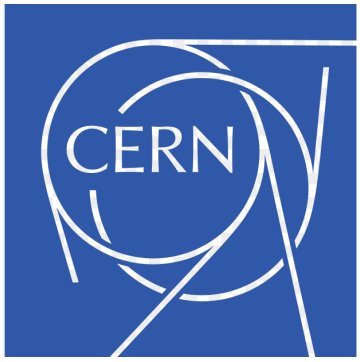
Machine Learning Models for Data Quality Monitoring



Project Report
CERN, Geneva

Submitted by

Sarah AlKhudari
CERN Summer Student, 2024

Supervised by
Dr. Roberto Seidita

Co-Supervised by
Dr. Antonio Vagnerini

1 Abstract

The following report examines the CMS Data Quality Monitoring (DQM) system and the implementation of machine learning models to improve anomaly detection. The project focuses on building an autoencoder model to achieve this goal. By training the model on what is considered "GOOD" data, extracted from CMS runs, the autoencoder is able to recognize normal patterns. Specifically, the model analyzes missing transverse energy (MET) significance in each run, identifying deviations that signal potential anomalies. This approach enhances the efficiency of data quality monitoring, aiding physicists in maintaining the integrity of the collected data.

Contents

2 Introduction

The European Organization for Nuclear Research, also known as CERN, was established in 1954 and collaborates with universities and scientists worldwide. Located near Geneva, on the border between Switzerland and France, CERN has a long history of groundbreaking particle physics experiments. While it is most recently known for the discovery of the Higgs boson within the Large Hadron Collider (LHC), CERN has been responsible for many other significant scientific advancements that have transformed our understanding of physics. These include the development of the World Wide Web, the discovery of the W and Z bosons, and ongoing research supporting the Standard Model of particle physics.

Figure 2: Map of LHC

The Large Hadron Collider (LHC) is currently the world's largest and most powerful particle accelerator. In short and simple terms, the LHC accelerates particles (mainly protons) in both clockwise and counterclockwise directions. When protons are accelerated to near the speed of light, they are made to collide. The kinetic energy of these collisions is partially converted into new particles that travel in all directions from the interaction point. These particles are studied using particle detectors. The four main detectors built to measure the products of the collision are ATLAS (A Toroidal LHC Apparatus), ALICE (A Large Ion Collider Experiment), CMS (Compact Muon Solenoid), and LHCb. Out of these, CMS is the focus of this work.

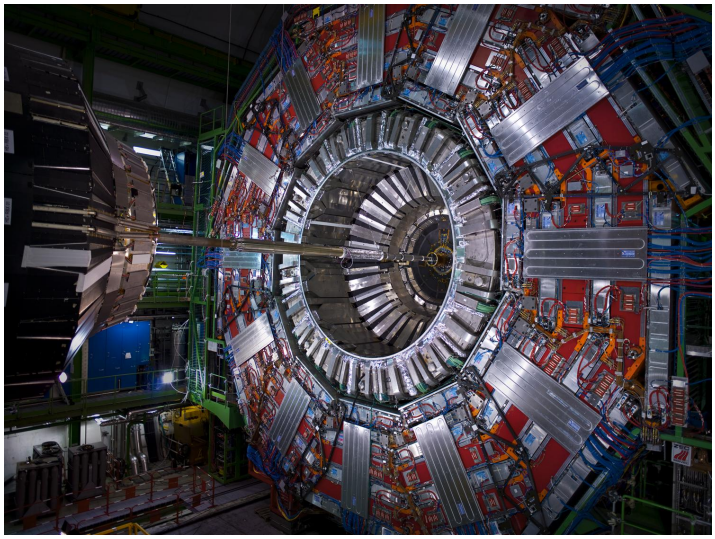


Figure 3: CMS detector

The main purpose of the CMS detector is to detect all particles produced during collision events in the LHC, as well as measure their kinematic properties. The CMS detector is capable of reconstructing and precisely measure the momentum of muons, electrons, photons, and hadrons produced during collisions. Each layer of the detector plays a specific role in studying the energies and momenta of the particles produced in these collisions, as will be explained in the next section. This detailed analysis enables the understanding of fundamental physics processes and discovering new phenomena.

2.1 Problem Statement

The large volume of data from the detectors often includes anomalies, which will be discussed further in this report. The current method of data monitoring in the DQM offline center at CMS is primarily human-based. Machine learning, having become widely utilized in high-energy physics over the last decades, provides a powerful toolset for anomaly detection within the DQM workflows of CERN and its experiments. While artificial intelligence is often discussed in popular contexts, machine learning is firmly based on statistical and mathematical techniques. It has been applied at CERN for many years, long before the recent rise of AI in other industries. By implementing machine learning models, the process of detecting anomalies can be simplified for shifters, allowing them to more efficiently flag data for analysis by physicists. The next step is to determine which machine learning model is best suited to achieve this goal.

3 The Compact Muon Solenoid and the Large Hadron Collider

3.1 The Large Hadron Collider

To better understand the function and workings of the CMS, we first need a proper understanding of the LHC. The Large Hadron Collider is injected with two beams of protons in opposite directions. These protons are initially formed from hydrogen atoms, which are stripped of their electrons. The protons are then injected and pre-accelerated outside the LHC using smaller accelerators, gaining speed and energy. The protons are accelerated within electric fields, gaining a "boost", meaning increasing speed. Once injected into the LHC, the protons are kept in a circular path by superconducting magnets that bend the path of the particles. Reaching a maximum energy of up to 6.8 TeV per beam, the beams continue to travel in opposite directions within the accelerator. There are four main detectors connected to the LHC: ATLAS, CMS, ALICE, and LHCb. These detectors are used to study the "events" and particles produced in each collision.

