Cookbook
On
Physical object identification in CMS experiment at CERN using ML approaches

Prepared By-

Mayur Jaisinghani
Chirag Lundwani
Orijeet Mukherjee
Neeharika Nagori

Under Guidance of-
Prof. Shashi Dugad(TIFR)
Dr. Sharmila Sengupta and Mrs. Sunita Sahu
(VESIT CMPN)

Table of Contents

1. Introduction

The Compact Muon Solenoid (CMS) is a general-purpose detector at the Large Hadron Collider (LHC). This detector is used to detect collisions between particles in LHC and generate a huge number of images of the event for further analysis. To face the new challenges of the High Luminosity LHC a new sampling calorimeter is planned to be introduced. The high-granularity calorimeter (HGCAL) is a major upgrade of CMS, and is necessary to maintain excellent calorimetric performance in the endcaps during HL-LHC operations. 8 inch Hexagonal Silicon Sensors are a key component of HGCAL with active thickness of 0.12 , 0.20, 0.30 mm. These sensors enable HGCAL to withstand an enormous level of radiation dose. The new calorimeter is equipped with 600 meters square of such silicon sensors. After the collision, the particles leave a unique energy deposit trail through the 47 layers of HGCAl, which are used for their identification. Currently, each layer is manually studied for identifying the particles generated after the collision. Our proposed system aims at making this process substantially less tedious and fast using complex neural network techniques like GNN and tools like ROOT for data handling. We would be training the model using the energy deposit form single particle Simulation using CMS software (CMSSW). Firstly, we would be converting the energy deposit in the form of effective ADC values. Then, the node values are aggregated w.r.t their Neighbors and do this till we get a good representation of nodes in the graph. Lastly, the Loss function is calculated on the objective parameters and uses this to improve the neural network. Using such techniques will make the process more machine reliant and eliminate the majority of manual components involved. Moreover, this model would be implemented in ROOT software by CERN for visualization and analysis purposes.

The project aims to leverage the power of GCN, a deep learning architecture designed to process graph-structured data, to perform regression analysis and predict the energy levels of electrons. Develop a predictive model that utilizes Graph Convolutional Networks (GCN) to accurately estimate the energy value of an electron as it passes through the various layers of HGCAL.

2. Requirements

2.1. Functional Requirements

- The Data Pre-processing model will reduce the number of points in the dataset to filter out the noise.
- Dataset conversion model will convert root type data to csv data, that will be the input for graph formation.
- Root software will be in charge of handling the huge data set competently.

- Deep learning model model of GNN will learn from the provided data trends and predict energy of electron particles.

2.2. Non-Functional Requirements

- Competent handling of huge datasets
- The system must be easy and portable
- It must sum up to be cost efficient
- The response time should be minimal

2.3. Hardware Requirements

- **OS/Laptop:** A laptop with good specifications with a minimum of 48 GB RAM is recommended along with Ubuntu/Windows as the OS. An i7 7th generation processor is also recommended.
- **GPU**: We used the hardware provided by TIFR, 8 core, 48GB RAM, 8GB GPU and RTX 1390 graphics card.
- **Storage**: 50 GB.

2.4. Software Requirements

- **ROOT**: ROOT is a framework for data processing, born at CERN, at the heart of the research on high-energy physics. It is a software toolkit which provides building blocks for Data processing, Data analysis, Data visualization, Data storage.
- **Pytorch Geometric**: PyTorch Geometric is specifically designed for tasks involving graphs, such as graph neural networks (GNNs), which are a type of deep learning model that can operate on graph-structured data, including social networks, biological networks, recommendation systems, and more.
- **NetworkX**: NetworkX is a Python library that provides tools and utilities for the creation, manipulation, and analysis of complex networks or graphs.
- **Ubuntu OS**: Ubuntu is a Linux distribution based popular operating system for cloud computing.
- **Matplotlib**: Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python. It can be used to make interactive figures and customize visual style.

