From Compressive Sensing to Machine Learning in Smart Grids

A Dissertation

by

ABDULAZIZ SAAD S. A. AL-QAHTANI

Submitted to the Graduate and Professional School of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Joseph Boutros
Co-Chair of Committee,	Robert Balog
Committee Member,	Othmane Bouhali
Head of Department,	Efstratios N. Pistikopoulos

December 2021

Major Subject: Energy

ABSTRACT

Traditional power grids are a single-layered physical system, while smart grids are an extension of traditional power grids that are cyber-physical networks, and the main difference is smart grids include an information layer. There is a huge amount of information being managed within recent smart grids, and the decentralized power generation adds an extra level of uncertainty to smart grids. The standard methods of monitoring and security available cannot work as expected when collecting and analyzing the large amount of data presented from different parameters in the power network.

Compressive sensing is a signal processing tool that is used to monitor single and simultaneous fault locations in smart distribution and transmission networks, to detect harmonic distortions, and to recognize patterns of partial discharge. Compressive sensing reduces the measurement cost and the management cost since it can detect or rebuild a signal from very few samples. In this thesis, we propose to design and implement the fault detection via a feedforward neural network using similar regularizations as in compressive sensing. We shall use the adaptivity of neural networks to tackle with state changes in the smart grid, proving the scalability and the decentralized capability of a neural network for fault detection in the grid.

Two codes have been created against two different databases, and it was found that indeed, a feedforward autoencoder would be great at fault detection, however, many things should be considered prior to implementing it on a large scale. The most important part of any autoencoder generation is a good dataset.

ACKNOWLEDGEMENTS

I would like to begin by saying that without the people mentioned in this list, I would not be able to be where I am today. I would like to begin by thanking my parents, my wife, and my sister Shaikha, for being on my journey of education since I was a child. I would also like to thank Texas A&M for allowing myself to be a part of this program. I would like to thank my thesis committee, Robert Balog and Othmane Bouhali. Othmane Bouhali has been a great mentor in my undergrad studies, and I am genuinely grateful. And finally, I would like to deeply thank Dr. Joseph Boutros for his never-ending help since I was an undergrad, and I will be sure to remain his student for as long as I can. Thank you all.

CONTRIBUTORS AND FUNDING SOURCES

Contributors

This work was supported by a thesis committee consisting of Professor Joseph Boutros (chair), Professor Robert Balog (co-chair) and Professor Othmane Bouhali (member). All other work conducted for the thesis was completed by the student independently.

Funding Sources

No funding sources.

TABLE OF CONTENTS

TABLE OF FIGURES

INTRODUCTION

With newer technologies older technologies become obsolete. The technological boom of the 20th century has been continuously evolving, and in turn infrastructures must be modified in a way that follows the technological evolution. The "upgrade" of already existing systems is extremely challenging, as criteria of which an upgrade is feasible has many angles, whether it was economically sensible, or would be safer to operate.

The most important source of energy available widely is electricity, and it is only natural that the ways of which we distribute the most important source of energy will develop over time. Since renewable energies have been slowly becoming competitive economically, we must account for them in our electrical grid. With traditional power grids, renewable energies cannot be utilized as the power generation is centralized, and with that we must have smart grids.

Smart grids are important for monitoring as well as distributing electricity. With smart grids, it is possible to have increased communication between devices and allowing remote control for the utility provider. This paper is concerned with the sensing aspect of power grids. The infrastructure of the traditional power grids does not have many sensors on the power lines, and hence the accuracy of error detection is low. Smart grids on the other hand have more sensors placed on the lines, and rerouting power is less of a challenge thus making error detection higher.

The amount of information to be handled is extremely dense, and the type of information can be digital or analog. Examples of digital signals can be circuit breaker status, while analog signals could be voltages or currents. Compressive sensing is the technology currently employed mostly in the field, and while it is abundant, using machine learning for this task is favorable as machine learning is designed to handle such tasks.

Compressive sensing is relatively a recent topic and could be an answer to the issue of addressing the new changes with traditional power grids, however, machine learning has been

consuming most fields. It could be the proper answer to the smart grid information collection as well as analyzing information gathered.

For machine learning to be used, the topology of the system must be found for the smart grid. The topology as well as fault detection has already been researched for compressed sensing look at [2], but for this research, we aim to transform the topology from compressive sensing to machine learning for smart grids.

QUICK LITERATURE REVIEW

There have been many publications discussing smart grids, compressive sensing, and even machine learning as of recently. To establish the understanding to reach what is the work to be done, we must understand what has been already founded.

A. Traditional Power Grid

The traditional power grid is essentially the interconnection of various power systems like the likes of transformers, transmission lines, and different types of loads. Long transmission lines transfer the power from a far location from the power consumption area. [3] There is a lot of differences between the traditional grid and smart grids.

B. Smart Grids

Smart grids (SGs) are electrical grids that can intelligently integrate the behavior and actions of all parameters, whether it is voltage, current, and the likes of it. The smart grid also can intelligently observe different users of the electrical grid [2], and this observation can be at a city level, to a country level, and even to a continental level.

Traditional power grids are a single-layer physical system, while smart grids are huge cyber-physical networks. SGs have many parameters, and some of them are load consumption, reactive powers, voltage, current, and many more.

C. Traditional Grids Vs. Smart Grids

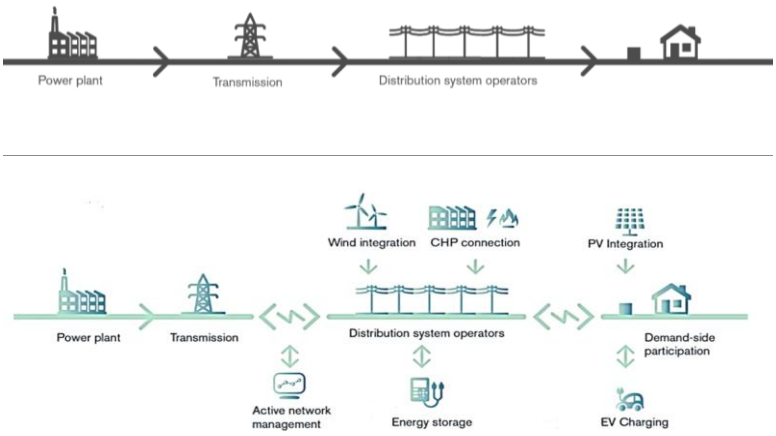

Figure 1. Traditional Grid VS. Smart Grid [4].

The differences between them are many, but there are key differences that can be noted and contrasted between the two technologies. [3]

Technology: Traditional power grids are electromechanically operated, while smart grids are digital. This means that the smart grid has more communication between devices thus allowing remote control and self-regulation.

Distribution: Traditional power grids have a one-way distribution while smart grids have a two-way distribution, meaning that traditional power grids cannot facilitate renewable energies, while smart grids are able to have a primary power plant, in addition to secondary providers. This means that if an individual has access to alternative energy sources, the user can feed back power to the grid.

Generation: Traditional power grids have centralized power generation, while smart grids have distributed power generation. For the renewable energy aspect, it must be noted that having the energy generation to be centralized, means that you cannot incorporate multiple sources of energy into the grid. On the other hand, power can be distributed from multiple plants for smart grids, allowing to balance loads, decrease peak time strains, and limit the number of outages.

Sensors: There are few sensors in traditional power grids, while there are sensors throughout the smart grid. This also follows up closely to the idea of utilizing machine learning for smart grids. Compressive sensing in smart grids have aided us this far, but we are not achieving the maximum potential of smart grids, and hence this brings up to the next point, which is compressive sensing.

