_formatted = str(datetime.timedelta(seconds=(end_time - start_time) * 10**(-9)))
# gives time in hh:mm:ss

print(f"Made predictions in {time_formatted}")
cm = confusion_matrix(y_test, y_pred)
plot_confusion_matrix(cm)
```

Bibliography

[1] Pierluigi Siano. Demand response and smart grids—a survey. *Renewable sustainable energy reviews*, 30:461–478, 2014. ISSN 1364-0321.

[2] Mohamed Abdel-Basset, Dr Nour Moustafa, and Hossam Hawash. Privacy-preserved generative network for trustworthy anomaly detection in smart grids: A federated semi-supervised approach. *IEEE transactions on industrial informatics*, pages 1–1, 2022. ISSN 1551-3203.

[3] Xiaohui Wang, Ting Zhao, He Liu, and Rong He. Power consumption predicting and anomaly detection based on long short-term memory neural network. In *2019 IEEE 4th International Conference on Cloud Computing and Big Data Analysis (ICCCBDA)*, pages 487–491, 2019. doi: 10.1109/ICCCBDA.2019.8725704.

[4] Yimin Zhou, Yanfeng Chen, Guoqing Xu, Qi Zhang, and Ludovic Krundel. Home energy management with pso in smart grid. In *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*, pages 1666–1670. IEEE, 2014. ISBN 9781479923991.

[5] Leping Zhang, Lu Wan, Yong Xiao, Shuangquan Li, and Chengpeng Zhu. Anomaly detection method of smart meters data based on gmm-lda clustering feature learning and pso support vector machine. In *2019 IEEE Sustainable Power and Energy Conference (iSPEC)*, pages 2407–2412. IEEE, 2019. ISBN 9781728149301.

[6] Ian Akyildiz, Su WY, Y. Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *Computer Networks*, 38:393–422, 03 2002. doi: 10.1016/S1389-1286(01)00302-4.

[7] Xiufeng Liu and Per Sieverts Nielsen. Streamlining smart meter data analytics. In *Proceedings of the 10th Conference on Sustainable Development of Energy, Water and Environment Systems*. International Centre for Sustainable Development of Energy, Water and Environment Systems, 2015. URL http://www.dubrovnik2015.sdewes.org/index.php. 10th Conference on Sustainable Development of Energy, Water and Environment Systems ; Conference date: 27-09-2015 Through 02-10-2015.

[8] Xiufeng Liu, Lukasz Golab, and Ihab F. Ilyas. Smas: A smart meter data analytics system. *2015 IEEE 31st International Conference on Data Engineering*, pages 1476–1479, 2015.

[9] Fadwa Maatug. Anomaly detection of smart meter data, 2021. URL https://hdl.handle.net/11250/2825119.

[10] Xiufeng Liu and Per Sieverts Nielsen. Regression-based online anomaly detection for smart grid data, 2016. URL https://arxiv.org/abs/1606.05781.

[11] Jui-Sheng Chou and Abdi Suryadinata Telaga. Real-time detection of anomalous power consumption. *Renewable and Sustainable Energy Reviews*, 33:400–411, 2014. ISSN 1364-0321. doi: https://doi.org/10.1016/j.rser.2014.01.088. URL https://www.sciencedirect.com/science/article/pii/S1364032114001142.

[12] Soo Wan Yen, Stella Morris, Morris Abraham Gnanamuthu Ezra, and Tang Jun Huat. Effect of smart meter data collection frequency in an early detection of shorter-duration voltage anomalies in smart grids. *International Journal of Electrical Power & Energy Systems*, 2019.

[13] Jiahua Liu, Shang Wu, Wanwan Cao, Yang Guo, and Shuai Gong. Smart grid data anomaly detection method based on cloud computing platform. In Xingming Sun, Xiaorui Zhang, Zhihua Xia, and Elisa Bertino, editors, *Artificial Intelligence and Security*, pages 338–345, Cham, 2021. Springer International Publishing. ISBN 978-3-030-78609-0.

[14] B. Joyce Beula Rani and L Sumathi M. E. Survey on applying gan for anomaly detection. In *2020 International Conference on Computer Communication and Informatics (ICCCI)*, pages 1–5. IEEE, 2020. ISBN 1728145147.

