Indian Institute of Space Science and Technology

Astrophysical spectrum parameter estimation using Deep learning

Course Project Deep-learning, 2020

Project by -
Shivam Kumaran
SC17B122
IIST

Kshitij Sunil
SC17B026
IIST

Guide
Dr. Deepak Mishra
IIST

January 13, 2021

Acknowledgement

We are thankful to Dr Deepak Mishra for teaching us the concepts of Deep Learning, and for his guidance and motivation to implement the knowledge earned.

We are also thankful to Dr Samir Mandal for making us understand the concepts of Data-fitting for high energy Astrophysical sources and the physical aspects behind such processes and also for his inspiring words and all time motivation.

We are grateful to all the Machine Learning researchers to develop such beautiful concepts and the team at KERAS and Tensorflow to make it several times easier to implement such complex algorithms.

This work uses the software package SOXS , Copyright 2018, Lynx Science Support Office for data generation.

Abstract

In this project work, we have worked on the problem of parameter estimation from received spectrum from astronomical source. We have developed a FC model using keras trained on data generated by *SOXS* python package to estimate parameters corresponding to `poowerlaw` model. We have developed a reconstruction network , which is our designed Denoising Autoencoder , for reconstruction of spectrum from a given spectrum having values missing in certain energy band, and then estimating parameter for powerlaw from this reconstructed spectrum. We have also tried to improve the reconstruction performance of denoising auto-encoder by an algorithm , which is reconstruction in 2 phases , which significantly improves the parameter prediction performance

Contents

Chapter 1

Introduction

Astronomical data received from a satellite needs to be processed further in order to draw meaningful conclusions. One of the primary steps in processing the data obtained from science observation satellites exploring deep space is the energy distribution of a radiating source received from a distant star or any other astronomical body over a range of different energy channels. There exist mathematical models for various radiation emitting bodies in deep space. The task is to fit the data received from a known source to one of these existing models and accurately identify the parameters pertaining to this fit for further scientific investigations.

The present work attempts to perform the task of fitting for the spectra extracted from the data intercepted by a satellite from a distant source to its mathematical model using Regression models built using Neural Networks. A complication that arises in this task is the noise that is present in the signal and the missing energy components in the signal. A typical energy flux spectrum picked up by X-ray instrument on observation satellites **MAXI/GSC** and **SWIFT/XRT** is shown in Fig. **??**. It is observed that data corresponding to certain bands of energies are missing. The noise in the signal must be filtered and also the missing data in specific energy channels must be predicted using scientific methods before a mathematical model can be fitted, and also we need to interpolate the missing gap in a very effective way, incorporating the prior knowledge of the model. This task is accomplished in the present work using Denoising Autoencoder network.

Autoencoders are versatile neural networks which are usually used in anomaly detection and data reconstruction purposes. A basic autoencoder comprises of an encoder and a decoder network stacked together. The 'bottled neck' shape of the encoder network is what gives it the its property of reconstruction ([1]). An autoencoder may have a simple feedforward neural network or complex neural networks like CNN depeding on the application. [2] reports the various applications of an Autoencoder. A typical autoencoder is shown in Fig. 1.2.

In this work, a novel strategy is used to implement a regression model along with a feedforward autoencoder to achieve the aforementioned objective.

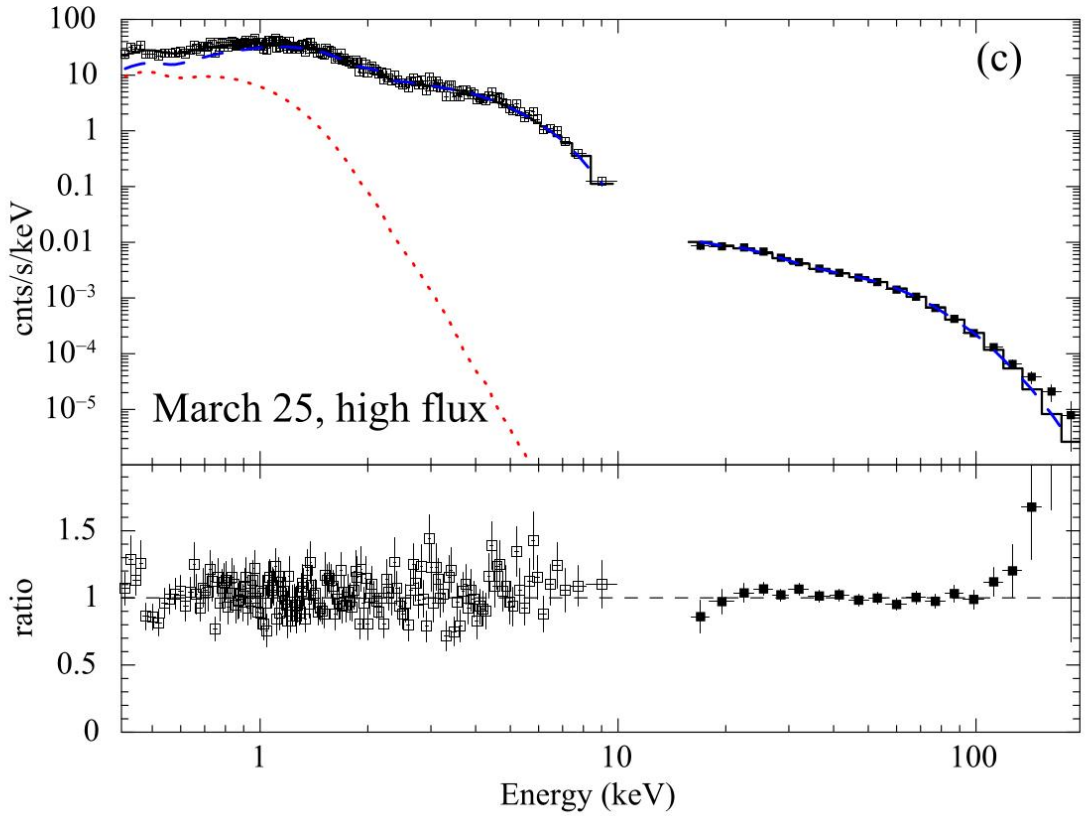

Figure 1.1: Spectrum extracted from Observed data from an **X-ray** binary **MAXI J1820+070** obtained from two satellite instruments **MAXI/GSC** and **SWIFT/XRT** , In the top panel spectra are shown for two different bands corresponding to **SWIFT/XRT** and **MAXI/GSC** respectively . In the bottom panel, shows the difference of the best fit model to the given spectrum.