3.2 The CMS Detector

With about 100 billion protons in each bunch, and over 2000 bunches in each beam, the bunches collide approximately 25 million times every second. These colliding particles are then studied in the CMS. The specialty of the CMS, compared to other detectors, lies in its layers and their unique functions, as will be discussed later. At 15 meters high and 21 meters long, the detector was built on the surface and slowly assembled fully by being lowered underground in slices in Cessy, France. It uses a 13-meter-long, 5.9-meter-diameter superconducting solenoid to generate a 4 Tesla magnetic field. This solenoid magnet is specialized in bending charged particles sufficiently to measure their momentum accurately.

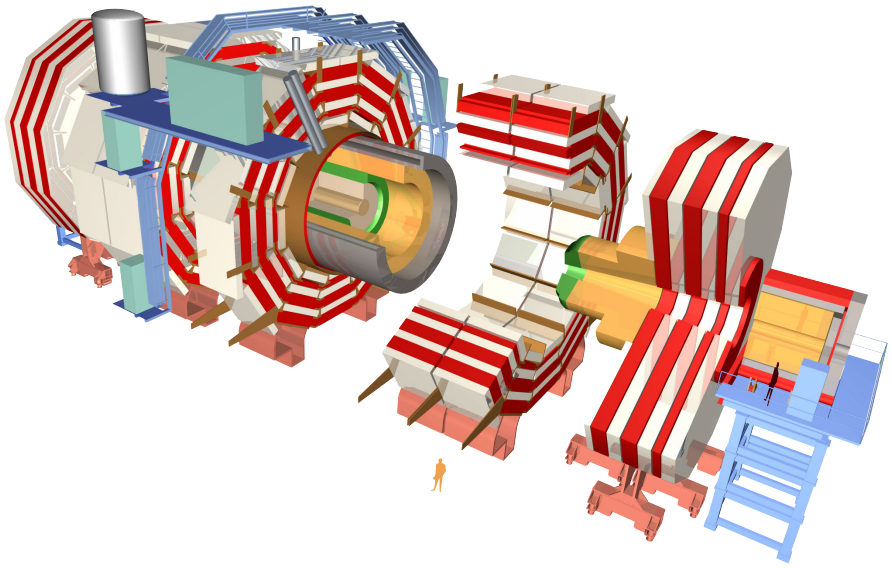


Figure 4: CMS Detector

The Detector Layers

The CMS detector is designed with cylindrical symmetry around the collision point of the particle beams. It is divided into a central part (the "barrel"), consisting of concentric layers of specialized detectors, and one closing section on each side (the "endcaps"). Each layer of the CMS plays a crucial role withing particle physics.

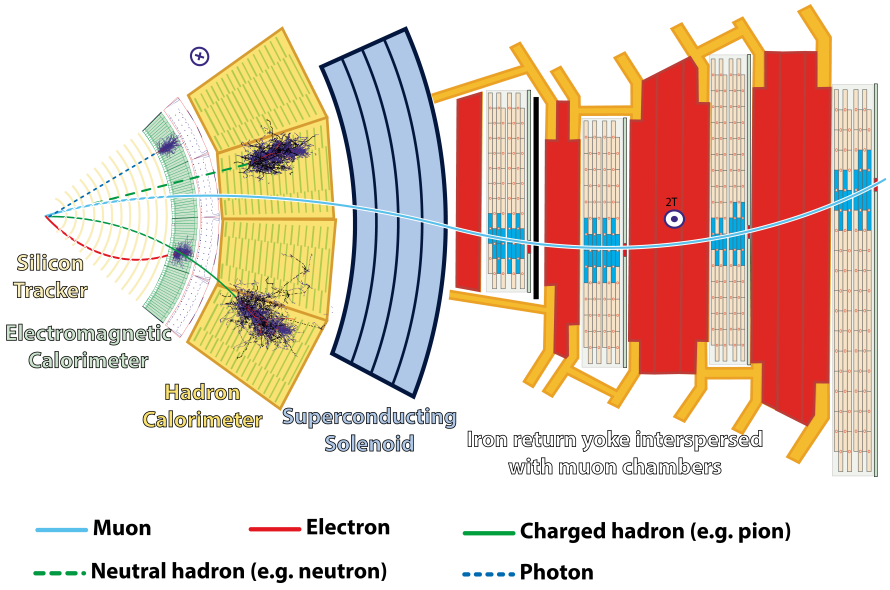


Figure 5: CMS detector Animated

1. **The Solenoid Magnet:** The main component to be discussed within the CMS is the 13-meter magnet. The solenoid magnet generates a strong magnetic field, made of a niobium-titanium alloy conductor that is superconducting at 4 K, inside an aluminium structure. By bending the paths of charged particles produced in collisions, the magnetic field of the solenoid helps in analyzing the collision products. The momenta of these collision products within the CMS are measured using the magnet. The curvature of the particles correlates with their momentum.

$$\kappa = \frac{1}{R} = \frac{qB}{p}$$

2. **The Silicon Tracker:** The innermost part of the detector tracks the paths of charged particles such as electrons and protons. It performs vertex reconstruction.

3. **Electromagnetic Calorimeter (ECAL):** The ECAL measures the energy of incoming electrons and photons, which produce a shower in the calorimeter. When high-energy electrons or photons enter the ECAL, they produce showers. The ECAL absorbs this energy and measures it for further study.

4. **Hadron Calorimeter (HCAL):** Particles that do not interact with the ECAL react and create showers in the HCAL to measure the energy of hadrons (composite particles that contain quarks). The HCAL absorbs the released energy, which is crucial for detecting hadron particles and distinguishing them from other particles.

5. **Muon Chambers:** Muons are similar to electrons but are heavier and penetrate more easily through the CMS layers. The muon detectors is the only part placed outside of the solenoid magnet. Muon chambers are specifically dedicated to tracking and measuring these muons.

3.2.1 CMS Data Acquisition and Triggers

Data first runs through a Level 1 Trigger that is hardware-based. It operates quickly and makes decisions at a high rate of 40MHz to process readout data in real-time. The Level 1 Trigger is made up of the Calorimeter Trigger, Muon Trigger, and a Global Trigger. The trigger's algorithm then selects which events are interesting for physics. Once accepted, the events are sent to the High-Level Trigger System (HLT). The Level 1 Trigger uses customized FPGAs, ASICs, DSPs, and more.