D. Compressive Sensing

Compressive sensing is a technique that uses data compression while accounting for the sparsity of the electrical consumption pattern in a transformation basis thus achieving sub-Nyquist compression. As the power consumption data has a varying sparseness level, the choice of transformation basis influences the compression performance significantly, adding the compression level as well as the reconstruction error. [1]

Compressive sensing for smart grids has the benefit of fault detection, but for that a topology must be identified for the smart grid.

There are a few problems with implementing compressive sensing such as the computational complexity of data collection and analysis procedures in large scale smart grids, the effect of distributed generation systems, and the uncertainty of system states and parameters caused by the behavior of loads. [2]

Due to the sparse nature for the optimization problem of smart grid topology identification, the topology can be interpreted as a Sparse Recovery Problem (SRP), and hence it could be efficiently solved with SRP solvers, or optimization-based algorithms. [2]

E. Machine Learning

Machine learning (ML) is where artificial intelligence (AI) is used to automatically learn and improve from experience gained from training parameters without being programmed.

ML is mainly used in processes where human operation is impossible, as the information is far too huge. The decision making ultimately is supposed to be completely controlled by AI, and hence the end goal for most machine learning systems is complete automation.

If smart grids could be completely operated by machine learning software, the end user would benefit from a more robust grid, as well as appropriate beneficial data could be available for the utility provider. This is due to the error detection capability of machine learning.

The utility provider will also in general have lower costs as less manpower is meant to operate the machine learning algorithms. While this is the most evident gain for the utility provider, long term benefits can be significant.

AUTOENCODERS

Machine learning has many forms, whether it was supervised, or unsupervised, it must be considered when the application of machine learning is questioned. The architecture of which the machine learning is designed is also an important consideration, and thus we will discuss autoencoders.

Autoencoders are the simplest deep learning architecture, where they are a specific type of feedforward neural networks. The input is compressed into a lower-dimensional code, and then the output is reconstructed from the compressed code. Auto encoders consist of an encoder, a code, and finally a decoder.[5]

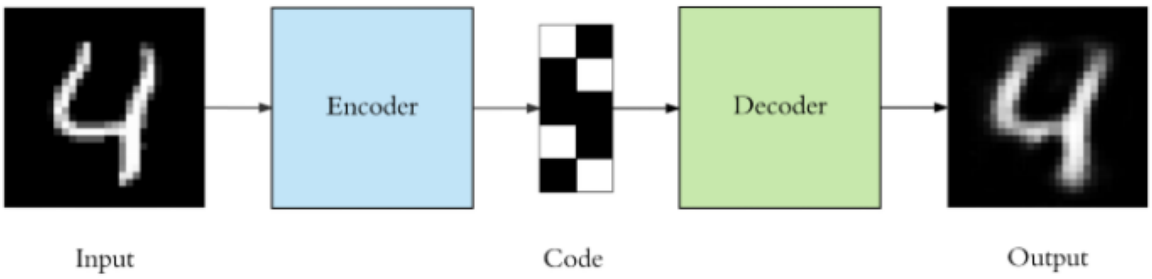

Figure 2. A representation of an autoencoder [5].

A. Autoencoder Architecture

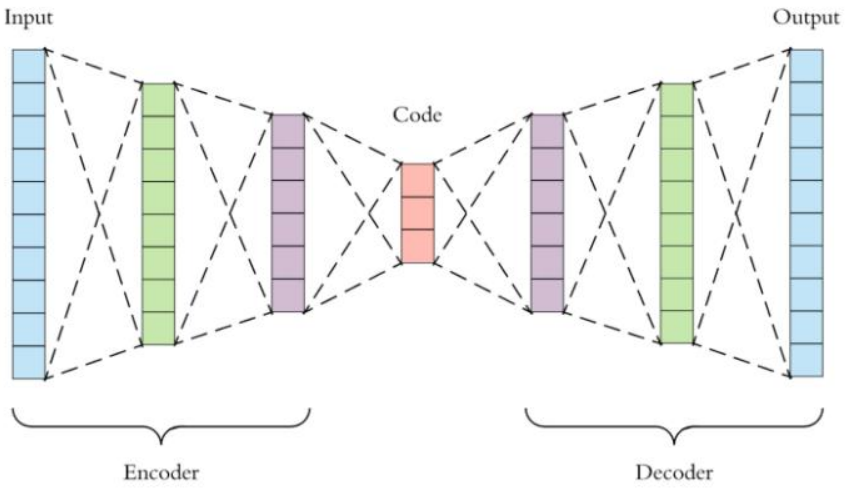

Figure 3. Architecture of a Symmetric, stacked, Autoencoder [5].

The main goal of an autoencoder is to get an output which matches the input signal given. This is achieved through compressing the input with an encoder that is a fully connected artificial neural network (ANN), thus creating the code. Then by only using the code, we achieve an output by running the code on a similarly fully connected artificial neural network. [5]

Each layer has a difference "size" and hence the code size is directly related to the compression rate. The smaller the code, the more compressed the input is. The number of layers is the choice of the user, but it is important to note that the number of layers does not account for the input and output layer. The middle layers are called "hidden layers."

A loss function is also incorporated, and it could be either mean squared error, or binary cross entropy. [6] Cross entropy is restricted for input values between the range of [0,1] and for any input that is not within that range, mean squared error is utilized. [5]

B. Autoencoder Number of Nodes in the Hidden Layers

The number of nodes in the hidden layers is a parameter that is completely under the control of the user. It is important to note that this could be the difference between different autoencoder types.

C. Sparse and K-Sparse Autoencoders

When an autoencoder has a single hidden layer, it is called a “sparse” autoencoder. With the advancements of sparse autoencoders, k-sparse autoencoders had been developed, where “k” is the number of neurons with the highest activation functions ignoring other functions by using ReLU activation functions and adjusting the threshold to obtain the largest neurons. This in turn tunes the value of k to obtain the best sparsity level for the dataset. [6]

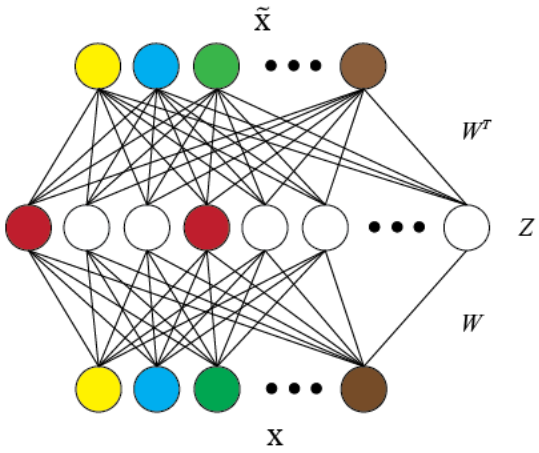

Figure 4. A sparse autoencoder [6].

D. Stacked Autoencoders

A stacked autoencoder could be viewed as a group of sparse autoencoders, where the input and output layers are the same, but there are multiple hidden layers. These hidden layers have their

outputs connected to other hidden layers to create a "code" which is used to extrapolate the output and recreate the input.

The stacked auto encoder could be symmetric or asymmetric, as we have control over these different parameters in the hidden layer, and hence the following figure could be a stacked auto encoder example, however, unlike figure 3, this one is asymmetric.