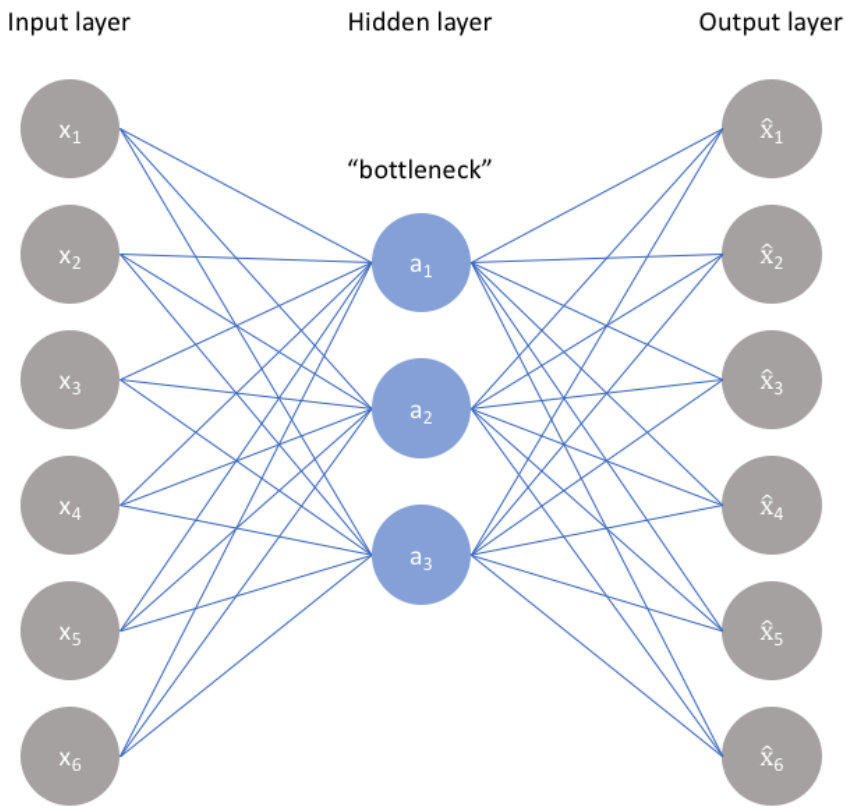

Figure 1.2: Simplest Autoencoder network

Chapter 2

Data: source and preparation

We have to do the entire analysis for tha data resembling closely to the real astronomical sources.

SOXS is a python software package to simulate *x-ray* spectrum from astronomical sources. This software is capable of generation x-ray data for the following source emission models.

- Constant Spectra
- Power-law spectrum
- Thermal Spectra
- any **XSPEC** Model

Note : XSPEC is another x-ray data analysis package, and contains quiet a large number of other models, hence from SOXS we can generate several other spectrum also using XSPEC For our task we have tried to generate data for powerlaw Model :

$$A(E) = K[E(1+z)]^{-\alpha}$$

We are working under assumption that from the observed readings from the instrument, after using suitable response files for the corresponding instrument, we have finally extracted Energy-vs-flux values. However **SOXS** also provide the commands to get evelnt list corresponding to certain model simulated for certain instruments such as **chandra , HITOMI** , etc.

- Model used : Powerlaw
- Parameter values :
  - Z : [0.5 , 2.5] , Uniform distribution
  - α : [0.5 , 2.5] , Uniform distribution
  - norm : 10^{-7}
- Energy range (0.3-7.0)keV
- Energy bins : 128
- Total instances generated 10,000
- **Dataset:** 10,000 instances , each 128 length vector havin flux values stored.

2.1 Data generation code

```
from soxs import Spectrum
import numpy as np

alpha = np.random.uniform(low = 0.5 , high= 2.5 , size = 10000)
z_val = np.random.uniform(low = 0.1 , high= 2.5 , size = 10000)
norm  = 1e-7
emin = 0.5
emax  = 5.0
nbins = 128
data = []
param = []
i = 0
for a, z in zip(alpha, z_val):
    #print(a)
    spec = Spectrum.from_powerlaw(a, z , norm, emin, emax,
        nbins)
    _ , flx = spec.ret_spectrum_krn()
    data.append(flx)
    param.append([a,z])
    if(i%25==0):
        print('Generating data for iteration:' , i)
    i+=1

data = np.asarray(data)
param = np.asarray(param)
np.savetxt('data/data_spectrum.csv' , data)
np.savetxt('data/params_spectrum.csv' , param)
data_log = np.log(data)
data_log_norm = (data_log - np.mean(data_log))/np.var(data_log)
np.savetxt('data/data_log_norm_spectrum.csv' , data_log_norm)
```

Listing 2.1: Data generation code from SPXS spectrum generation package

2.2 Data Output

- **Dataset:** 10,000 instances , each 128 length vector having flux values stored.
- **Target :** 10,000 instances , each vector of length -2 (parameters z and α)
- Stored in `data.csv` , `params.csv`

Chapter 3

Important Utility routines

3.1 Train test split routine

```
def train_test_split(data, params , split):
    split = 1-split
    n_split = int(len(data)*split)
    d_list = [[d,p] for d,p in zip(data, params)]
    d_list = np.asarray(d_list)
    np.random.shuffle(d_list)
    d_list_train = d_list[:n_split,:]
    d_list_test = d_list[n_split:,:]

    train_data = d_list_train[:,0]
    train_data = np.asarray([t for t in train_data])
    test_data = d_list_test[:,0]
    test_data = np.asarray([t for t in test_data])

    train_p = d_list_train[:,1]
    train_p = np.asarray([t for t in train_p])

    test_p = d_list_test[:,1]
    test_p = np.asarray([t for t in test_p])

    return (train_data , train_p) , (test_data , test_p)
```

3.2 Model Performence evaluation routine

```
def model_score(model , data , params , plot=False , epochs = 20 , verbose
   = 0):

    def plot_loss(history):
        plt.plot(history.history['loss'], label='train loss')
        plt.plot(history.history['val_loss'], label='val_loss')
        #plt.ylim([0, 10])
        plt.xlabel('Epoch')
        plt.ylabel('Error [MPG]')
```

```
        plt.legend()
        plt.grid(True)
        plt.show()

    (train_data, train_p) , ( test_data , test_p) = train_test_split(data
        , params , 0.2)

    history = model.fit(train_data, train_p,validation_split=0.2,
        epochs=epochs , verbose=verbose)

    if(plot):
        plot_loss(history)

    y_true_test = np.copy(test_p)
    y_pred_test = model.predict(test_data)

    y_true_train = np.copy(train_p)
    y_pred_train = model.predict(train_data)
    mse_train = mean_squared_error(y_true_train, y_pred_train)
    mse_test = mean_squared_error(y_true_test, y_pred_test)
    model.summary()
    return mse_train , mse_test
```