Once they reach the HLT, running on GPUs and CPUs, the events undergo a more intricate process of event reconstruction. The HLT is built using over 13,000 CPUs and primarily relies on a full software

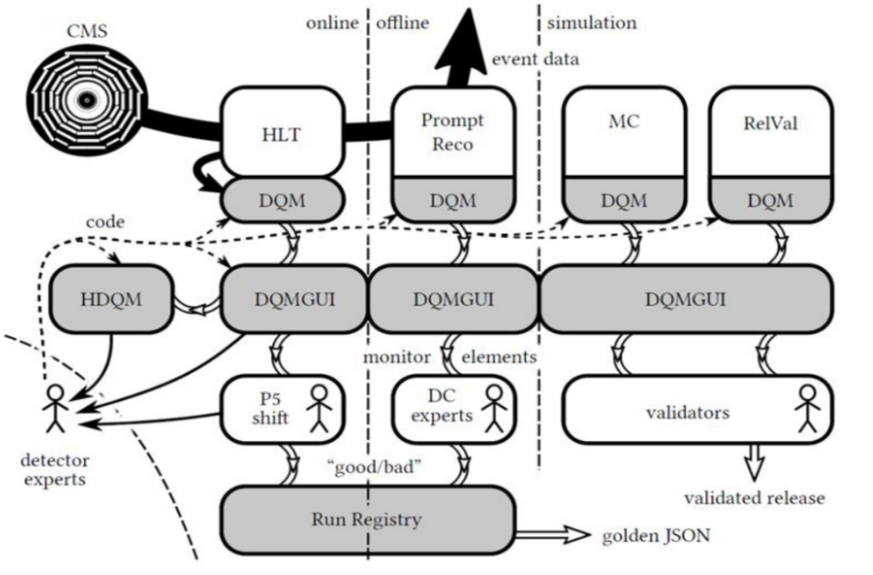


Figure 6: Data Processing Pipeline in CMS

farm. Combining both triggers helps to reduce the frequency of data acquisition from 40 MHz to 4000 Hz.

After passing through the HLT, data goes through the Data Quality Monitoring (DQM) System. Given the complexity of a detector like CMS, a number of faults can result in corrupted or incomplete data, so the DQM system is crucial for identifying and diagnosing these issues in real-time to ensure the quality and integrity of the data being collected. Both online and offline components play important roles in finalizing whether the data can be used in physics analyses. Starting with online DQM, the data is monitored in real-time. Once the online DQM receives data from runs, it must make quick decisions based on the available information. Small parts of the data are displayed in plots and for each subsystem during the run; these help shifters decide whether to mark the data as GOOD or BAD.

Finally, the last step before analysis is carried out by the offline DQM. This role focuses on obtaining the fully reconstructed data and verifying that the online DQM decisions were accurate. Offline DQM teams have started using DIALS, a GUI that stores reconstructed data from runs, with a prompt reconstruction done in 3 hours and a full reconstruction in 24 hours. Offline DQM focuses on reviewing the complete data and ensuring that the flags (GOOD/BAD) match the information provided, allowing the data to be sent for physics analysis.

4 Data Quality Monitoring (Offline)

Offline DQM plays a crucial role after the CMS data acquisition and trigger system. Once the runs are pushed into the Run Registry by the shifters on the Online DQM, the Offline DQM steps in after a set period of time. It first analyzes a prompt reconstruction of the data, followed by a full reconstruction. Experts then study the fully reconstructed data to monitor the detectors' particle reconstruction performance and overall health. Based on the analysis, the data is officially labeled as "GOOD" or "BAD" and pushed into the Run Registry.

"BAD DATA" indicates anomalies, such as issues within the calibration of the detector for each run or problems with detector components. This classification helps identify and address potential problems within the system.

During a run of the CMS, the detector is on for a time that can range from a span of minutes to several hours. During the time the CMS is on, the detector is collecting data from the collisions. "A run's duration depends on both the stability of the LHC and the CMS system. If any issues arise or problematic results are detected, the run may be affected.

Lumisections are time frames of approximately 23.3 seconds, used to facilitate data analysis for quality and detector health. Segmenting data into these smaller intervals allows for ore accurate monitoring compared to analyzing an entire run in one histogram. This approach makes it simpler to identify and select useful lumisections for research while discarding those that are not. By restricting problematic data to just a handful of lumisections instead of an entire run, we can "throw away" much less good data, preserving more valuable information for analysis.

Each lumisection has multiple elements being monitored. Elements in the context of DQM are simply distributions of physical observables. These monitored elements are studied by the offline DQM team. Its important to understand in data taking, some of these elements may be off or not meet the necessary requirements; meaning an anomaly can be detected.

5 Machine Learning

Within this project, the goal is to implement machine learning models into the offline DQM process to simplify the analysis and monitoring of data for physicists. Before discussing how machine learning will be implemented, it's important to understand what machine learning is and the specific type that will be used.

Machine learning is a branch of artificial intelligence that focuses on developing algorithms that can process data and make predictions or decisions based on statistical patterns. While deep learning is a subcategory of machine learning, it involves more complex models, often with multiple layers of neural networks. Unlike traditional machine learning models, which can include algorithms like decision trees or support vector machines, deep learning models are typically more intricate and may require more computational resources. Although deep learning and traditional machine learning models can operate autonomously after being trained, they still require regular updates, monitoring, and maintenance to remain effective. This is necessary to ensure the models continue to perform well as new data becomes available, similar to the upkeep required for models like ChatGPT.

Machine learning is a process where algorithms are trained on data to identify patterns and make predictions. Initially, the model's performance may be limited, but as it is exposed to more data and iteratively adjusted, its accuracy improves over time.

5.1 Autoencoders

When discussing the type of machine learning model to use in the project, autoencoders were found to be most effective for anomaly detection. Anomaly detection involves training the model to recognize expected patterns so that when something deviates from these patterns, it is flagged as an "anomaly."