- **Uproot**: Uproot is a ROOT I/O module based on pure Python. No ROOT infrastructure is needed. ROOT file can be converted into other formats frequently used in the python environment.
- **Numpy**: NumPy (Numerical Python) is a popular open-source numerical computing library for Python that provides support for large, multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays.
- **Pandas**: Pandas is a popular open-source data manipulation library for Python that provides data structures and data analysis tools for efficient and effective data handling and analysis.

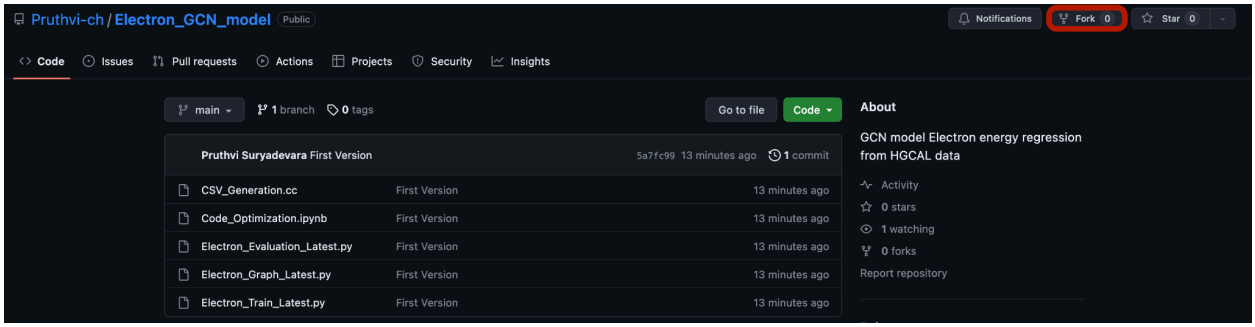
3. Python Libraries installation

- Please install the necessary libraries in a GPU enabled PC using python virtual environment.
- Use the following commands for setting up the virtual environment with name ('GCN_train'):
  - python3 -m pip install --user --upgrade pip
  - python3 -m venv GCN_train
  - source GCN_train/bin/activate
- Once activated, install the necessary libraries mentioned below.
  - Package Version
  - ----------
  - networkx 3.1
  - numpy 1.22.4
  - torch 2.0.1+cu118
  - pandas 1.5.3
  - seaborn 0.12.2
  - Sklearn 1.0
- Note: Always activate the virtual environment before training using 'source GCN_train/bin/activate'
- To deactivate the virtual environment use the command 'deactivate'.

4. Codes availability through github

- All the scripts used for training and evaluation are available through github on the following link: https://github.com/Pruthvi-ch/Electron_GCN_model

- To access the git repository, clone using ‘git clone https://github.com/Pruthvi-ch/Electron_GCN_model.git’.
- For further development of the model, please create a personal fork of the model by using the button shown in the image below.



- Any update to the scripts should be maintained using github and substantial improvements have to be propagated to the main fork using pull-requests.
- Please refer to GitHub and git user-guides for development work.

5. Steps to get started

5.1. CSV formation setup

We recommend using Google Colab’s GPU for working with and for trial and error, once the final model is implemented then you can use it for the implementation on Local Laptops. Use the same virtual environment created earlier for the Application setup as this process will be the same for any OS.

1. Use the codes from Github **“CSV_Generation.cc”** (or) Download the C++ file of code from the following link:
   Google Drive link for CSV Generation:
   https://drive.google.com/file/d/1R9wvtWI5ZJuiDhG9bpvxyLSHWut73RVG/view?usp=drive_link
2. If any changes or updation related to classes in the dataset need to be done, then you need to update the file named **“CSV_Generation.cc”** .
3. The final CSVs’ will be stored in the folder specified by the user. All the data from csvs’ will be used further in graph formation and energy prediction.
4. The input root file can be changed in the line 54, and the output folder is mentioned in the line 127. Note: the output folder (currently ‘./CSV’) has to be created before running the script using ‘mkdir CSV’)
5. Please use the command ‘g++ CSV_Generation.cc `root-config --cflags --libs` -lPhysics -o CSV_Generation.out’ (also mentioned at link) to compile the cpp script. And ‘./CSV_Generation.out’ to run the executable.