3.3 Data generatio routine

```
from soxs import Spectrum
import numpy as np

alpha = np.random.uniform(low = 0.5 , high= 2.5 , size = 10000)
z_val = np.random.uniform(low = 0.5 , high= 2.5 , size = 10000)
norm  = 1e-7
emin = 0.3
emax  = 7.0
nbins = 128
data = []
param = []
i = 0

for a, z in zip(alpha, z_val):
    #print(a)
    spec = Spectrum.from_powerlaw(a, z , norm, emin, emax, nbins)
    _ , flx = spec.ret_spectrum_krn()
    data.append(flx)
    param.append([a,z])
    if(i%25==0):
        print('Generating data for iteration:' , i)
    i+=1

data = np.asarray(data)
param = np.asarray(param)
np.savetxt('data/data_spectrum.csv' , data)
np.savetxt('data/params_spectrum.csv' , param)
```

3.4 Model performence plot routine

```
def plot_loss(history):
  fig = plt.figure(figsize=(8,6))
  plt.plot(history.history['loss'], label='train loss')
  plt.plot(history.history['val_loss'], label='val_loss')
  #plt.ylim([0, 10])
  plt.xlabel('Epoch')
  plt.ylabel('Error [MPG]')
  plt.legend()
  plt.grid(True)
  return fig
```

Chapter 4

Regression Network for Predicting Model Parameters

The signals intercepted by a satellite pertain to the energy fluxes in different energy channels. It is usually required to fit such a data to any one of the following existing models based on the source of the signal and find the parameters corresponding to the model.

Thus a regression model is required for the existing models so that the relevant parameters of the parent model can be obtained for further study. Mathematical curve fitting (like least square fit for linear relationships) can be used for simple cases however, neural networks provide a fast and convenient way to fit the data to a particular model and obtain the relevant parameters even for complex model. Data can be fed into a regression network obtained by training a neural network for a particular existing model and the network outputs the required parameters. This approach is attempted in the present study.

A neural network is trained for the power law model using artificially generated data. This neural network is used in subsequent parts of the study to obtain the parameters for a specific input data. These parameters of fit form the final output for an intercepted data.

: Power Law Model

For an input x (usually corresponding to the energy channels), the output y is given by:

$$y = k(x(1+z))^{-\alpha}$$

The parameters of the model are z and α whereas k is a scaling factor.

4.1 Training data generation

To generate the training data, various values of parameters are chosen and outputs are obtained for a specific range of inputs. 10,000 values of α and z are obtained from a uniform distribution in the range of 0.5 to 2.5. Input x correspond to energy channels

and is varied from 0.1 to 10 and 128 discrete points are obtained.

4.1.1 Data Pre-processing and Normalization

To generalize the present model, the input data is log-normalized in the following way:

$$x' = \log(x)$$

$$x_{inp} = \frac{x' - \bar{x'}}{\mathrm{var}(x')}$$

The input to the neural network is thus x_{inp} while the outputs are the parameters of the power law model.

A typical plot of the training instances is shown in Fig. 4.1. Each curve in the plot is a training instance for the network and there are 10,000 such instances. The data overview for the network is presented in Table 4.1.

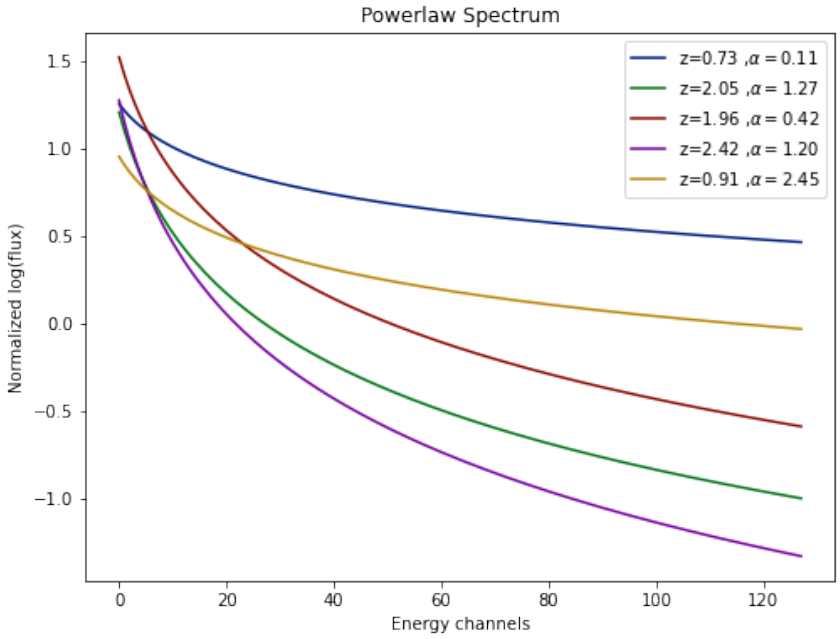

Figure 4.1: Sample Training Instances

Table 4.1: Data Overview

Input data vector length	128
Target data vector length	2
Number of instances	10000

4.2 Regression Network Training

Since the purpose of this neural network is a regression fit, fully connected layers with relu-activated neurons are stacked together to form a basic neural network and has been

trained using the data above. Hyperparameter tuning is done using the metrics from the testing dataset and is described in the following subsection. The definition of model is put into a function that designs and returns a Network model coresponding to given input shape (number of layers and the number of nodes in ecah layer)

: Regression Neural Network

- Network structure : kept as argument to a function for an efficient hyperparameter tuning
- Optimizer ; *ADAM*
- Loss : MSE and MAE tested

Listing 4.1: Function to generate FC model for parameter regression

```
def pred_model(l_struct):
  '''
  Prediction Model
  Returns A fully-connencted Model for
  Network structure given by number of
  layers in the input argument.
  Argument l_struct basically encodes the
  information about layer structure.
  '''
  inp = keras.Input(shape=(128 , ))
  x = layers.Dense(l_struct[0] , activation='relu')(inp)
  l_struct = l_struct[1:]
  for l in l_struct:
    x = layers.Dense(l, activation='relu')(x)
  op = layers.Dense(2, activation='relu')(x)
  model = keras.Model(inputs=inp , outputs=op , name = 'regressor')
  return model
```

4.2.1 Hyperparameter Tuning

A basic regression model often performs best with ReLu hidden layers long with an ADAM optimizer. Hence, this configuration is taken with a learning rate of 0.001. This was found suitable for the current data.

The data is split into testing and training data in the ratio 20-80. The training data is further subdivided into validation and training data in the ratio 20-80 using internal functions in Keras.

The hyperparameter tuning for getting the best network structure is encpsulated into a single fiunction, that takes care of desingning a ccertain structured fully connected model, to fit the data to training model, and to get the validation and test loss.