Autoencoders are a type of machine learning model that compress input data through encoding layers and then attempt to reconstruct it using decoding layers. The goal is for the model to recreate the input as closely as possible, minimizing the difference between the original and the output. This difference, or reconstruction error, is typically measured by mean squared error (MSE). When normal data is processed, the MSE remains low. However, when unusual or anomalous data is fed in, the model struggles to recreate it accurately, causing the MSE to spike. These spikes are often visualized in plots, allowing users to clearly spot deviations and identify potential anomalies.

5.2 Supervised Learning vs. Unsupervised Learning

Machine learning has multiple methods; the two popular being "supervised" and "unsupervised". Supervised learning involves training a model on labeled data, where each input is paired with the correct output. For example, when creating a classifier to differentiate between dogs and cats, a supervised model would be trained on datasets containing images of dogs and cats, each labeled as 'dog' or 'cat'. This labeled data allows the model to learn the distinguishing features of each category.

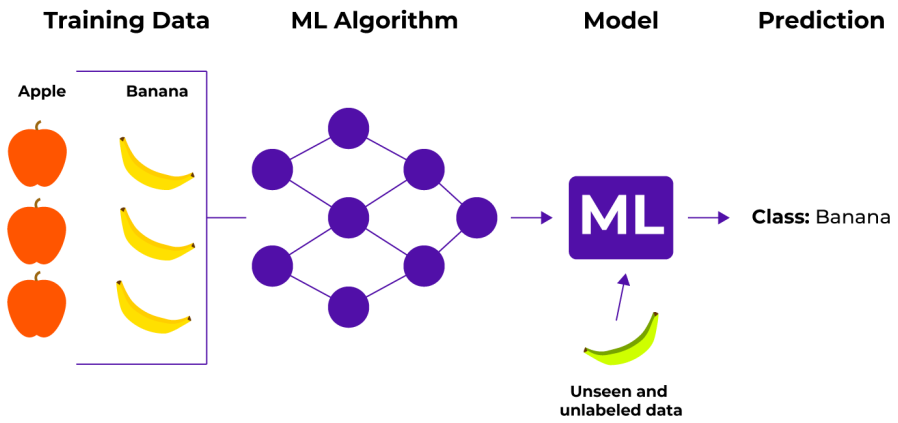


Figure 7: Supervised Learning Data Diagram

Figure 7 shows a diagram explaining the process of supervised learning. In this process, the model

is trained on a labeled dataset containing examples of apples and bananas. After training, the model is tested by being given an example it has never seen before, and it makes a prediction based on the knowledge gained during training. The model's ability to use labeled data during training is what defines it as supervised learning.

Unsupervised learning involves training a model on unlabeled data, where no specific labels describe what the data represents. In this approach, the model identifies patterns and similarities within the data, allowing it to group or categorize the data into clusters based on those patterns without prior knowledge of the categories.

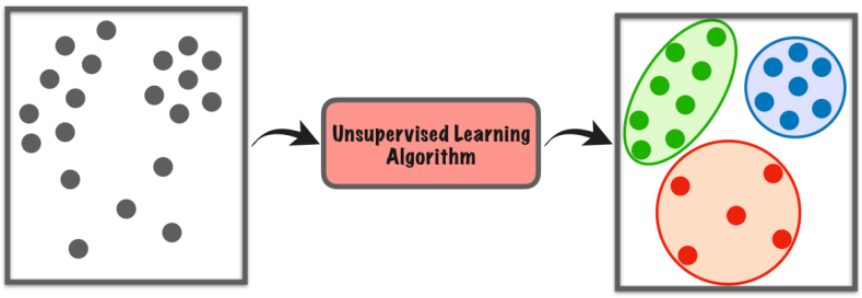


Figure 8: Unsupervised Learning Data Diagram

Figure 8 illustrates the process of unsupervised learning. Unlike supervised learning, the model here does not have predefined labels to guide its training. Instead, the model identifies patterns and similarities within the data and organizes them into groups or categories. This type of training is more flexible, as it allows the model to work with unlabeled data, though overfitting can still occur if the model becomes too closely fitted to the specific patterns in the training data.

In machine learning, overfitting refers to a model that has been trained too specifically on the details of the training data, causing it to perform poorly on unseen data. While unsupervised learning deals with unlabeled data and focuses on identifying patterns or structures, it is not inherently less prone to overfitting. In fact, overfitting can still occur if the model becomes too finely tuned to the specific patterns or noise in the data, even without predefined labels.

5.3 Choosing Unsupervised Learning

Understanding the difference between learning types is crucial when deciding which to use in the model. After careful consideration, it was concluded that unsupervised learning would be the most effective choice for several reasons.

When considering the use of supervised learning to classify data as 'GOOD' or 'BAD', several potential issues arise. One significant challenge is the inherent complexity and variability of the data from the detectors. The behavior of particles and the scale of measurements can vary greatly, making it difficult to create a comprehensive labeled dataset that covers all possible scenarios. If a model is trained on a specific set of 100 'GOOD' and 100 'BAD' runs, it becomes tailored to those specific conditions. However, in the context of DQM at CMS, this approach is limited because it's impossible to anticipate every type of 'BAD' data that could arise in the future. Machine learning models are only as good as the data they are trained on, and a supervised model may fail to correctly classify new, unseen types of 'BAD' data.

In contrast, unsupervised learning is a better approach in this context because it focuses on defining what 'GOOD' data looks like. This method allows the model to flag deviations from the norm as potential anomalies, without needing a comprehensive labeled dataset of all possible 'BAD' conditions. This makes unsupervised learning more robust for the dynamic and unpredictable nature of CMS data.

Additionally, an unsupervised learning model might offer a more effective approach in this context, depending on the nature of the data and the goals of the task. By training the model exclusively on 'GOOD' data, it can establish a baseline of what normal data looks like. When 'BAD' data is introduced, the model, having been conditioned to recognize 'GOOD' data, will produce a higher error

rate, indicating an anomaly. This approach could enhance the detection of anomalies and improve the overall monitoring process.