5.2. Graph Formation setup

We recommend using Google Colab's GPU for working with and for trial and error, once the final model is implemented then you can use it for the implementation on Local Laptops. Use the same virtual environment created earlier for the Application setup as this process will be the same for any OS.

1. Use the codes from Github **"Electron_Graph_Latest.py**" (or) Download the python file of code from the following link:
2. Google Drive link for Graph formation: https://drive.google.com/file/d/1R9wvtWI5ZJuiDhG9bpvxyLSHWut73RVG/view?usp=drive_link

```
file_names = glob.glob('/home/4tb_Drive_1/HGCAL_HLT/CSV/*.csv')
for file_name in file_names:
    f = int(file_name.split('_')[-1].split('.')[0])
    df = pd.read_csv(file_name)
    print(f'graph going on is for electron {f}')

    if not df.empty:  # check if the DataFrame is empty
        # Split the dataset by layer number using a for loop
        layer_data = []
        energy_event = df['E'].iloc[0]
        energy.append(energy_event)
        print("Energy for event", f, "is equal to:", energy_event)

        for layer_num in range(1, 47):  # since electron travels upto 30th layer
            layer_df = df.loc[df['Layer'] == layer_num]  # filter rows by layer number
            layer_data.append(layer_df)
    else:
        print("Empty DataFrame for event", f)
        #print(layer_data)
```

3. This part of the code is where we import the dataset in the form of CSV files. Change the path of the folder and file depending from where we wish to upload the files.
4. We need to change the **range for num_layer** for any other particle depending on the maximum layers it penetrates. If we are working with data other than energy, we need to append that data into a list similar to energy list.

```
    # Create nodes and add them to the graph
    for index, row in df.iterrows():
        graph.add_node(index, x=row['X'], y=row['Y'], layer=row['Layer'], adc=row['Eff_ADC'])

    # Connect nodes within all layers. Nodes of each layer, connected to all nodes of that layer.
    layers = df['Layer'].unique()

    for l in layers:
        nodes_in_layer = df[df['Layer'] == l].index.tolist()
        first_node = nodes_in_layer[0]
        for node in nodes_in_layer[1:]:
            graph.add_edge(first_node, node)
```

This is where the nodes and edges of the graphs are getting formed. The node features can be changed in "**graph.add_node(x, y, z……)**" We use the variable "distance" to

calculate the euclidean distance between all the node pairs of two consecutive layers and join only the least 5 distances. Along with that, we join the all the nodes of layer 1 with themselves to avoid any disjoint graphs.

```
try:
  arr = np.array(adj_matrix)
  rows, cols = np.where(arr == 1)
  adj_tensor = torch.tensor([rows, cols], dtype=torch.long)
  edge_index.append(adj_tensor)
except (ValueError, IndexError) as e:
    print(f"Skipping file: {f}. Error: {e}")
    continue
```

5. In this part we form the edge_index. For handling exceptions, you can change the conditions of except statement depending on how you wish to handle them. “**except(ValueError, IndexError) as e**:” can be changed.

```
#print("----------------------------Feature Matrix-----------------------------")
# Extract the two feature columns as a 2D numpy array
adc_data = []
with open(file_name, 'r') as file:
    reader = csv.DictReader(file)
    for row in reader:
        adc_data.append(row
```

```
# create the dictionary with node features as ADC values
node_features = {}
for i, row in enumerate(adc_data):
    node_features[i] = {'adc': float(row['Eff_ADC']), 'layer': int(row['Layer']), 'X': float(row['X']), 'Y': float(row['Y'])}
```