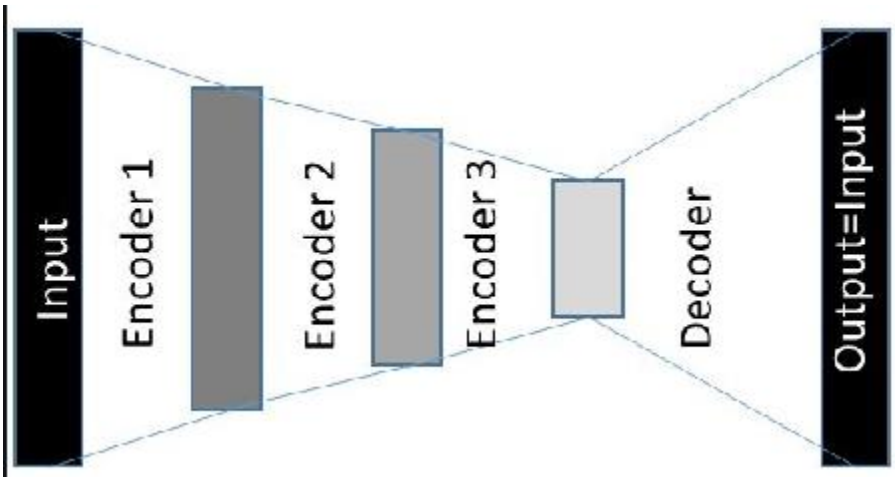

Figure 5. Asymmetric stacked autoencoder [7].

As it could be noted from the figure above, this stacked autoencoder has three hidden layers for encoding, and only a single layer for decoding.

E. Variational Autoencoders

A variational autoencoder's input is mapped to a distribution instead of being mapped to a fixed vector. The only difference between a variational and a normal autoencoder is the hidden layer has two different vectors where one represents the mean of the distribution, while the other represents the standard deviation of the distribution.

The loss function consists of two terms, the first one represents the reconstruction loss, and the other regularizes the autoencoder. [6]

$$li(\theta,\phi) = -Ez\sim q\theta(z|xi)\,[\log p\phi(xi|z)] + KL(q\theta(z|xi)||p(z))$$

Figure 6. The loss function of a variational auto encoder [6].

Kullback-Leibler (KL) is the divergence between the encoder's distribution qθ(z|x) and p(z). The divergence is the measurement of lost information when q is used to represent P. [6]

DATA SETS

In machine learning, when an autoencoder is to be applied, the information provided must be meaningful to our purpose. For this paper two data sets have been chosen. One is a fourteen-bus data set, and the other is a Fifty-Six-Node.

A. The Fourteen-Bus Data Set

For the fourteen-bus data set, the following figure represents how the data is measured.

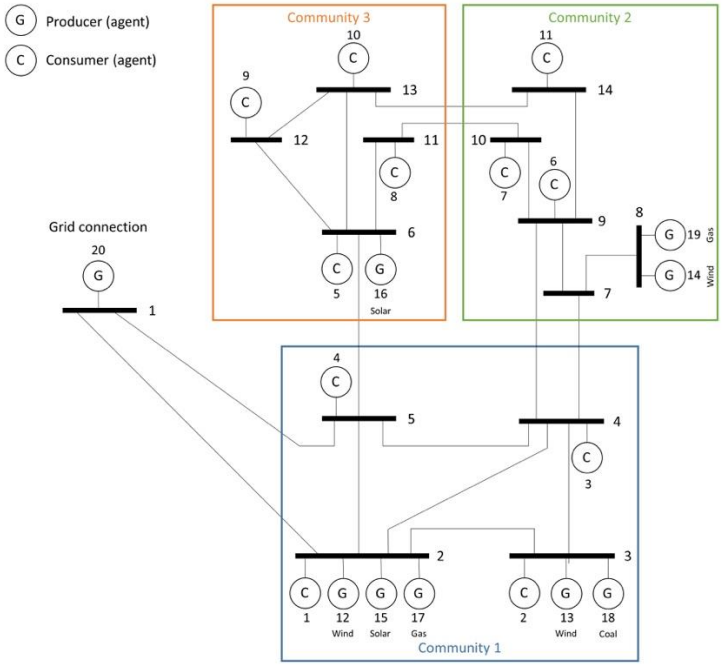

Figure 7. The fourteen-bus system layout [8].

The data has been recorded from Australia and is real. It spans a year of measurements in 30-minute intervals from July of 2012 to June of 2013. The reason this data was chosen for this

paper is to understand how a real system should operate if an autoencoder were to be applied on it.

B. The Fifty-Six-Node Data Set

For the Fifty-Six-Node data set, the following figure represents how the data is measured.

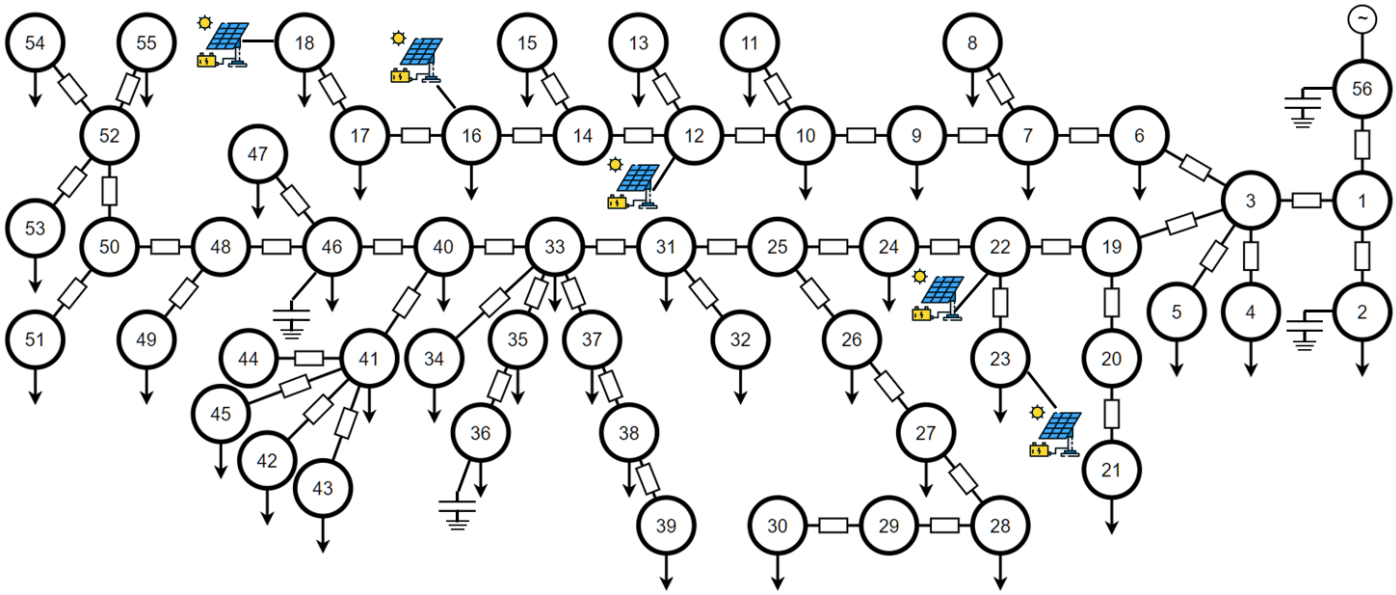

Figure 8. Fifty-Six-Node data set [9].

The data has been generated using MATLAB. The data spans a year of measurements for each of the above nodes, one measurement for each hour. The reason this data was chosen for this paper is to apply an autoencoder on it.

DATA PREPARATION AND ANALYSIS

When training any data set, the features must be transformed by utilizing certain techniques for certain situations. The reason for this is to improve both performance and stability of any model that will be trained.

There are four common normalization techniques that is generally useful to any machine learning training. [10]

1- Scaling to a range.

2- Feature Clipping.

3- Log Scaling.

4- Z-Score.