6 Methodology

When first introduced to this project and its goals, several issues arose, one of which was the lack of experience in building and creating machine learning models. The first step in any machine learning task is understanding the underlying mathematics to fully grasp whether the code is working correctly. This was achieved by first breaking down the fundamentals of machine learning, one of which is "weights." In machine learning, weights refer to the parameters within neural networks that adjust the strength of connections between neurons.

When building machine learning models like autoencoders, there are three key parts: the input layer (which encodes the data), the hidden layers (which represent the compressed version of the data), and the output layer (which decodes the data back to its original form). These layers are made up of neurons, and the connections between them are called weights. The weights are essential for learning—they start with random values and are gradually adjusted as the model is trained.

To be more precise, the activation of each neuron in a layer depends on the activations of the neurons in the previous layer. This happens through a process where the inputs from the previous layer are multiplied by the weights and then passed through an activation function. The process can be described with the following formula:

$$a_j = f\left(\sum_i w_{ij} a_i + b_j\right)$$

Here, each neuron's activation (a_j) is calculated by adding up the contributions from the neurons in the previous layer (a_i), multiplied by their corresponding weights (w_{ij}), and a bias term (b_j). The function f, known as the activation function, determines how the neuron reacts to this sum.

During training, the model learns by adjusting these weights to reduce the difference between the original data and the data reconstructed by the autoencoder. Over time, this helps the model better understand the important features of the data.

6.1 Programs and Software

The programs and software used within the project must be flexible and be able to work with in different places. The project uses a range of python libraries mainlyTensorFlows, Keras, Optuna, and Matplotlib

6.1.1 Keras, Optuna, and Matplotlib

Keras

TensorFlow's Keras was used to construct the machine learning model. TensorFlow, an open-source platform by Google for machine learning developers, provides access to high-level APIs for developing and training models. Compared to other platforms, Keras is simple and user-friendly when it comes to defining, developing, and training models.

In this project, the Sequential API was used to build an autoencoder. The Sequential model allows for stacking layers in a linear fashion, which is suitable for creating the structure of an autoencoder. The autoencoder itself consists of an encoder, a latent space (hidden layer), and a decoder. However, it's important to note that while a Sequential model is a convenient way to implement an autoencoder, not all autoencoders are built using the Sequential API, and not all Sequential models are autoencoders.

```python
def train_autoencoder_sequential(normalized_inputs, encoding_dim, encoding_dim_2, learning_rate, batch_size, epochs=2000):
    # Define the Sequential model
    model = keras.Sequential()

    # Add layers to the model
    model.add(layers.InputLayer(input_shape=(normalized_inputs.shape[1],)))  # Define the input layer
    model.add(layers.Dense(encoding_dim, activation='relu', activity_regularizer=tf.keras.regularizers.l2(learning_rate)))  # Encoder layer
    model.add(layers.Dense(encoding_dim_2, activation='sigmoid'))  # Latent space layer
    model.add(layers.Dense(encoding_dim, activation='sigmoid'))  # Decoder layer
    model.add(layers.Dense(normalized_inputs.shape[1], activation='sigmoid'))  # Output layer

    # Compile the model with Adam optimizer and mean squared error loss
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate), loss='mse')
    model.summary()

    # Define early stopping to prevent overfitting
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

    # Train the autoencoder model
    history = model.fit(normalized_inputs, normalized_inputs, epochs=epochs, batch_size=batch_size, verbose=1, validation_split=0.2, callbacks=[early_stopping])
    model.save('model.h5')
    # Return the trained model
    return model, history
```

Figure 9: Sequential Model

Optuna

Along with Keras, the model would use the Optuna library. Optuna, an open-source framework, is used to find the best hyperparameters in machine learning models. Hyperparameters are considered the settings or parameters made to a machine learning model that are not learned. They influence the learning process and the model's performance. An example of a set hyperparameter would be the learning rate of a model. It's known for its high flexibility and various uses in machine learning libraries, in this case, Keras and TensorFlow.

```python
# Create and run the Optuna study
study = optuna.create_study(direction='maximize')  # maximizing the metric
study.optimize(objective, n_trials=50)

# Print the best parameters
print("Best hyperparameters: ", study.best_params)
```

The following code snippet above uses Optuna to find the best hyperparameters. It first initializes the core object called a study to perform the optimization, with the goal of maximizing the objective function to achieve the most accurate outcome possible. The study.optimize function is then called with the objective function, which defines what data is to be optimized and suggests hyperparameters. The optimization process is set to run for 50 trials, during which the hyperparameter set is adjusted based on the suggestions from the objective function. After completing all the trials, it prints the best hyperparameters that resulted in the best performance, such as minimizing the loss if the objective was to reduce Mean Squared Error (MSE) The following code is calculating the MSE of runs' lumisections.

Matplotlib

Matplotlib is a Python library used for visualizing data in graph format. The library is used multiple times within the project to visualize the data and ensure the reconstructed output matches the initial input data. Matplotlib is also used to show the MSE (Mean Squared Error) of each run through the model and to display the epoch at which early stopping is triggered.

```python
plt.plot(auto_output[ls])
plt.plot(normalized_input[ls])
plt.xlabel('Bin')
plt.ylabel('Value')
plt.title(f'Autoencoder Output for Lumisection {ls}')
plt.show()
```

The following code is an example of how matplotlib is used to visualize the inputted data after being normalized.

6.1.2 DIALS and CMS Run Registry

After selecting the necessary libraries, the next step was to connect to the data that will be used. As discussed earlier, the project focuses on using data extracted from the CMS detector.