6. In this part, we form the feature matrix where the “**node_features[i] = {}**” can be changed depending on our requirements.

```
pruthvi@pop-os:/home/4tb_Drive_1/HGCAL_HLT$ ls
'\'                          Electron_Graph_Latest.py              Electron-timing-1.txt                 Raw_Hits_Regular_Mu_Minus_PU_000.root
 Code_Optimization.ipynb     Electron-graph-latest.txt             Electron-timing-2.txt                 Raw_Hits_Regular_Mu_Plus_PU_000.root
 code_optimization.py        Electron_Graph.py                     Electron-timing-3.txt                 requirements.txt
 CSV                         Electron_latest.pkl                   Energy-1.txt                          result-1.txt
 CSV_Generation.cc           Electron.pkl                          Energy-2.txt                          result-2.txt
 CSV_Generation.out          Electron-test-5.txt                   Energy-test.py                        result-3.txt
 CSV.tar.gz                  Electron-test-latest-3.txt            event-313.csv                         result-4.txt
 Electron_Data               Electron-test-latest-4.txt            filename.pkl                          scatter_plot.png
 Electron_dataset.py         Electron_Test_Latest.py               histogram_plot.png                    ssh_graph_formation_Root_dataset_in_csv.py
 Electron-eval-1.txt         Electron-test-latest-result-final.txt New_Data                              train-1.txt
 Electron-eval-2.txt         Electron-test-latest-result.txt       nohup.out                             train-2.txt
 Electron-eval-3.txt         Electron-test-new-1.txt               pickle-1.txt                          train-test.py
 Electron_Evaluation.py      Electron_Test_New.py                  predictions_values.pkl                Updated_Data
 Electron-graph-1.txt        Electron-test.py                      Raw_Hits_Regular_Electron_PU_000.root WeightElectronFile.pt
```

7. If any changes or updation related to classes in the dataset need to be done, please commit the changes to the git repository fork.

8. On executing this file, graphs are formed based on the concept of Euclidean distance. Which is then stored in the form of a pickle file and transferred over as input to the GCN Model for prediction of energy.

5.3. Electron energy prediction setup

We recommend using Google Colab's GPU for working with and for trial and error, once the final model is implemented then you can use it for the implementation on Local Laptops. Use the same virtual environment created earlier for the Application setup as this process will be the same for any OS.

1) Use the codes from Github "**Electron_Train_Latest.py**" (or) Download the python files of code from the following links:
   - Google Drive link for training and testing the model : https://drive.google.com/file/d/1SbwoUidAgJKOYXhYMs0hMNLnAf6DDuvl/view?usp=sharing
   - Google Drive link for Evaluating the model : https://drive.google.com/file/d/1sEDopyOiHSSpzmF95D2J7oJ3liAsfC-8/view?usp=sharing

2) If any changes or updation related to classes in the dataset need to be done, please commit the changes to the git repository fork.

3) This file takes the pickle file containing the graphs as an input and splits the data in 80:20 ratio. Thus, 80% of data is given to the training of model and testing is performed on the remaining 20% events whose energies are predicted. (described at line).

4) Now that the predictions for each epoch are being calculated and printed, the best epoch has been found out which gives the Least Root Mean Square Error (RMSE) and the model state at that epoch has been saved into a Model Weight File "**WeightElectronFile.pt**".

5) This weight file can be used for any testing dataset without actually training the model all over again. In the "**Electron_Evaluation_Latest.py**" a testing dataset was taken and the predictions were calculated using the best model state that we saved earlier from our training. Moreover, the RMSE value is being calculated and the scatter plot of actual vs predicted and Gaussian distribution of ratio of predicted by actual values is being saved in png format.

6. Electron Energy Prediction Workflow

6.1. CSV Formation

1. The purpose of this code is to process physics event data stored in ROOT files and extract specific information related to particle hits in different layers of a detector. For each event, it creates a CSV file containing the positions, energy deposits, and effective ADC (Analog-to-Digital Converter) values of the hits.

```
//Variables for adc, nHit, x, y, simhitE, .etc
int nHit_ = 0;
float X_[50000] = {0.0};
float Y_[50000] = {0.0};
float E_[50000] = {0.0};
float t_[50000] = {0.0};
uint16_t adc_[50000] = {0};
UShort_t thick_[50000] = {0};
uint16_t adc_mode_[50000] = {0};
int16_t zside_[50000] = {0};
```