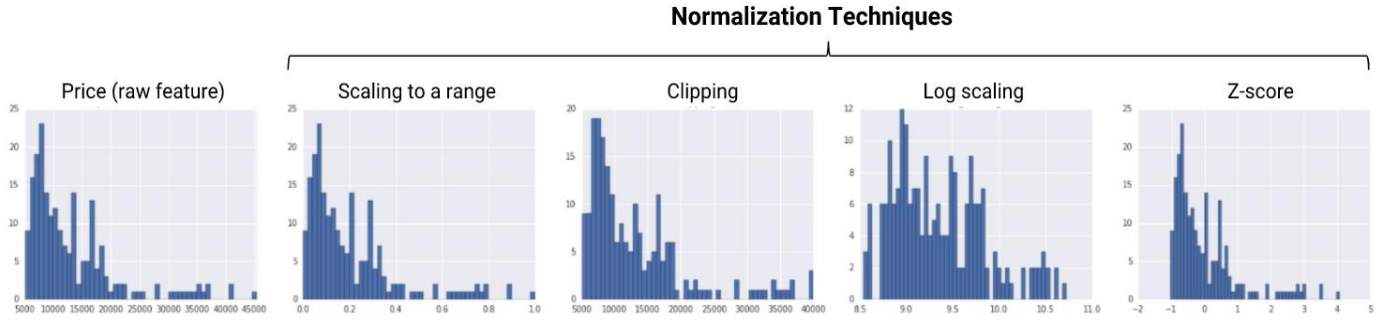

Figure 9. Normalization techniques as illustrated by Google [10].

A. Scaling to a Range

When a data set range is known, such as the normal height of humans, we can limit the height constraints to 50cm to 260cm. Hence scaling to a range is applicable when the following two conditions are met.

1- The approximate upper and lower bounds are known, with few to no outliers.

2- The data is distributed uniformly across the specified range.

Therefore, it could be noted that as per those two conditions, that scaling to a range for a data set containing a bias towards a certain extreme, that scaling to a range will compromise the training. An example for a biased data set can be seen as the income of a society. As there are far fewer people at the top than there are in the middle and bottom.

B. Feature Clipping

When a data set contains an extremely long "tail" that data could be considered as "outliers". The solution for such an issue is to simply clip that data. Clipping the data would both provide us with a more meaningful (remaining) data set, as well as reduce the outliers that would not aid the training of our code.

Feature clipping could be used in an example where the extreme values whether low or high are pointless to us. An example is temperature where anything less than -10 Celsius and +50 Celsius is pointless to consider depending on the place where those readings are taken.

C. Log Scaling

Log scaling is used to "scale" a data set. This scaling is done by compressing the data set such that the range between the points are narrowed down from a wide range to a narrow one.

Log scaling is useful when a few points are assigned to most of the values in the data set, while the majority has many points. The power law distribution is known as this distribution, and it can be shown in an example by Google below.

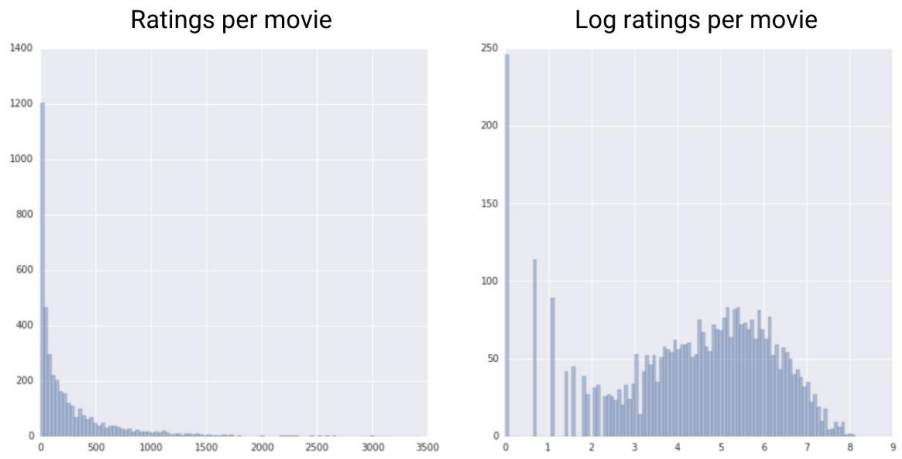

Figure 10. Ratings per movie vs log ratings per movie by Google [10].

Log scaling manipulates the distribution such that it improves the linear model's performance.

D. Z-Score

Z-score ensures that the features distributions has mean = 0 and standard deviation = 1. It is useful when there are outliers in the data set, but not so extreme that the data set requires clipping.

E. Summary of Data Preparation

Normalization Technique	Formula	When to Use
Linear Scaling	$x' = (x - x_{min})/(x_{max} - x_{min})$	When the feature is more-or-less uniformly distributed across a fixed range.
Clipping	if x > max, then x' = max. if x < min, then x' = min	When the feature contains some extreme outliers.
Log Scaling	x' = log(x)	When the feature conforms to the power law.
Z-score	x' = (x - μ) / σ	When the feature distribution does not contain extreme outliers.

Figure 11. A Summary by Google [10].

F. Looking at our first Data set

By plotting the entire first bus of the data set, it can be noted that there is no time pattern that will allow us to know what to do with the data set.

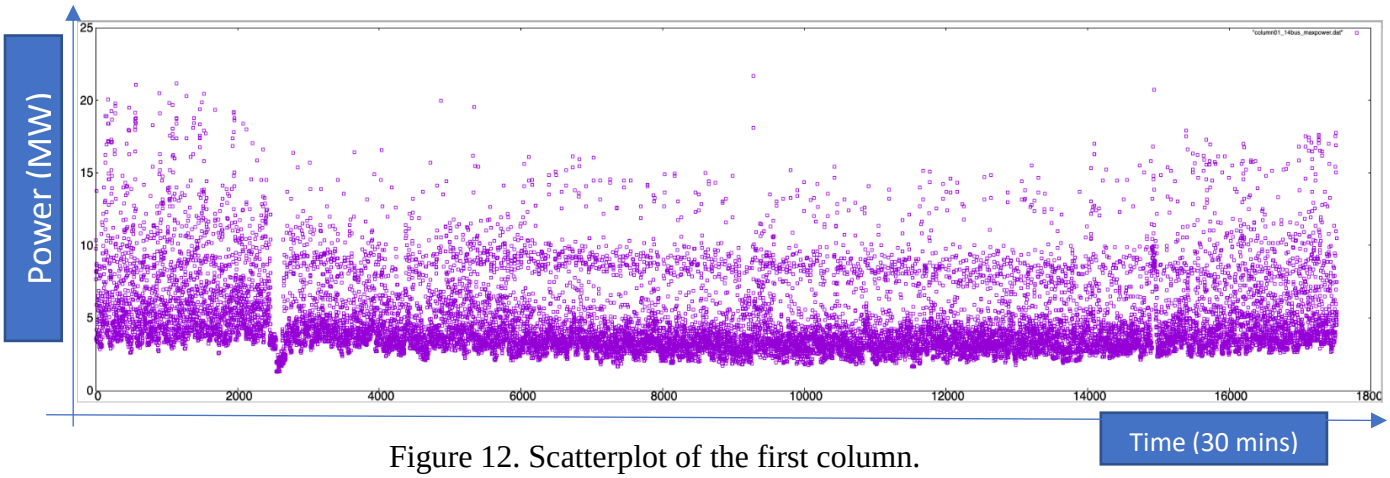

Figure 12. Scatterplot of the first column.

However, when the histogram is considered, we can understand that the data distribution indeed has a recognizable pattern.