Hyperparameter tuning code

```
def model_score(model , data , params , plot=False , epochs = 20 , verbose
    = 0):
```

```
'''
Take in certain model , complete dataset and returns
the training and testing performence.
The user can select whether to plot the performence metrices,
and the number of epochs.
'''
    def plot_loss(history):
        plt.plot(history.history['loss'], label='train loss')
        plt.plot(history.history['val_loss'], label='val_loss')
        #plt.ylim([0, 10])
        plt.xlabel('Epoch')
        plt.ylabel('Error [MPG]')
        plt.legend()
        plt.grid(True)
        plt.show()

    (train_data, train_p) , ( test_data , test_p) = train_test_split(data
        , params , 0.2)

    history = model.fit(train_data, train_p,validation_split=0.2,
        epochs=epochs , verbose=verbose)

    if(plot):
        plot_loss(history)

    y_true_test = np.copy(test_p)
    y_pred_test = model.predict(test_data)

    y_true_train = np.copy(train_p)
    y_pred_train = model.predict(train_data)
    mse_train = mean_squared_error(y_true_train, y_pred_train)
    mse_test = mean_squared_error(y_true_test, y_pred_test)
    model.summary()
    return mse_train , mse_test
```

Using the above hyperparameters, the architecture of the network is varied randomly and results are tabulated in Tables 4.2 and 4.3. Table 4.2 gives the results for a loss function of Mean Squared Error (MSE) and Table 4.3 gives the results for a loss function of Mean Absolute Error (MAE). The metrics shown are evaluated using MSE for both (for a comparison) after the last epoch. Every architecture is run for 20 epochs for tuning.

Table 4.2: Architecture tuning (Loss function: MSE)

Architecture	Training error (MSE)	Testing error (MSE)
128-64-64-32-32-2	3.56e-05	3.48e-05
128-64-64-64-64-2	1.29	1.285
128-32-32-32-32-2	2.581	2.578
128-64-64-2	3.12e-4	3.124e-4
128-32-32-2	1.29	1.294
128-64-32-2	6.34e-4	6.33e-4

Table 4.3: Architecture tuning (Loss function: MAE)

Architecture	Training error (MSE)	Testing error (MSE)
128-64-64-32-32-2	9.55e-4	9.77e-4
128-64-64-64-64-2	1.29	1.28
128-32-32-32-32-2	1.29	1.30
128-64-64-2	9.68e-4	9.58e-4
128-32-32-2	2.58	2.584
128-64-32-2	1.42e-4	1.40e-4

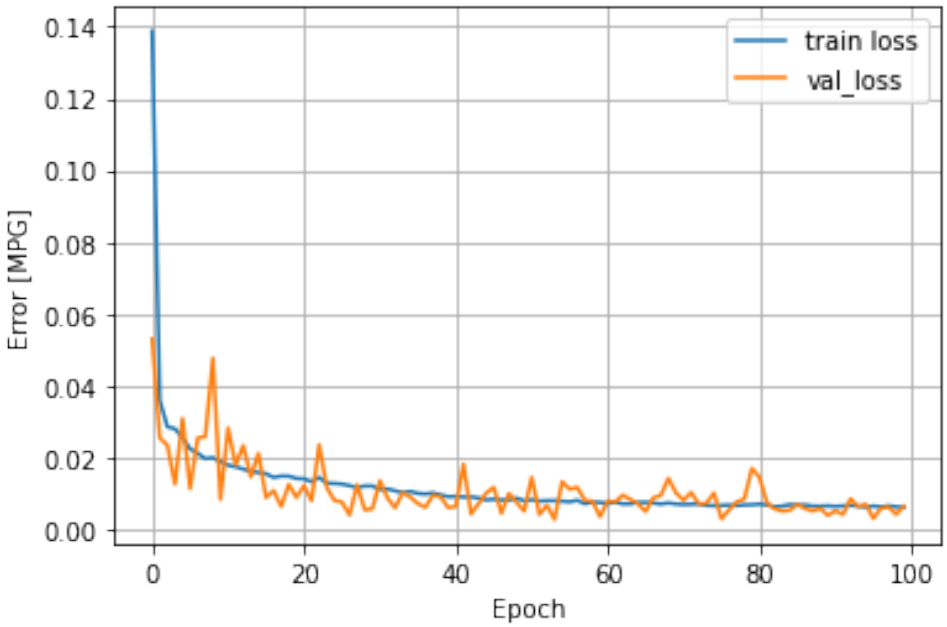

Figure 4.2: Learning curve for the trained model - 128-64-64-32-32-2

4.2.2 Inference and Final Network

The trained network is a 5 layer network with the architecture 128-64-64-32-32-2. Each layer uses a ReLU activated neuron. ADAM optimizer is used along with a mean squared error loss function. The learning curve during training of this model is shown in Fig. 4.2. The results of this network are shown in the scatter plot in Fig. 4.3.

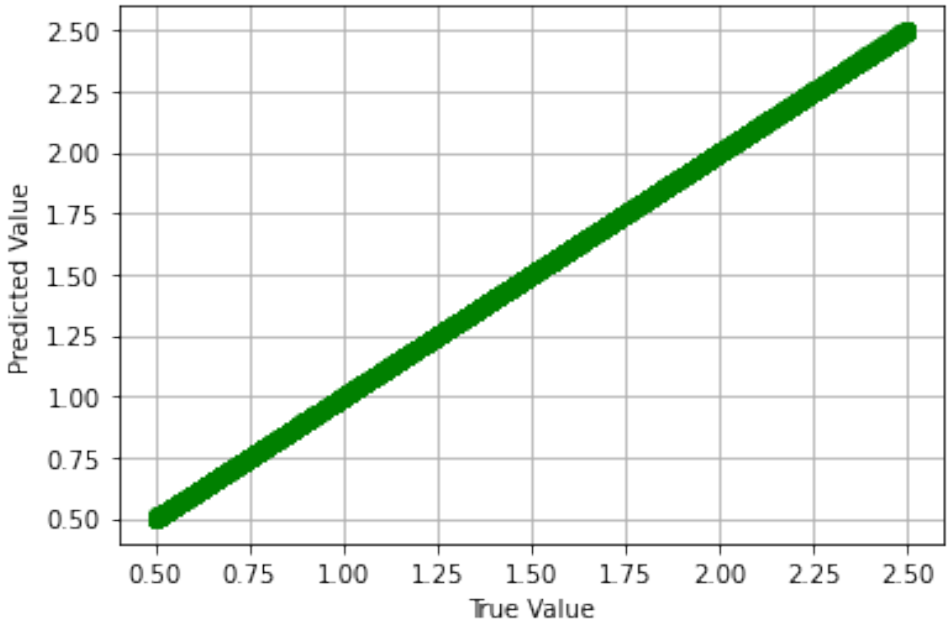

Figure 4.3: Scatter plot for the regression network - 128-64-64-32-32-2

Chapter 5

Data Reconstruction using Denoising Auto Encoder

The data received from Astrophysical sources are usually do not have very high signal to noise ratio, and hence removing noise to get the actual signal becomes a vital task for successful model fitting and estimating parameters to a high accuracy.