2. ‘float ADC_effective(int ADC, bool mode, int thick)’: This function calculates the effective ADC value based on the input ADC count, a mode flag, and the thickness of the detector layer.

```
float ADC_effective(int ADC, bool mode, int thick){
  float scale[5] = {0.0, 1.69, 1.0, 0.62, 2.3};
  float Eff_ADC = 0.0;
  if(mode){
    if(thick==4){
      Eff_ADC = scale[thick] * (37000.0 + (165.5 * ADC)) / 44.74;
    } else{
      Eff_ADC = scale[thick] * (1320 + (55.26 * ADC)) / 2.2;
    }
  } else{
    Eff_ADC = scale[thick] * ADC;
  }
  return Eff_ADC;
}
```

3. ‘int CSV_Generation()’: The main function responsible for processing the physics event data and generating CSV files. It opens the input ROOT file, reads the particle information (such as position, energy, etc.) from different detector layers, calculates the effective ADC value, and writes the information to CSV files for each event.

```
int CSV_Generation(){
  std::cout<< "start" << std::endl;
  TFile* infile = new TFile("Raw_Hits_Regular_Electron_PU_000.root", "READ");
  // Z pisition fo all the layers
  double Z_[47] = {322.155,323.149,325.212,326.206,328.269,329.263,331.326,332.32,334.383,335.377,
87,353.375,354.369,356.757,357.751,360.139,361.133,367.976,374.281,380.586,386.891,393.196,399.501
.151,480.376,488.601,496.826,505.051,513.276};
  //Scaling of adc or energy
  double scale[] = {0, 1.69, 1, 0.63, 2.3};
  double scale_sim[] = {0, 1.69, 1, 0.63, 0.113};
  int in_id = 11;
  int n_layers = 47;
  int n = 20000; // number of events to analyze
  //TTree **intree = new TTree*[n_layers];
  TTree *intree = new TTree;
  //Variables for adc, nHit, x, y, simhitE, .etc
  int nHit_ = 0;
  float X_[50000] = {0.0};
  float Y_[50000] = {0.0};
  float E_[50000] = {0.0};
  float t_[50000] = {0.0};
  uint16_t adc_[50000] = {0};
  UShort_t thick_[50000] = {0};
  uint16_t adc_mode_[50000] = {0};
  int16_t zside_[50000] = {0};

  int n_gen = 0;
  float eta_[100] = {0.0};
  float phi_[100] = {0.0};
  float pT_[100] = {0.0};
  float pz_[100] = {0.0};
  int id_[100] = {0};
  float Egen_[100] = {0.0};
```

4. For each event in the input ROOT file, the program generates a separate CSV file (named "Event_#.csv", where "#" is the event number) in a subdirectory called "CSV." Each CSV file contains the X and Y coordinates of particle hits, the detector layer number, the calculated effective ADC value, and the corresponding particle energy.

```
pruthvi@pop-os:/home/4tb_Drive_1/HGCAL_HLT/CSV$ cat Event_3.csv
X,Y,Layer,Eff_ADC,E
49.8855,-93.6113,1,97.34,467.789
5.94478,-103.307,1,85.56,467.789
32.0512,-105.109,1,107.88,467.789
34.2317,-106.31,1,117.8,467.789
35.2719,-106.911,1,1275.25,467.789
36.3122,-107.512,1,215.14,467.789
37.3524,-106.911,1,215.76,467.789
31.0609,-108.199,1,136.4,467.789
32.1012,-108.799,1,138.26,467.789
32.1012,-107.598,1,128.34,467.789
34.1817,-107.598,1,839.198,467.789
35.2219,-108.199,1,329.84,467.789
12.9954,-37.0961,2,111.54,467.789
49.8855,-94.8125,2,158.1,467.789
```

6.2. Graph formation

1. In this the first step is forming the nodes and giving them their features. We add the X and Y coordinates, Effective ADC and layer number.

2. After that we calculate the euclidean distance between each pair of nodes between consecutive layers. We join the nodes with the 5 least distances and also all the nodes of layer 1 are connected with themselves.

3. Once edges are formed, we make the adjacency matrix and feature matrix. Both matrices are converted to tensors and then store into a data list which can be passed to the GCN model.