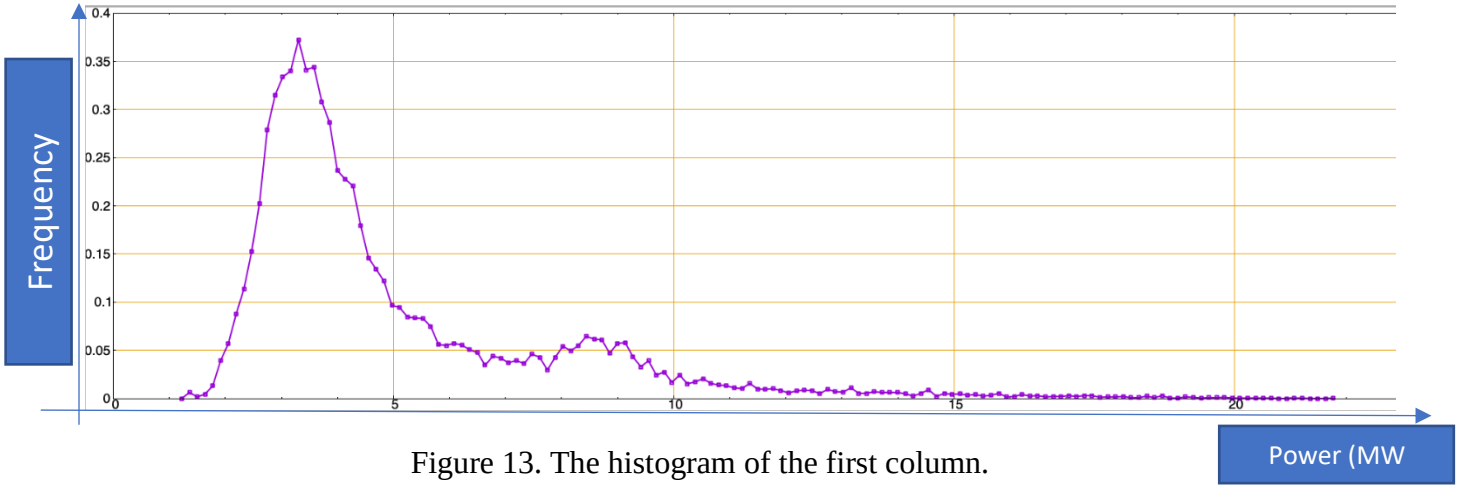

Figure 13. The histogram of the first column.

The above plot can be estimated as a Fréchet, and hence, utilizing a mixture density network can allow us to produce a neural network that could estimate this shape.

MIXTURE DENSITY NETWORKS (MDN)

Our first data set is using real data (14-Bus data set), and in our day-to-day life, real data is generally noisy. While noise is not preferred, it is important towards building our model. A mixture density network is special because not only does it predict the expected value of a target, but it also predicts the underlying probability distribution. [11]

A. Mixture Density Network Code

A mixture density network code is simply a neural network that can predict a distribution such that the neural network is mapped to each parameter that determines the distribution.

A Fréchet distribution requires three parameters to be mapped as a Generalized Extreme Value (GEV) distribution.

```
def mdn_cost(ksi, sigma, mu, power):
    ##value=-math.log(alpha/s*pow((y-m)/s,-1-alpha)*math.exp(-pow((y-m)/s,-alpha)))
    dist = tfp.distributions.GeneralizedExtremeValue(loc=mu, scale=sigma, concentration=ksi)
    cost=-dist.log_prob(power)
    return tf.reduce_mean(tf.where(tf.math.is_nan(cost), 100000., cost))
```

Figure 14. Defining the GEV in the code.

There are three important parameters to estimate our GEV. As it could be seen, they are ξ, σ, and μ. The reason we mention these three parameters, as it is important towards building our neural network. The neural network will operate in the following manner.

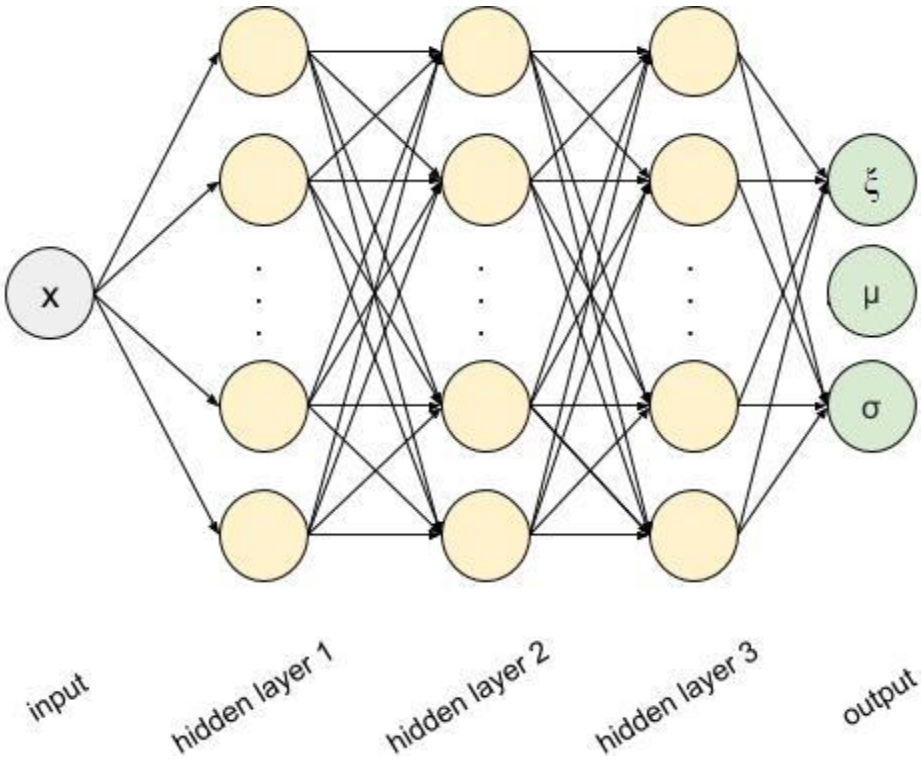

Figure 15. Our Neural Network for Estimating a GEV.

The reason we need to find the three variables is because once we plug in the values in the equation of a GEV, the estimation will be plotted.

$$F(x;\mu,\sigma,\xi) = \exp\left\{-\left[1+\xi\left(\frac{x-\mu}{\sigma}\right)\right]^{-1/\xi}\right\}$$

Figure 16. The GEV equation [12].

Once all of this is considered, it is now time to construct the code to predict the values of ξ, σ, and μ through our neural network.

It is important to note that a cost function is used, and the purpose of using a cost function is such that we know how well our model is fitting. The cost function is minimized through the gradient descent algorithm, which allows to increase how well our model fits.

```
epochs = 1000
batch_size = 2000
learning_rate = 0.00001
InputLayer = Input(shape=(1,))
Layer_1 = Dense(12,activation="tanh")(InputLayer)
Layer_2 = Dense(12,activation="tanh")(Layer_1)
ksi = Dense(1, activation=lambda x: tf.nn.elu(x) + 1)(Layer_2) # GEV: this replaces the alpha of Fréchet
sigma = Dense(1, activation=lambda x: tf.nn.elu(x) + 1)(Layer_2) # GEV: this replaces the s of Fréchet
mu = Dense(1, activation=lambda x: tf.nn.elu(x) + 1)(Layer_2) # GEV: this replaces the m of Fréchet
##ksi= Dense(1, activation='linear')(Layer_2) # GEV: this replaces the alpha of Fréchet
##sigma = Dense(1, activation='linear')(Layer_2) # GEV: this replaces the s of Fréchet
##mu = Dense(1, activation='linear')(Layer_2) # GEV: this replaces the m of Fréchet
lossF = mdn_cost(ksi,sigma,mu,InputLayer)
model = Model(inputs=[InputLayer], outputs=[ksi, sigma, mu])
model.add_loss(lossF)
adamOptimizer = optimizers.Adam(learning_rate=learning_rate)
model.compile(optimizer=adamOptimizer,metrics=['mse'])
history_cache = model.fit([power_arr], #notice we are using an input to pass the real values due to the inner workings of keras
                          verbose=0, # write =1 if you wish to see the progress for each epoch
                          epochs=epochs,
                          batch_size=batch_size)
print('Final cost: {0:.4f}'.format(history_cache.history['loss'][-1]))
ksi_pred, sigma_pred, mu_pred = model.predict(power_test) # our model has 1 input
print(ksi_pred, sigma_pred, mu_pred);
```

Figure 17. The Neural Network That Finds the GEV Parameters.