Additional challenge is it is very rare that a satellite receive signal from the source for the entire energy band. Hence we need to combine the spectrum received from different observatories (satellites) to get the entire spectrum. In the process we are bound to have spectrum missing from certain energy band.

A Denoising Auto-encoder becomes a suitable choice in such cases, as once the Denoising Auto-encoder trained noisy data to reconstruct missing data, it becomes capable to generate the data in the missing energy band also , once the noisy spectrum is fed into it.

5.1 Strategy

Our objective is to fill in missing data and remove noise.

1. Design a basic Denoising Auto-encoder.
2. Add noise to training data
3. Train DAE on this noisy data
4.
   - input to trained- noisy data
   - Target to network training - Clean data.
5. Introduce gap in the test data
6. fill the gap with random normal noise
7. Find output of the network to this band-missing data

5.1.1 Data Preparation

We need three kind of datasets for entire training testing reconstruction and parameter regression routine.

1. Training Test and validation data
2. Data With artificially introduced band-gaps **Z**ero **F**illed **M**issing **D**ata , **ZFMD**.
3. Band-missing data filled with gaussian noise $\mathcal{N}(0, 0.5)$**N**oise **F**illed **M**issing **D**ata , **NFMD**

Code

Listing 5.1: Training and test data

```
en = np.linspace(0.3, 7.0 , 128)
data = np.loadtxt('data/data_spectrum.csv')
params = np.loadtxt('data/params_spectrum.csv')
print(data.shape)
print(params.shape)
data = log_norm(data)
data_noisy = data + 0.7*np.random.normal(size = data.shape)

'''
Training testing split
(use our custom designed function)
more in Appendix
'''
(x_train,y_train) , (x_test,y_test) = train_test_split(data , params , 0.2)
```

Listing 5.2: Introduce artificial Gap in data to simulate missing data

```
def simulate_missing_data(data):
'''
takes in full-band data,
returns two kind of data:
data with missing gap
in between for two-bands [20-40] and [80-100]
and :
data with Gaussian noise introduced in missing band.
'''
    data_missing = []
    data_noise_filled = []
    for x in data:
        temp = np.copy(x)
        temp[20:40] = np.zeros(20)
        temp[80:100] = np.zeros(20)
        data_missing.append(temp)
        temp_g = np.copy(x)
        temp_g[20:40] = 0.4*np.random.normal(0.0, 0.5 , 20)
        temp_g[80:100] = 0.4*np.random.normal(0.0, 0.5 , 20)
        data_noise_filled.append(temp_g)
    data_missing = np.asarray(data_missing)
    data_noise_filled = np.asarray(data_noise_filled)
    return data_missing , data_noise_filled
```

```
x_train_gap , x_train_filled = simulate_missing_data(x_train)
x_test_gap , x_test_filled = simulate_missing_data(x_test)
```

5.2 Network design

We designed our own Auto-encoder and Denoising Auto encoder using *KERAS Functional API*

: Auto-encoder

- Library Used - KERAS functional API for model design.
- Encoder layer structure
  - Layer structure - [128 , 64 , 64 , 32 ,64 , 64 , 128]
  - Fully connected layers
- Hidden layer activations: **ReLU**
- Output Layer Activation function : **LeakyRelU ,** $\alpha = 0.7$

: Denoising Auto-encoder

- Library Used - KERAS functional API for model design.
- Encoder layer structure
  - Layer structure - [128 , 64 , 64 , 256 ,64 , 64 , 128]
  - Fully connected layers
- Hidden layer activations: **ReLU**
- Output Layer Activation function : **LeakyRelU ,** $\alpha = 0.7$

5.2.1 AE : Code

```
enc_inputs = keras.Input(shape=(128 , ))
x = layers.Dense(64 , activation='relu')(enc_inputs)
x = layers.Dense(64, activation='relu')(x)
enc_outputs = layers.Dense(32, activation='relu')(x)

x = layers.Dense(64 , activation='relu')(enc_outputs)
x =  layers.Dense(64 , activation='relu')(x)
x = layers.Dense(128)(x)
dec_outputs = layers.LeakyReLU(alpha = 0.8)(x)
encoder = keras.Model(inputs=enc_inputs , outputs=enc_outputs , name =
    'encoder')
#encoder.summary()
ae = keras.Model(inputs=enc_inputs , outputs=dec_outputs , name =
    'autoencoder')
ae.summary()
```

5.2.2 DAE : Code

```
enc_inputs = keras.Input(shape=(128 , ))
x = layers.Dense(64 , activation='relu')(enc_inputs)
x = layers.Dense(64, activation='relu')(x)
enc_outputs = layers.Dense(256, activation='relu')(x)

x = layers.Dense(64 , activation='relu')(enc_outputs)
x =  layers.Dense(64 , activation='relu')(x)
x = layers.Dense(128)(x)
dec_outputs = layers.LeakyReLU(alpha = 0.8)(x)
encoder = keras.Model(inputs=enc_inputs , outputs=enc_outputs , name =
    'encoder')
#encoder.summary()
den_ae = keras.Model(inputs=enc_inputs , outputs=dec_outputs , name =
    'autoencoder')
den_ae.summary()
```