4. Finally, the python script **"Electron_Graph_Latest.py"** is to be executed using the command -

```
$ nohup python3 -u Electron_Graph_Latest.py >> Electron-graph.txt &
```

6.3. Loading the graph data from the pickle file for splitting

1. When the "**Electron_Test_Latest.py"** has been run using this command -

```
$ nohup python3 -u Electron_Test_Latest.py >> Electron-test.txt &
```

2. The entire data of all the graphs which was saved in the pickle file from "**Electron_Graph_Latest.py**" will be loaded in a new data list.

```
file_path = "/home/4tb_Drive_1/HGCAL_HLT/Electron_latest.pkl"

 # Load the data from the pickle file
with open(file_path, "rb") as f:
   data_list = pickle.load(f)
```

3. The loaded data is stored in the variable 'data_list'. The data is split into a training set and a test set. The first 4/5 (80%) of the data is used for training, and the remaining 1/5 (20%) is used for testing.

4. Here, the split for training is 0-9000 and for testing is 9000-11,093.This is because of irregularity of the number of graphs

```
train_data = data_list[0:9000]
test_data = data_list[9000:11093]

# 80:20 split for 'n' number of graphs
#num_test_graph=20000
#train_data = data_list[0:int(4*num_test_graph/5)]
#test_data = data_list[int(4*num_test_graph/5):num_test_graph]
```

6.4. GCN Model

1. The training dataset has been fed to the GCN model for training.

2. A Graph Convolutional Network (GCN) model is defined using the torch.nn.Module class. The model consists of two GCNConv layers, followed by two fully connected (Linear) layers for regression. The activation function used between layers is ReLU.

3. Training parameters such as the number of features, hidden channels, learning rate, and the number of epochs are defined which can be changed accordingly.

```
class Net(torch.nn.Module):
    def __init__(self, num_features, hidden_channels):
        super(Net, self).__init__()
        self.conv1 = GCNConv(num_features, hidden_channels)
        self.conv2 = GCNConv(hidden_channels, hidden_channels)
        self.fc1 = Linear(hidden_channels, hidden_channels)
        self.fc2 = Linear(hidden_channels, 1)  # Single output node for regression
        self.reg_lambda = reg_lambda

    def forward(self, x, edge_index, batch):
        x = F.relu(self.conv1(x, edge_index))
        x = F.relu(self.conv2(x, edge_index))
        x = global_add_pool(x, batch)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

```
reg_lambda = 0.0001

# Define the training parameters
num_features = 4
hidden_channels = 16
lr = 0.0001
epochs = 1000

# Create the GCN model and optimizer
model = Net(num_features, hidden_channels)
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

# Create data loaders for the training and testing datasets
batch_size = 1
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_data, batch_size=batch_size, shuffle=False)

q = 0.5  # Choose the desired quantile value
criterion = torch.nn.SmoothL1Loss()
```

6.5. Training and Testing using GCN model

1. The model is trained using a loop that runs for the specified number of epochs. Within the loop, the model is set to training mode, and for each batch of training data, the optimizer's gradients are zeroed, forward pass is performed, and the loss is calculated using the Smooth L1 loss function. L1 regularization is also applied. The loss is then back propagated, and the optimizer's step is taken.

2. After training for each epoch, the model is set to evaluation mode. The code then iterates through the test data using the test data loader and calculates predictions on the test data. The predicted values and actual labels are stored in separate lists. The loss for the test data is also calculated.

6.6. Calculation of Tracking Metrics

1. The minimum loss, minimum mean squared error (MSE), and their corresponding epochs are tracked during training.
2. After training is complete, the code calculates the minimum mean squared error and its corresponding root mean squared error (RMSE) for the test data.

6.7. Saving the Model's Weight File for Best Tracking Metrics

1. Since, the performance metrics of all the epochs are calculated, the best training epoch can be determined.

2. For this epoch, a model weight file has been created (i.e "**WeightElectronFile.pt**") which saves the state of GCN model of the best epoch.