The batch size has been selected as we have 17,520 different inputs to our model, and having to go through 1000 epochs, it is preferred that we take batches of 2000 to reduce the computation time.

After the neural network has finished compiling, the output is as follows:

```
Final cost: 1.7432
[[0.49454015]] [[0.89197576]] [[3.831363]]
```

Figure 18. Final results of MDN.

It is important to note that our cost is not almost zero, which means that this model could be further optimized, however, we can still use the values gained by the neural network and compare to the original data. The reason why the final cost is not stabilizing near zero is due to the random initialization. If somehow the initialization could be properly controlled our results could be more consistent.

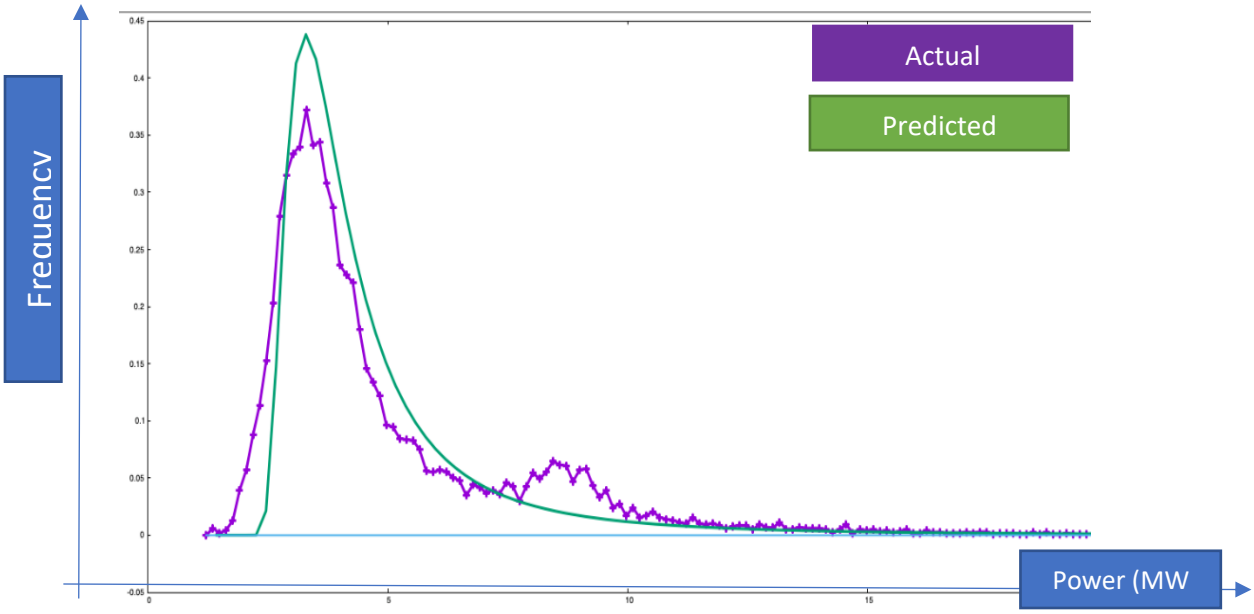

Figure 19. Our GEV Distribution vs. Original Distribution.

While the predicted distribution is not a perfect fit to our model, it could be noted that this method is useful when applying a prediction to a noisy real-world data. The strength of this code is that most distributions could be predicted such that the neural network is adjusted in a way where the parameters of the distributions are mapped as the output.

The reason the model is not fitting perfectly is due to the nature of the real-world data being of a random distribution. Furthermore, the initialization of each run of the code will result in different results for each parameter. This should be controlled such that the model can robustly predict each distribution with great confidence.

The great thing about this code is most distributions are not neat, and this code allows to map a neural network to recreate a distribution no matter how complex it is. If the model could be estimated as a GEV, then this code can be utilized to approximate the plot of the data set.

Looking at our second data set, we can see that the node system operates much like a smart grid, where we have different nodes that supply/take energy in MW from different places. This means that we can apply an autoencoder that will behave like an autoencoder in a smart grid.

A. Data Set

In our data we have 56 different columns, and for our use, using only half is more than enough to prove the theory. Hence, we took 24 columns, which is one more than half.

```
names=[]
for i in range(24):
    names.append("Power"+str(i))
gridload_train = pd.read_csv("R1-1247-2and4-balanced-phase_24cols_commas.csv")
print(gridload_train.head())
```

```
       Power0       Power1       Power2       Power3      Power4  \
0  2101.753486  4806.526185  2944.672121  1425.134979  383.408132
1  1885.534319  4280.180006  2962.816176  1387.156142  260.646728
2  1696.527510  4156.099292  2979.608335  1465.743428  300.544737
3  1707.297935  4037.543541  2559.597910  1342.301931  338.363139
4  1680.249659  3764.174422  2516.934635  1231.732641  296.171024

       Power5      Power6       Power7      Power8      Power9  ...  \
0  1462.894087  302.356003  3922.988590  856.847755  241.211486  ...
1  1333.142426  234.803734  3498.905373  703.896518  271.579258  ...
2  1414.762000  285.302719  3132.758634  893.898141  321.178674  ...
3  1196.421604  174.676827  3191.605383  739.918897  199.044075  ...
4  1031.688703  192.574018  3159.037114  887.011503  255.517337  ...

      Power14     Power15      Power16      Power17      Power18      Power19  \
0  347.036846  847.787726  1414.003506  3523.241320  2262.496785  2840.068287
1  253.308792  724.403463  1433.792228  2715.127106  1650.030346  2579.961264
2  278.794158  743.084853  1329.694014  2979.326924  2060.278287  2802.202110
3  296.393716  846.563643  1097.026759  2456.925501  1780.437862  2536.001162
4  280.905085  773.662307  1277.413687  2220.022196  1828.193533  2553.043337

       Power20      Power21     Power22     Power23
0  1507.159442  1560.277109  963.592056  969.067573
1  1657.742017  1261.283219  862.675838  938.651303
2  1198.735631  1454.240667  806.067003  642.091518
3  1277.523045  1366.458105  842.385287  729.194376
4  1195.726387  1252.199165  849.859337  650.359908
```

Figure 20. Our data set with only 24 columns.

Contrary to the first data set, this data set is more correlated, and this correlation can be found out by plotting the entire data set in the time domain. We have taken all the columns and plotted against a “time” domain generated by ourselves.