CMS Run Registry

The CMS Run Registry is the final component of the offline DQM (Data Quality Monitoring) system. It serves as a database that stores metadata for each run recorded by CMS. This includes, but is not limited to, the DQM quality flag, which helps in assessing the quality and usability of the data collected during each run. To train the model, it is crucial to use data considered "GOOD," which can be found in the CMS run registry. It's important to use the most recent runs for training to maintain consistency in the age of the data. Training the model on older runs alongside more recent ones can introduce problems, as differences in data, calibrations, and settings between these runs could lead to inconsistencies and spikes in reconstruction performance. The data varies because the detector undergoes constant aging and changes over time.

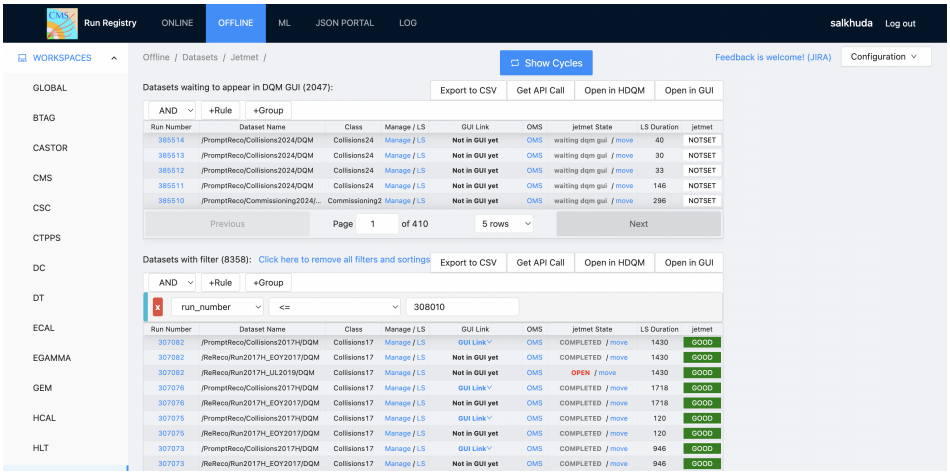


Figure 10: CMS Run Registry UI

Run 380360 was chosen as the test run after the model is trained. To ensure effective training, it was important to use runs close to 380360 that were labeled as "GOOD." Therefore, any runs within the range of 360000 to 390000 labeled as "GOOD" were used for training and testing.

DIALS

DIALS, created by Gabriel Moreira, a member of the central DQM team, provides access to a subset of the distributions used in the CMS DQM pipeline with per-lumisection granularity for all registered CMS plots. It collects, organizes, and shares this data through a user-friendly web interface, and more importantly, an API to fetch data, making it accessible even to those outside the DQM team for analyzing past data, understanding detector health, and conducting studies.

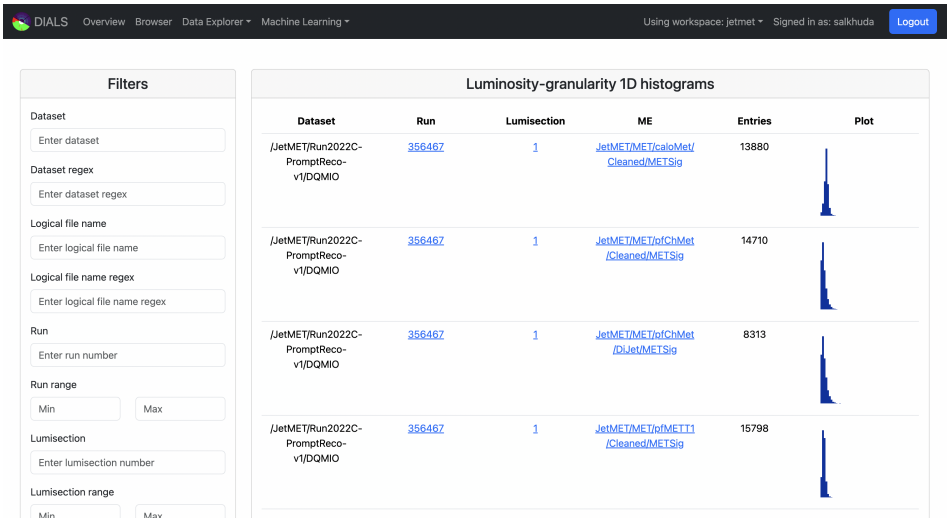


Figure 11: Dials UI

The model access the DIALS program through API, extracting any RUN data as necessary, specifically data pertaining hadronic jets and missing transverse momentum (MET), as will be explained in the following.

6.2 What Data is being trained on?

As previously mentioned, the detector collects a variety of data for the DQM, including detector health, particle reconstruction, and physics studies. When developing a machine learning model to detect anomalies in this data, it's important to remember that each run measures many different elements. The current model will focus on studying and detecting anomalies specifically in one of these monitored elements within CMS runs: hadronic jets and MET.

Jets and MET

Jets refers to the sprays of particles produced when particles are scattered during collisions, while MET (Missing Transverse Energy) is the missing energy in the transverse plane. Monitoring these measurements provides shifters with valuable information about the health of the detector and the calibration of its subsystems. Specifically, the data extracted from JetMET will focus on METSig.

METSig refers to MET Significance. As mentioned, MET is the missing transverse energy within a collision. Initially, the transverse plane's total energy is expected to be zero, and theoretically, it should remain zero after the collision, as particles spray in opposite directions, balancing out. However, this isn't always the case. MET significance calculates the significance of the imbalance in the transverse energy observed by the detector after a collision. It measures the deviation from the expected zero energy balance, highlighting discrepancies that could indicate missing energy. This missing energy, known as MET, could be due to unseen particles (like neutrinos or potential dark matter candidates) or issues within the detector itself. The following shows the necessary equation to calculate the missing energy.

$$\vec{E}_T^{\text{miss}} = -\sum_{\text{observable}} \vec{p}_{T,i} \tag{1}$$

$$S = \frac{E_T^{\text{miss}}}{\sqrt{\sum E_T}} \tag{2}$$

METSig helps quantify how significant this missing energy is compared to what is expected, assisting in identifying potential anomalies or calibrating the detector's response to such events.