```
if(mse<min_mse):
  min_mse=mse
  min_mse_epoch=epoch+1
  torch.save(model,'/home/4tb_Drive_1/HGCAL_HLT/WeightElectronFile.pt')
```

6.8. Evaluation using Model's Weight File

1. In the "**Electron_Evaluation.py**" file, the model weight file can be used for any testing dataset to make predictions without actually training the model all over again. For running this file, the command used will be -

```
$ nohup python3 -u Electron_Evaluation.py >> Electron-eval.txt &
```

2. Firstly, A testing dataset has been taken (i.e here, we load a specific number of graph data from the same pickle file which contained all the graph data).

```
with open('/home/4tb_Drive_1/HGCAL_HLT/Electron_latest.pkl', 'rb') as f:
    data_list = pickle.load(f)

test_data=data_list[9000:11093]
```

3. Further, the GCN model has been defined just like the one used previously.

4. The code loads the pre-trained model weights from the file "**weightelectronfile.pt**" using torch.load(). This model has been previously trained and saved.

5. The model is set to evaluation mode using model.eval(), which disables the gradient computation and dropout during testing.

6. Now, the predictions are made for the testing dataset using the model weights assigned.

7. The code calculates the mean squared error (MSE) and root mean squared error (RMSE) using mean_squared_error from sklearn.metrics.

8. The code saves the scatter plot and the histogram with the Gaussian distribution in png format.

7. Evaluation Measures -

Result of last i.e 1000th epoch

```
Actual label was-> tensor([193.7190]) Predicted output: tensor(188.9514)
Actual label was-> tensor([819.3750]) Predicted output: tensor(809.6274)
Actual label was-> tensor([1296.2800]) Predicted output: tensor(1276.9955)
Actual label was-> tensor([480.3500]) Predicted output: tensor(471.2504)
Actual label was-> tensor([1040.1000]) Predicted output: tensor(1036.0662)
Actual label was-> tensor([1005.8700]) Predicted output: tensor(978.1016)
Actual label was-> tensor([362.8840]) Predicted output: tensor(364.6880)
Actual label was-> tensor([736.8990]) Predicted output: tensor(718.9254)
Actual label was-> tensor([212.6040]) Predicted output: tensor(207.2046)
Actual label was-> tensor([557.2780]) Predicted output: tensor(562.2927)
Actual label was-> tensor([370.2070]) Predicted output: tensor(374.5566)
Actual label was-> tensor([979.0430]) Predicted output: tensor(961.7391)
Actual label was-> tensor([509.1170]) Predicted output: tensor(479.7744)
Actual label was-> tensor([223.1430]) Predicted output: tensor(222.4286)
Actual label was-> tensor([191.8020]) Predicted output: tensor(189.4080)
Actual label was-> tensor([294.1990]) Predicted output: tensor(287.5164)
Actual label was-> tensor([250.2490]) Predicted output: tensor(247.6243)
Actual label was-> tensor([173.4910]) Predicted output: tensor(170.6236)
Actual label was-> tensor([369.9250]) Predicted output: tensor(372.8518)
Actual label was-> tensor([542.6390]) Predicted output: tensor(535.7668)
Actual label was-> tensor([698.6610]) Predicted output: tensor(674.1397)
Mean Loss: 9.75
Mean Squared Error (MSE): 217.46
Root mean Squared Error (RMSE) : 14.75
Testing Time for Epoch: 40.312 seconds
```

Best Results obtained at 542th epoch.

```
Min Mean Squared Error (MSE): 134.93
Min Root Mean Squared Error (RMSE): 11.62
min loss epoch is  542
min rmse epoch is  542
Average Training Time per Epoch: 47.546 seconds
Average Testing Time per Epoch: 47.546 seconds
```

8. Contact Details

- Mayur Jaisinghani (2019mayur.jaisinghani@ves.ac.in)
- Chirag Lundwani (2019chirag.lundwani@ves.ac.in)
- Orijeet Mukherjee(2019orijeet.mukherjee@ves.ac.in)
- Neeharika Nagori (2019neeharika.nagori@ves.ac.in)