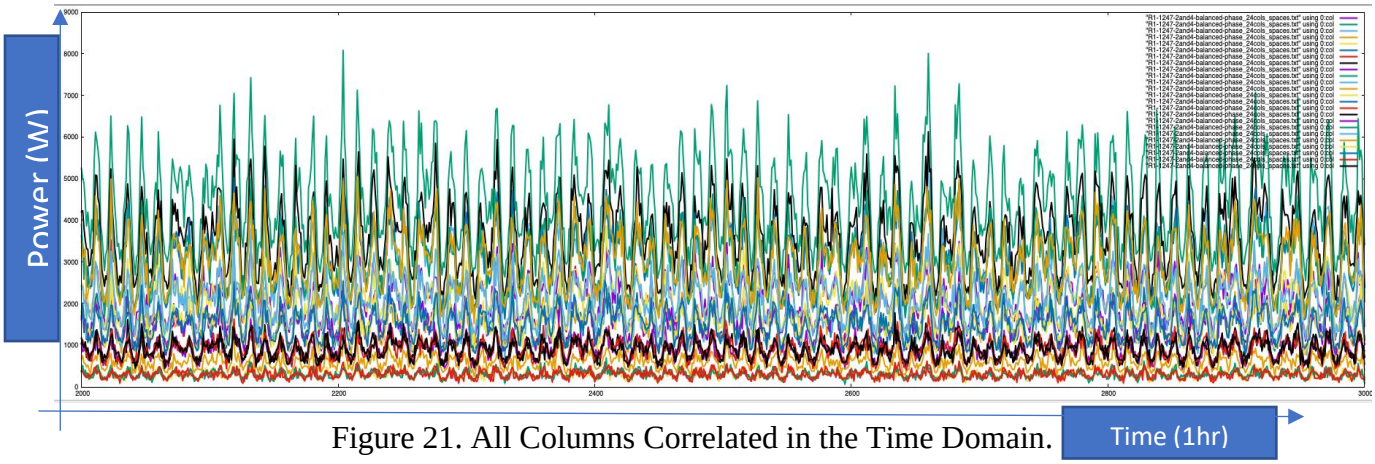

Figure 21. All Columns Correlated in the Time Domain.

Unlike the first data set, we can see that this data is much more organized than the original information. This is a great indication that upon utilizing an autoencoder from TensorFlow, we can find a great fit from the generated prediction

The reason that data set correlation is important for our autoencoder, is that not all information is useful towards our autoencoder. If we have too much noise in our system, then our autoencoder will not be able to detect a pattern, and hence the prediction as well as the fitting of the model will not occur in the intended way.

Remembering our first data set, the reason we needed to create an MDN, is due to the data set's randomness in and of itself. By comparing figure 12 to figure 21, it can be easily seen that figure 21 would be a better "fit" for our TensorFlow autoencoder.

To have a better idea on how the data set is distributed, for each column, a histogram was created, resulting in 24 different histograms. Upon inspecting each different histogram, again, it was shown that they mostly have a Fréchet shape, or a GEV function to be general.

This means that while our original code will work, since this data set is more correlated, the utilization of TensorFlow, and deep learning will be a better fit for this data set.

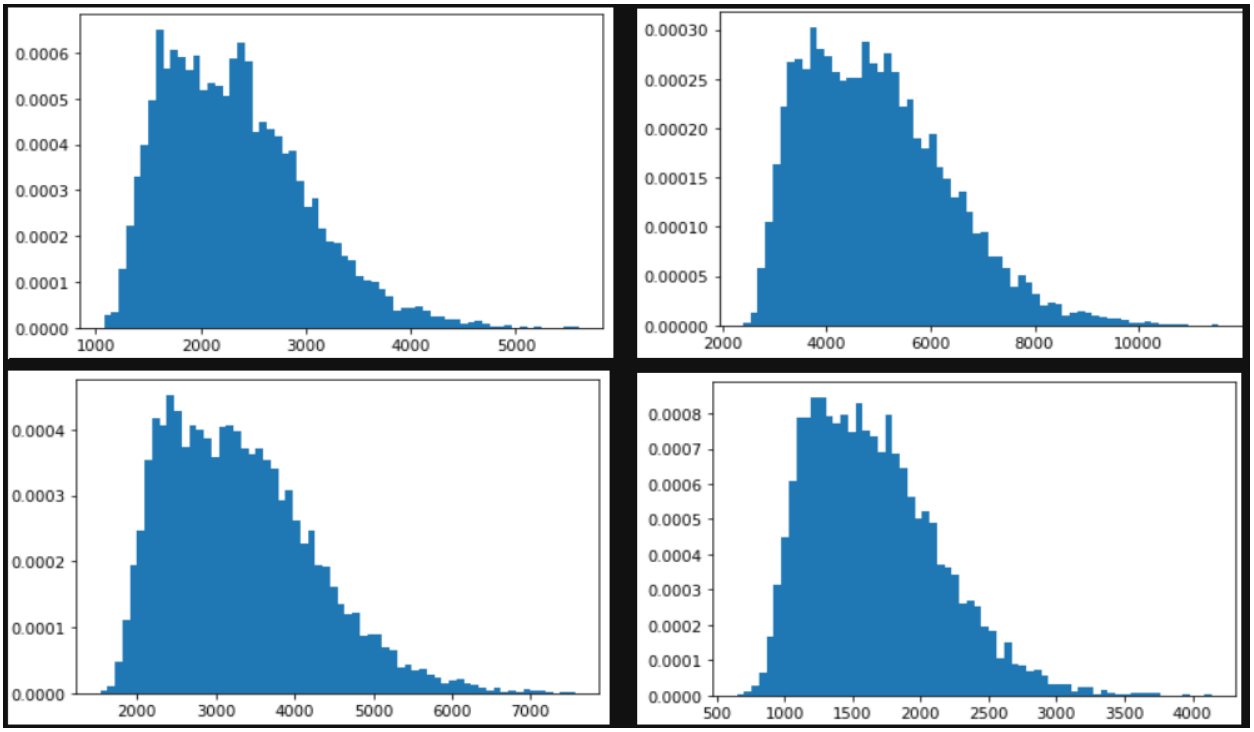

Figure 22 Figure 22 The First Four Columns for Power Histograms.

B. Building the Autoencoder

```
[12]: latent_dim =6  ## 24 16 10 worked very well
      class Autoencoder(Model):
        def __init__(self, latent_dim):
          super(Autoencoder, self).__init__()
          self.latent_dim = latent_dim
          self.encoder = tf.keras.Sequential([
            ##layers.Input(shape=(24, 1)),
            layers.Dense(12, activation='relu'),
            layers.Dense(latent_dim, activation='relu')])
          self.decoder = tf.keras.Sequential([
            layers.Dense(12, activation='relu'),
            layers.Dense(24, activation='linear')])

        def call(self, x):
          encoded = self.encoder(x)
          decoded = self.decoder(encoded)
          return decoded

      autoencoder = Autoencoder(latent_dim)
```

Figure 23. The Smart Grid Autoencoder.

The autoencoder has 24 inputs, 12, 6, 12 hidden layers, and 24 outputs. This means the autoencoder is a stacked symmetric autoencoder. It should be noted that ReLU functions were used for the hidden layers, and linear function was used for the output layer.

```
adamOptimizer = optimizers.Adam(learning_rate=0.00001)
autoencoder.compile(optimizer=adamOptimizer, loss=losses.MeanSquaredError())
##autoencoder.compile(optimizer='adam', loss=losses.MeanSquaredError())
autoencoder.fit(gridload_train_mat, gridload_train_mat,
                epochs=50, batch_size=500,
                shuffle=True,
                validation_split=0.1)
```

Figure 24. Fitting the Autoencoder.

For the fitting of the auto encoder, 50 epochs with a batch size of 500 was used. This was used as the entire data set has only 8760 different rows, 500 batch size is more than enough. The optimizer was Adam, and the loss was used as MeanSquaredError.