[15] Federico Di Mattia, Paolo Galeone, Michele De Simoni, and Emanuele Ghelfi. A survey on gans for anomaly detection, 2019. URL https://arxiv.org/abs/1906.11632.

[16] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of systems architecture*, 98:289–330, 2019. ISSN 1383-7621.

[17] Ben Zhang, Nitesh Mor, John Kolb, Douglas S. Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward Lee, and John Kubiatowicz. The cloud is not enough: Saving IoT from the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015.

USENIX Association. URL https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/zhang.

[18] Ben Zhang, Nitesh Mor, John Kolb, Douglas S. Chan, Ken Lutz, Eric Allman, John Wawrzynek, Edward Lee, and John Kubiatowicz. The cloud is not enough: Saving IoT from the cloud. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Santa Clara, CA, July 2015. USENIX Association. URL https://www.usenix.org/conference/hotcloud15/workshop-program/presentation/zhang.

[19] Rituka Jaiswal, Fadwa Maatug, Reggie Davidrajuh, and Chunming Rong. Anomaly detection in smart meter data for preventing potential smart grid imbalance. In *2021 4th Artificial Intelligence and Cloud Computing Conference*, AICCC '21, page 150–159, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384162. doi: 10.1145/3508259.3508281. URL https://doi.org/10.1145/3508259.3508281.

[20] Facebook Prophet. Forecasting time series. https://facebook.github.io/prophet/.

[21] Rituka Jaiswal, Reggie Davidrajuh, and Chunming Rong. Fog computing for realizing smart neighborhoods in smart grids. *Computers*, 9:76, 09 2020. doi: 10.3390/computers9030076.

[22] Redowan Mahmud, Ramamohanarao Kotagiri, and Rajkumar Buyya. *Fog Computing: A Taxonomy, Survey and Future Directions*, pages 103–130. Springer Singapore, Singapore, 2018. ISBN 978-981-10-5861-5. doi: 10.1007/978-981-10-5861-5_5. URL https://doi.org/10.1007/978-981-10-5861-5_5.

[23] Jayant Baliga, Robert Ayre, Kerry Hinton, and Rodney Tucker. Green cloud computing: Balancing energy in processing, storage, and transport. *Proceedings of the IEEE*, 99:149 – 167, 02 2011. doi: 10.1109/JPROC.2010.2060451.

[24] Anne-Cecile Orgerie, Marcos Dias de Assuncao, and Laurent Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Comput. Surv.*, 46(4), mar 2014. ISSN 0360-0300. doi: 10.1145/2532637. URL https://doi.org/10.1145/2532637.

[25] Rituka Jaiswal, Reggie Davidrajuh, and S. M. Wondimagegnehu. *Fog Computing for Efficient Predictive Analysis in Smart Grids*. Association for Computing Machinery, New York, NY, USA, 2021. ISBN 9781450385756. URL https://doi.org/10.1145/3487923.3487937.

[26] Store Norske Leksikon. Maskinlæring / Machine learning. https://snl.no/maskinl

[27] Google Developers. Overview of GAN structure. https://developers.google.cn/machine-learning/gan/gan_structure.

[28] Composing Melodies with Bidirectional LSTM Generative Adversarial Networks. Source Code. https://github.com/vee-upatising/Music-Composition-GAN.

[29] Keras. The Python deep learning API, Adam optimizer documentation. https://keras.io/api/optimizers/adam/.

[30] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014. URL https://arxiv.org/abs/1412.6980.

[31] Rituka Jaiswal, Antorweep Chakravorty, and Chunming Rong. Distributed fog computing architecture for real-time anomaly detection in smart meter data. pages 1–8, 08 2020. doi: 10.1109/BigDataService49289.2020.00009.

[32] Ausgrid. Shared Data for Research. https://www.ausgrid.com.au/Industry/Our-Research/Data-to-share/Solar-home-electricity-data, .

[33] Ausgrid. Datasets with informational document. https://cdn.ausgrid.com.au/-/media/Documents/Data-to-share/Solar-home-electricity-data/Solar-home-half-hour-data—1-July-2012-to-30-June-2013.zip?rev=de594e37789744738fe747c37e1e67bfhash=FACD8C1C02A7D10E17059946F7D65EF6, .

[34] Datamadness. TensorFlow 2 - CPU vs GPU Performance Comparison. https://datamadness.github.io/TensorFlow2-CPU-vs-GPU.