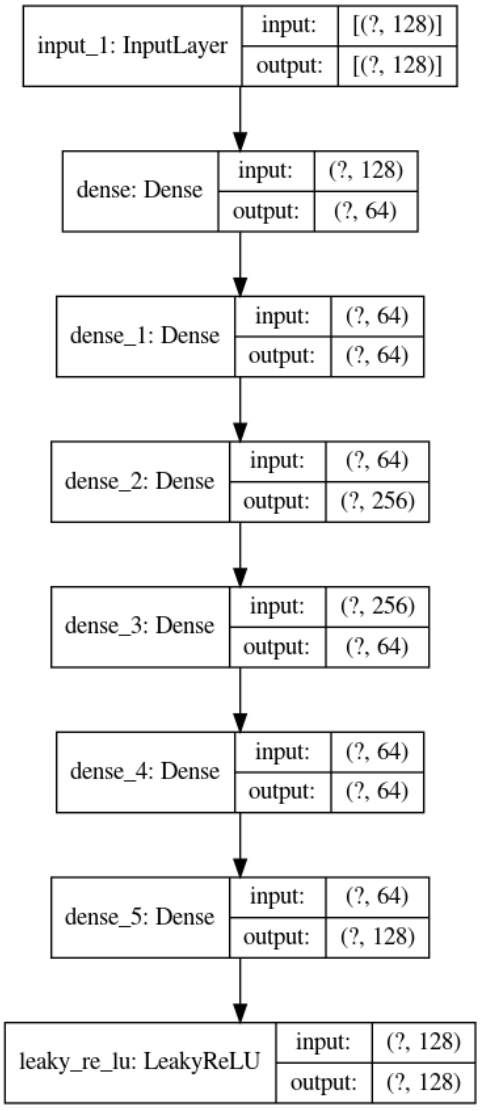

Figure 5.1: Denoising Auto-encoder Network structure

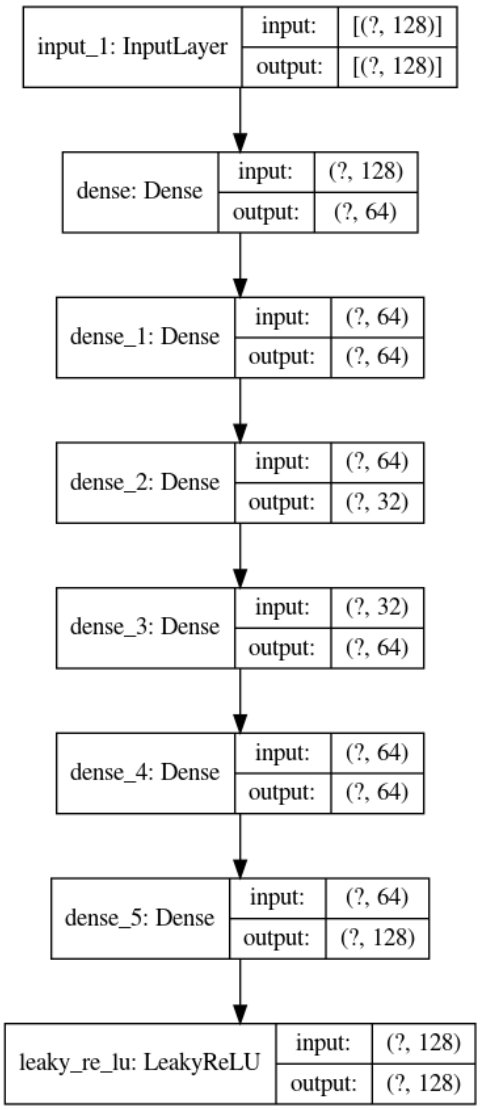

Figure 5.2: Auto-encoder Network structure

5.3 Reconstruction Algorithm

Once the Auto encoder is trained , it will generate reconstructed data, which will be most likely similar to the original data spectrum.

We try reconstruction for both *Denoising Auto encoder* and **Denoising Auto encoder** by giving as input:

- Original data
- Data With missing band, filled with constant as '0' value - we call it **Zero-filled missing data , ZFMD**
- Fill the missing band with gaussian noise with 0 mean : - we call it **Noise filled missing data , NFMD**

5.4 Losses and Result

To evluate our model's performance we have defined two kind of losses.

- Reconstruction Loss :

  Estimated how close is our reconstructed data to the original data. Defined as *Mean Squared Error* between true and reconstructed Output

- Prediction Loss :

  When we use the dataset to predict parameter using fully our regression Neural Network , defines how close is our estimated parameter to the true parameter.Defined as *Mean Squared Error* between true and estimated parameter.

5.4.1 Reconstruction Performance

Entire Model evaluation was encapsulated into a single function:

Listing 5.3: Reconstruction and Prediction model evaluation code

```
def recon_score(recon_model , bad_data , good_data):
    '''
    returns reconstruction error for a given model
    '''
    recon_data = recon_model.predict(bad_data)
    recon_error = mean_squared_error(good_data , recon_data)
    return recon_error
def pred_error(reg_model , data , params):
    '''
    returns parameter prediction error
    for the given data set
    '''
    y_pred = reg_model.predict(data)
    mse = mean_squared_error(params , y_pred)
    return mse
```

Performance On No-Noise , no missing data

For both the encoders , in this input data is full-band data.

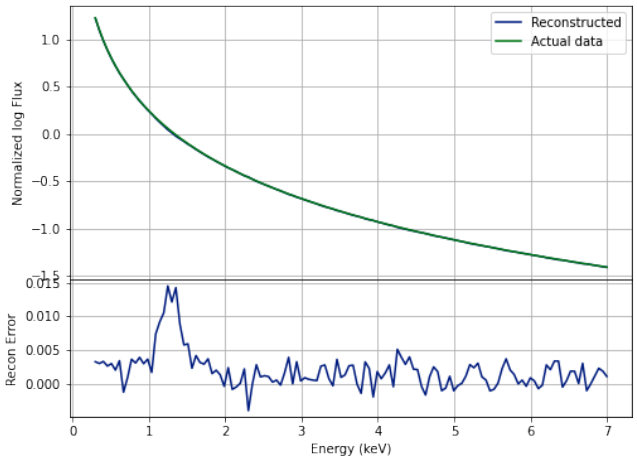

Figure 5.3: Reconstruction from Auto-encoder for the first sample in test dataset.

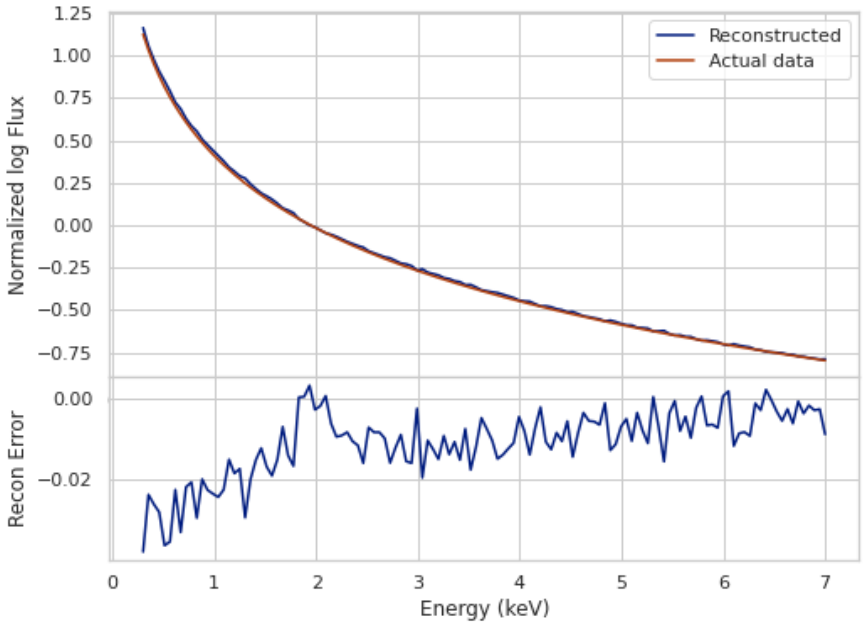

Figure 5.4: Reconstruction from Denoising Auto-encoder the first sample in test dataset.