C. Testing the Autoencoder

```
[14]: test_input=np.array([[2324, 4977, 3318, 1660, 332, 1660, 331, 3984, 999, 332,
                            1660, 665, 2329, 3322, 331, 994, 1662, 3320, 2325, 3321, 1660, 1661, 996, 995]])
      test_input_norm=np.empty([1,24])
      ##test_input.shape
      for j in range(24):
          for i in range(1):
              test_input_norm[i][j]=test_input[i][j]-mean[j]
              test_input_norm[i][j]=test_input[i][j]/std[j]
      encoded_loadmeasurement = autoencoder.encoder(test_input_norm).numpy()
      decoded_loadmeasurement = autoencoder.decoder(encoded_loadmeasurement).numpy()

[15]: print("Before inverting the normalization:\n", decoded_loadmeasurement)
      for j in range(24):
          for i in range(1):
              decoded_loadmeasurement[i][j]=decoded_loadmeasurement[i][j]*std[j]
              decoded_loadmeasurement[i][j]=decoded_loadmeasurement[i][j]+mean[j]
      print("After inverting the normalization:\n")
      for j in range(24):
          for i in range(1):
              print("{:.1f}".format(decoded_loadmeasurement[i][j]), end=" ")
```

Figure 25. Testing the auto encoder with reasonable values.

After the model was fitted, a test_input array was created where meaningful values were injected to see if the autoencoder will detect that these values are fine. After following the different steps involved, the test values were compared against the predicted model.

```
[16]: MSE=0.0
      Energy=0.0
      for j in range(24):
          for i in range(1):
              Energy=Energy+test_input[i][j]*test_input[i][j]
              MSE=MSE+(test_input[i][j]-decoded_loadmeasurement[i][j])*(test_input[i][j]-decoded_loadmeasurement[i][j])
      MSE=MSE/Energy
      print("\n Reasonable (Correct) MSE=",MSE)

       Reasonable (Correct) MSE= 0.023748437385020936
```

Figure 26. MSE of reasonable values.

It was found that the MSE of the reasonable values to be 0.024, meaning the module is 97.6% accurate at detecting whether the values returned by the grid are reasonable or not.

```
[17]: test_input=np.array([[8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000,
                            8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000, 8000]])
      ##test_input.shape
      for j in range(24):
          for i in range(1):
              test_input_norm[i][j]=test_input[i][j]-mean[j]
              test_input_norm[i][j]=test_input[i][j]/std[j]
      encoded_loadmeasurement = autoencoder.encoder(test_input_norm).numpy()
      decoded_loadmeasurement = autoencoder.decoder(encoded_loadmeasurement).numpy()
      print("Before inverting the normalization:\n", decoded_loadmeasurement)
      for j in range(24):
          for i in range(1):
              decoded_loadmeasurement[i][j]=decoded_loadmeasurement[i][j]*std[j]
              decoded_loadmeasurement[i][j]=decoded_loadmeasurement[i][j]+mean[j]
      print("After inverting the normalization:\n")
      for j in range(24):
          for i in range(1):
              print("{:.1f}".format(decoded_loadmeasurement[i][j]), end=" ")
```

Figure 27. Testing the auto encoder with unreasonable values.

After seeing that our model can detect good values, we tested the model against values that are beyond the maximum values. This was imposed such that the model would be tested in its error detection capabilities.

```
MSE=0.0
Energy=0.0
for j in range(24):
    for i in range(1):
        Energy=Energy+test_input[i][j]*test_input[i][j]
        MSE=MSE+(test_input[i][j]-decoded_loadmeasurement[i][j])*(test_input[i][j]-decoded_loadmeasurement[i][j])
MSE=MSE/Energy
print("\n Wrong (Faulty) Input MSE=",MSE)

 Wrong (Faulty) Input MSE= 1.1850594204852023
```

Figure 28. the MSE of unreasonable values.

The autoencoder was able to detect, with 120% accuracy that the inputted test numbers were erroneous, meaning that our autoencoder has an error detection capability that is 120% accurate. This is because our inputted signal was unreasonable, so the behavior is expected.

CONCLUSIONS AND FUTURE WORK

To sum, machine learning is booming in both researching and the industry, and the implementation of machine learning in smart grids is inevitable. There are a few points that must be noted for any researcher looking to implement autoencoders in grids.

The most difficult thing in this project was finding a proper data set. Machine learning is extremely reliant on data and finding a proper data set was for our purpose much more difficult than what followed. Hence, it is advised that for whatever implementation that this program, real data must be captured from the grid of which it will be used on. Should that data be available, the analysis and implementation of the autoencoder will be much clearer. This is one of the reasons two data sets were used in this project, as one on its own was not going to suffice our usage.

Another aspect is determining the threshold of which the alarm for fault detection to trigger. This threshold will vary greatly based on the initialization of the autoencoder, as well as the quality of the data inputted into the autoencoder.

In the first code created, the MDN worked very well in estimating the power probability distribution. This is very interesting as the first data set was extremely noisy and being able to predict such a noisy data is a benefit for all future work in this field.

As for the second code, it was found that an autoencoder with a relatively reasonable size can predict with great accuracy the error in signals, even though our dimension was 24, it was very fast in fitting as well as accurate in error detection.

The following is a suggestion of research axes to be investigated. Among them, we would like to consider the following:

a. Training for smart grids requires a large database with thousands of physical measurements on dozens of grid parameters. Surprisingly, a neural network playing chess against itself (See AlphaZero at Google) is capable to beat any Chess Grand Master. We propose to investigate the extension of a measurement database via a variational autoencoder.
b. In almost all implementations for smart grids, engineers are using symmetric auto-encoders. However, it would be interesting to investigate asymmetric structures where the number of layers and their size is tuned to the input dimension and distribution.

REFERENCES

[1] Amit Joshi, "Effect of Transformation in Compressed Sensing of Smart Grid Data", ***Indian Institute of Technology Gandhinagar*****, 2019, pp.177-182.**

[2] Mohammed Babakmehr, "Compressive Sensing-Based Topology Identification for Smart Grids", ***IEEE TRANSACTIONS ON INDUSTRIAL INFORMATICS*****, 2016, pp.532-543.**

[3] "Difference between Traditional Power Grid and Smart Grid", ***Electrical Academia, n.d.***

Accessible at: https://electricalacademia.com/electric-power/difference-traditional-power-grid-smart-grid/#:~:text=The%20traditional%20power%20grid%20is,and%20different%20types%20of%20loads.

[4] "Why Smart Grids?", ***EDSO*****, 2021.**

Accessible at: https://www.edsoforsmartgrids.eu/home/why-smart-grids/

[5] Arden Dertat, "Applied Deep Learning – Part 3 Autoencoders", ***Towards Data Science,*** **Oct. 2017.**

Accessible at: https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798

[6] Venkata Krishna Jonnalagadda, "Sparse, Stacked, and Variational Autoencoder", ***Medium,*** **Dec. 2018.**

Accessible at: https://medium.com/@venkatakrishna.jonnalagadda/sparse-stacked-and-variational-autoencoder-efe5bfe73b64

[7] A. Majumdar and A. Tripathi, "Asymmetric stacked autoencoder," ***2017 International Joint Conference on Neural Networks (IJCNN)*****, Anchorage, AK, 2017, pp. 911-918, doi: 10.1109/IJCNN.2017.7965949**

[8] Tiago Sousa, Tiago Soares, Pierre Pinson, Fabio Moret, Thomas Baroche, & Etienne Sorin. (2018). The P2P-IEEE 14 bus system data set [Data set]. Zenodo.

Accessible at: http://doi.org/10.5281/zenodo.1220935

[9] xb00dx, Fifty-Six Node Data Set.

Accessible at: https://github.com/xb00dx

[10] Google, Normalization.

Accessible at: https://developers.google.com/machine-learning/data-prep/transform/normalization

[11] Shaked Zychlinski, Predicting Probability Distributions Using Neural Networks. Nov.2018

Accessible at: https://engineering.taboola.com/predicting-probability-distributions/

[12] NASA, Generalized Extreme Value distribution and calculation of Return value.

Accessible at: https://gmao.gsfc.nasa.gov/research/subseasonal/atlas/GEV-RV-html/GEV-RV-description.html