7 Model Implementation

When implementing the code, several key functions help drive the training process of the model. But before diving into the details of these functions, it's important to first explain the structure of the data being used. The data is extracted from the JETMET of a CMS run and focuses on specific lumisections. Using DIALS, the data is gathered from histograms and stored in matrix form for each lumisection.

The autoencoder then processes these matrices as input, learning to compress and reconstruct the data to capture its most essential features. The model is trained with specific parameters, like batch size and encoding dimensions, which are tuned to optimize performance. The next sections will explain how each function contributes to data extraction, preparation, and model training, with the ultimate goal of improving the model's ability to capture and reproduce key patterns from the CMS run data.

7.1 Installations and Imports

```
# Installations
pip install cmsdials==1.3.0
pip install optuna
pip install numpy
pip install tensorflow
pip install keras
pip install matplotlib
pip install pandas
pip install pickle-mixin

# Python Imports
import os
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import plot_model
from tensorflow.keras import layers, models, optimizers, callbacks, regularizers
from tensorflow.keras.models import Sequential, load_model
from keras.models import Model
from tensorflow.keras.layers import Dense
import cmsdials
from cmsdials.auth.client import AuthClient
from cmsdials import Dials
from cmsdials.auth.bearer import Credentials
from cmsdials.filters import LumisectionHistogram1DFilters
import pandas as pd
import pickle
import optuna
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
```

As mentioned previously, libraries and open-source tools such as TensorFlow/Keras, Optuna, and Matplotlib are crucial for developing the model. CMS DIALS is also a key component, as it allows for connecting through the API and authorizing the connection to the personal computer. Additionally, several other imports and libraries are used to prepare and set up the environment for data extraction, analysis, and model training and testing.

7.2 Harvesting DIALS

```
def harvest_DIALS(run, ME='JetMET/MET/pfMETT1/Cleaned/METSig'):
    # Authenticate and initialize Dials client
    credentials = Credentials.from_creds_file()  # Assuming you have a credentials file
    dials = Dials(credentials, workspace="jetmet")
```

```
    # Fetch data using the specified filters
    data_ = dials.h1d.list_all(LumisectionHistogram1DFilters(me=str(ME), run_number=run))
    # Process the results
    results_ = [{'ls_number': result.ls_number, 'data': result.data, 'dataset': result.dataset}
                for result in data_.results]
    # Filter the results to only include those with 'PromptReco' in the dataset
    filtered_results = [result for result in results_ if 'PromptReco' in result['dataset']]
    # Sort the filtered results by lumisection number
    sorted_ = sorted(filtered_results, key=lambda x: x['ls_number'])
    # Extract lumisection numbers from the sorted results
    ls_numbers = [histogram['ls_number'] for histogram in sorted_]
    # Create a 2D array by vertically stacking the 'data' arrays from the sorted results
    data_array = np.vstack([histogram['data'] for histogram in sorted_])
    # Create the result as a tuple containing the data array,
    #lumisection numbers, and sorted results
    result = (data_array, ls_numbers, sorted_)
    # Return the result
    return result
```

The following function is used within the training model, to call into the DIALS API and extract the necessary data the model will train or test on. It first accesses the workspace "jetmet" using the credentials, and the parameters set from METSig are used to easily retrieve the necessary run data. After the data is harvested, it is filtered into the relevant data that will be used and saved into the variable `results_`. Important data to use includes the lumisection number, the data calculated for MET significance, and the dataset. After the results are filtered to only include the prompt Reco in the set.

```
def preprocess(output):
    min_val = tf.reduce_min(output, axis=0)
    max_val = tf.reduce_max(output, axis=0)
    normalized_inputs = tf.where(max_val - min_val != 0,
                                 (output - min_val) / (max_val - min_val + 1e-8),
                                 tf.zeros_like(output))
    return normalized_inputs

def calculate_metric(pred, true, anomaly_indices):
    mse = tf.math.reduce_mean(tf.math.pow(pred - true, 2), axis=1)
    mse_no_anom = mse[~anomaly_indices]
    mse_anom = mse[anomaly_indices]
    mean_mse_no_anom = tf.reduce_mean(mse_no_anom)
    std_mse_no_anom = tf.math.reduce_std(mse_no_anom)
    mean_mse_anom = tf.reduce_mean(mse_anom)
    metric = np.abs(mean_mse_anom - mean_mse_no_anom) / std_mse_no_anom
    return metric.numpy()

def objective(trial):
    # Suggest values for hyperparameters using Optuna
    batch_size = trial.suggest_categorical('batch_size', [32, 46, 64, 128])
    encoding_dim = trial.suggest_int('encoding_dim', 10, normalized_inputs_good.shape[1])
    encoding_dim_2 = trial.suggest_int('encoding_dim_2', 2, encoding_dim)
    # Fixed learning rate
    learning_rate = 1e-4
    # Train the autoencoder model with the suggested hyperparameters
    model, history = train_autoencoder_sequential(normalized_inputs_good,
                                                  encoding_dim, encoding_dim_2,
                                                  learning_rate, batch_size,
                                                  epochs=2000)
    # Train the model on the good data
    model.fit(normalized_inputs_good, normalized_inputs_good,
              epochs=2000, batch_size=batch_size, verbose=0)
    # Print the shape of the normalized input data
```

```
    print(normalized_inputs_good.shape)
    # Predict using the trained model on the anomaly data
    pred = model.predict(normalized_inputs_anomaly)
    # Create a boolean array to mark the anomalous LS (Luminosity Sections)
    anomaly_indices = np.zeros(normalized_inputs_anomaly.shape[0], dtype=bool)
    anomaly_indices[anomaly_ls_index] = True  # Marking the known anomalous indices
    # Calculate the metric based on the predictions and true anomaly data
    metric = calculate_metric(pred, normalized_inputs_anomaly, anomaly_indices)
    # Print the type of the metric
    print(metric.dtype)
    # Return the calculated metric (to be maximized by Optuna)
    return metric
```

The following functions are designed to assist in preprocessing, optimizing, and training the model. The preprocess function is used to normalize the data. It does this by scanning through the data and finding the minimum and maximum values for each feature. The data is then scaled to fit within the range of 0 to 1, which aligns with the requirements set in the model's layers. This normalization process is crucial for ensuring successful training results.