	Training dataset	Test Dataset
Auto-encoder	1.2022e-05	1.1786e-05
Denoising Autoencoder	0.000637	0.000650

Table 5.1: Reconstruction Losses on No-missing data

Performance on Missing-data

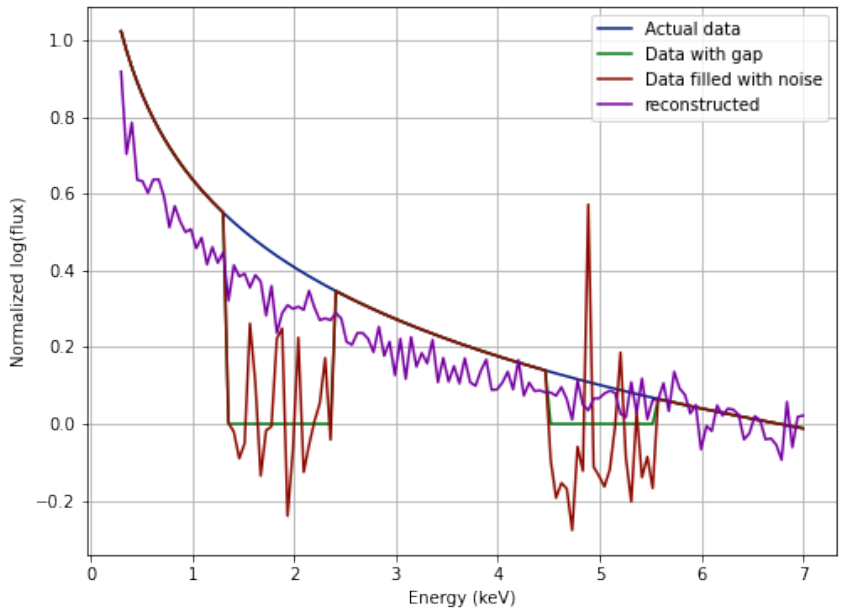

Figure 5.5: Reconstruction from Auto-encoder for one of the missing-data samples

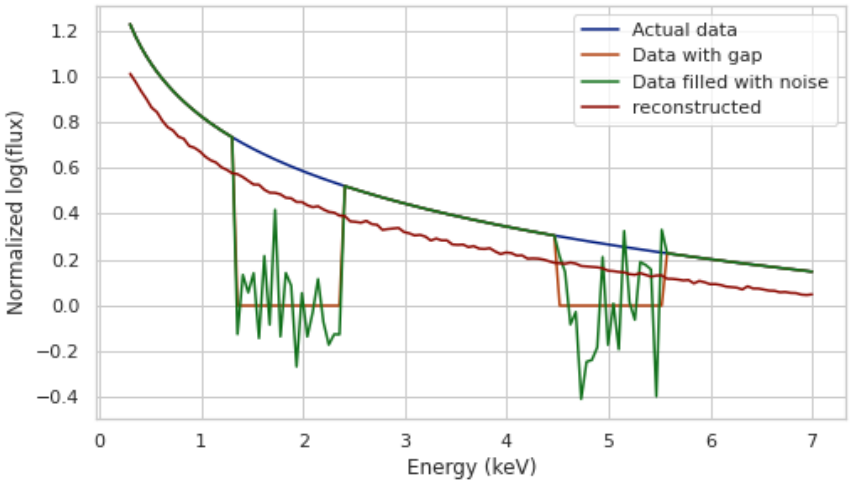

Figure 5.6: Reconstruction from Denoising Auto-encoder for one of the missing-data samples

We see similar almost similar reconstruction performance for AE and DAE, since our final goal is parameter prediction , the most important loss we need to worry about is *Prediction loss*. Prediction loss is computed only for the reconstructed data for both the encoder models, as the dataset (missing dataset) fed to the prediction model is same , (it has yet not passed through the encoders...and hence same prediction would be given).

: prediction loss on reconstructed data

- Auto-encoder : 3.18
- Denoising Auto-Encoder : 0.39

5.5 Inference

1. Auto-encoder is able to reconstruct data mush better than Denoising auto encoder for Data with no-missing values
2. Reconstruction performance for both AE and DAE is almost similar for data with missing-values and noise filled missing data **NFMD**
3. Even after having comparable reconstruction performance, reconstruction from DAE performed almost 10 times better then that of reconstructed by AE , for given NFMD.
4. Hence we will proceed with DAE , and try to make it better in the next section.

Chapter 6

Improving Encoder : Smart Reconstruction

6.1 Phased reconstruction

Earlier we filled the missing band with Noise havin 'zero' mean, but for a satisfactory reconstruction would be given if this noise filled data would be closer to the true value. But before any reconstruction we have no clue of what this true men would be. So we directly try to reconstruc for once. But after one phase of reconstruction, we have a better picture of what is likely a closer value in this missing band.

Procedure: Phased reconstruction

1. Fill the Bnad-missing data with noise of zero $\mathcal{N}(0, 0.5)$ call it **NFMD**
2. Pass it through *DAE* . and get the reconstructed output. , call it **RE**
3. In the missing -band region of the **NFMD**, add the reconstructed output. this makes the mean of the noise to be closer to the true likely data
4. Now do prediction on this added data.
5. Find Reconstruction and prediction losses
6. Repeat the procedure from step 3 , till we get the prediction error reducing.
7. Find an ideal value for number of iterations.

6.2 Code for Phased reconstruction

For model prediction performence and the phased reconstruction for a given number of loops, a code is written , which takes in the $\mathcal{N}(0, 0.5)$ and output the prediction losses for **n** number of iterations:

Listing 6.1: Reconstruction and parameter estimation performance evaluation code

```
def filling_algo(n , data_filled):
```

```
    x_train_filled = np.copy(data_filled)
    x_train_recon = den_ae.predict(x_train_filled)
    pred_loss = pred_error(reg_model , x_train_recon , y_train)
    print('_____________________________________________________')
    print("Parameter regression error:{:.4f}".format(pred_loss) )
    data_to_plot = (x_train[0] ,  x_train_gap[0] , x_train_filled[0] ,
        x_train_recon[0])
    legend = ['Actual data' , 'Data with gap' , 'Data filled with noise' ,
        'reconstructed']
    fig = plot_all(data_to_plot, legend)
    fig.savefig('plots/reconstruction_plot_loop_0.png')
    plt.show()
    for i in range(1,n):
        for x , x_r in zip(x_train_filled , x_train_recon):
            x[20:40] = x[20:40]+x_r[20:40]
            x[20:40] = x[20:40]+x_r[20:40]
        x_train_recon = den_ae.predict(x_train_filled)
        pred_loss = pred_error(reg_model , x_train_recon , y_train)
        print('_____________________________________________________')
        print("Parameter regression error:{:.4f}".format(pred_loss) )
        data_to_plot = (x_train[0] ,  x_train_gap[0] , x_train_filled[0] ,
            x_train_recon[0])
        legend = ['Actual data' , 'Data with gap' , 'Data filled with
            noise' , 'reconstructed']
        fig = plot_all(data_to_plot, legend)
        fig.savefig('plots/reconst_plot_loop_'+str(i)+'.png')
        plt.show()

filling_algo(3 , x_train_filled)
```

6.3 Result

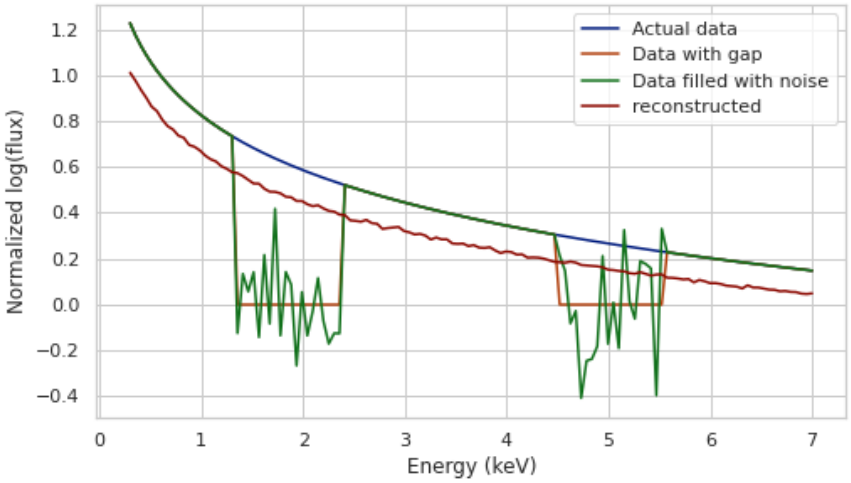

Figure 6.1: Reconstruction From DAE for Iteration:0

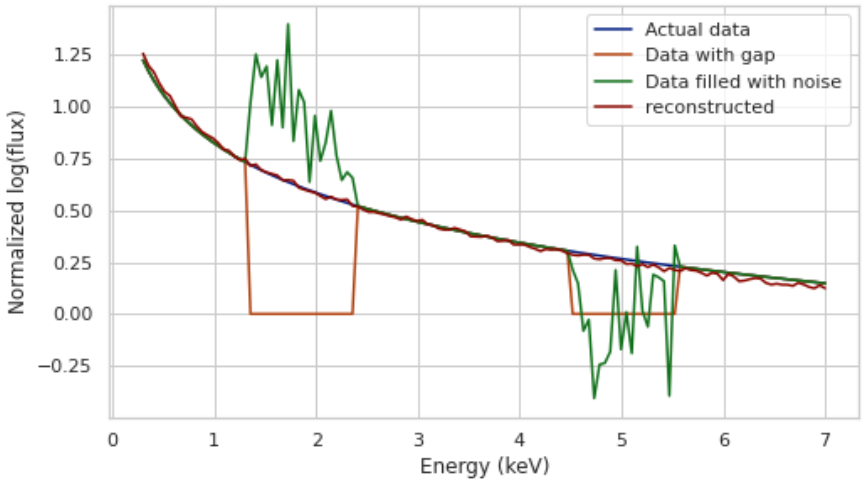

Figure 6.2: Reconstruction From DAE for Iteration:1

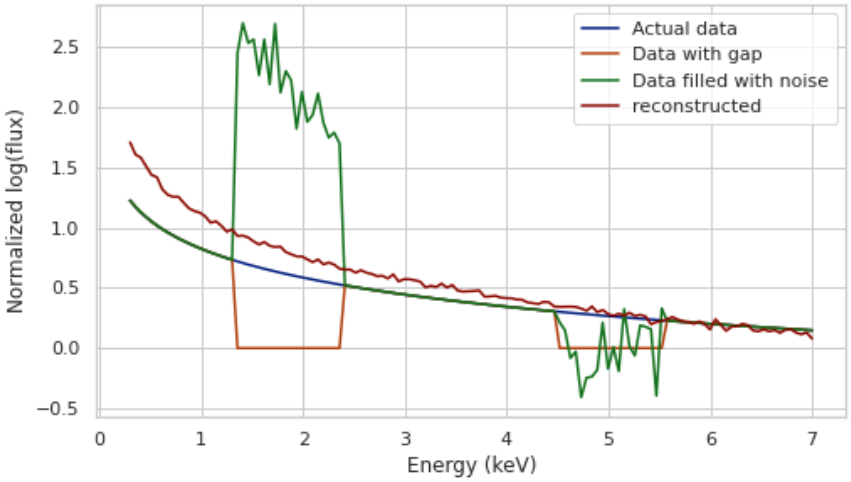

Figure 6.3: Reconstruction From DAE for Iteration:2

: Prediction Loss for $i^{th} iteration$

- Iteration 1 : 0.3924
- Iteration 2 : 0.1474
- Iteration 3 : 0.6819

6.4 Inference

- We see that after 1st iteration, adding the reconstructed output to the noise-filled missing data (NFMD) in the band-missing region, makes the mean of noise closer to the true value.

- Prediction on this improved input data gives better parameter prediction.
- But after 2nd iteration , the reconstructed data overshoots the actual data and hece prediction error increases
- This algorithm gives idea result ffor 1st iteration, that is only one phase of *improved iteration*

Chapter 7

Conclusion and Future

7.1 Conclusion

In this Project we have successfully designed a regression neural network Model, which when provided with a 128 length vector, (flux spectrum corresponding to energy channels) , can predict the parameter corresponding to powerlaw model.

We have also simulated data available only in certain energy bands , (spectrum with band gap) and found out that with Denoising Auto encoder , using that band-missing spectrum , we can satisfactorily reconstruct the data in the entire energy-band considered.

Further we also tried to develop an algorithm 'reconstruction in phases' , that updates the missing part of the data using reconstructed data and then do another phase of reconstruction , giving a much better enhancement in the performance as far as parameter estimation is concerned.

7.2 Future work

- To design and train on complex emission model
- To further improve the performance of regression model
- To explore other generative models for reconstruction of the missing spectrum
- To develop the entire reconstruction and parameter estimation as a single package.
- Creating an open source project for further involvement of people in this project and better implementation.

Bibliography

[1] Will Badr. "Auto-Encoder: What Is It? And What Is It Used For?(Part 1)". In: *Towards Data Science. Online: https://towardsdatascience. com/auto-encoder-whatis-it-and-what-is-it-used-for-part-1-3e5c6f017726 (Zugriff: 22.02. 2020)* (2019) (page 1).

[2] Junhai Zhai, Sufang Zhang, Junfen Chen, and Qiang He. "Autoencoder and its various variants". In: *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE. 2018, pp. 415–419 (page 1).