The objective function is called and used during the optimization phase of the model. In this function, hyperparameters are given suggested values, and the model is trained and fitted with these values. The outcome returned by the metric determines the best hyperparameters for the model. The objective function takes in two types of data: GOOD and BAD. This ensures that when an anomalous run is encountered, the function can choose the hyperparameters best suited to detect future anomalies.

Within the objective function, the model first trains on a GOOD run (normal data) and then predicts and evaluates a BAD run (anomalous data). The objective function then calls the `calculate_metric` function. This function calculates the Mean Squared Error (MSE) for both the normal data and the anomalous data, as well as the standard deviation of the MSE for the normal data. The `calculate_metric` function then returns a value stored in a variable called metric.

This is where Optuna comes into play. Optuna uses this metric value to optimize the hyperparameters, aiming to find the best configuration that maximizes the model's ability to detect anomalies.

```
# Create and run the Optuna study
study = optuna.create_study(direction='maximize')  # maximizing the metric
study.optimize(objective, n_trials=15)

# Print the best parameters
print("Best hyperparameters: ", study.best_params)
```

The following code is used to normalize the data and find the best hyperparameters for the model. The code calls the objective function and runs it for a specified number of trials—in this example, 15 trials. This means the model will be trained and evaluated 15 times, each with different hyperparameters suggested by Optuna. The objective function runs a different trial with varied parameters each time. At the end of the optimization process, the best hyperparameters, which result in the lowest loss, will be printed.

Figure 12 shows the run of Optuna finding the best parameters for 14 trials. The trials run for 2000 batches that are hard coded, but stop training the models on the parameters per trial based on the early stopping.

```
batch_size = best_params['batch_size']
encoding_dim = best_params['encoding_dim']
encoding_dim_2 = best_params['encoding_dim_2']
learning_rate = 1e-4

print(best_params)
print(f"BEST PARAMS :\nbatch size: {batch_size}\nencoding dimension:
↪  {encoding_dim}\nencoding dimension 2: {encoding_dim_2}\n")

# Build the autoencoder model
input_data = keras.Input(shape=(normalized_inputs_good.shape[1],))
```

```
8/8 [==============================] - 0s 8ms/step - loss: 0.0117 - val_loss: 0.0352
Epoch 733/2000
8/8 [==============================] - 0s 8ms/step - loss: 0.0117 - val_loss: 0.0352
Epoch 734/2000
8/8 [==============================] - 0s 7ms/step - loss: 0.0117 - val_loss: 0.0352
Epoch 735/2000
8/8 [==============================] - 0s 7ms/step - loss: 0.0117 - val_loss: 0.0352
Epoch 736/2000
8/8 [==============================] - 0s 8ms/step - loss: 0.0117 - val_loss: 0.0352
Epoch 737/2000
8/8 [==============================] - 0s 7ms/step - loss: 0.0117 - val_loss: 0.0352
Epoch 738/2000
8/8 [==============================] - 0s 9ms/step - loss: 0.0117 - val_loss: 0.0352
(1278, 51)

[I 2024-08-30 15:02:09,330] Trial 14 finished with value: 0.3165105975832287 and parameters: {'batch_size': 128, 'encoding_dim': 17, 'encoding_dim_2': 2}. Best is trial 13 with value: 0.3222638746799335
6.

float64
Best hyperparameters:  {'batch_size': 128, 'encoding_dim': 15, 'encoding_dim_2': 2}
```

Figure 12: Results of 15 trials from the Optuna optimization process, as described in the code above. The training stops at epoch 738, indicating that this was the optimal number of epochs to achieve the best results. The shape of the output is 51 for the y-axis, with all trials having the same shape for index [1]. At the end of each trial, the hyperparameters used and the resulting MSE (Mean Squared Error) are printed. A lower MSE indicates better performance. Finally, the code prints the best hyperparameters identified from the 15 trials for training the model.

```
encoded = layers.Dense(encoding_dim, activation='relu',
↪  activity_regularizer=tf.keras.regularizers.l2(learning_rate))(input_data)
decoded = layers.Dense(encoding_dim_2, activation='sigmoid')(encoded)
decoded = layers.Dense(encoding_dim, activation='sigmoid')(decoded)
decoded = layers.Dense(normalized_inputs_good.shape[1], activation='sigmoid')(decoded)

model = keras.Model(input_data, decoded)
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate), loss='mse')
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,
↪  restore_best_weights=True)

# Train the model with the correct variable
history = model.fit(normalized_inputs_good, normalized_inputs_good, epochs=2000,
↪  batch_size=batch_size, verbose=1, validation_split=0.2, callbacks=[early_stopping])

# Save the trained model
model.save('model_new.h5')
```

In previous code shown above; it builds the layers of the autoencoder model. Each parameter plays a significant role in training the autoencoder:

- **Normalized Inputs**: This refers to the input data that has been preprocessed and normalized. Normalization ensures that the data has the same scale, which helps the model train and evaluate more effectively.
- **Encoding Dimensions**: These refer to the number of neurons in each layer. The first encoding layer reduces the input data size, while the second layer, known as the bottleneck layer, further compresses the data by having fewer neurons than the first layer.
- **Learning Rate**: This parameter controls how much the model's weights are adjusted during training. It optimizes the model by determining the step size during each update to minimize the loss function.
- **Batch Size**: This is the number of samples the model processes before updating its weights. Smaller batch sizes allow for more frequent updates, while larger batch sizes offer more stable updates.
- **Epochs**: This parameter defines the number of times the model will go through the entire dataset during training. For example, if the number of epochs is set to 5, the model will train on the dataset five times.

These parameters work together to fine-tune the autoencoder, improving its performance in compressing and reconstructing the input data.

The API used in the code follows a straightforward approach to adding layers in a neural network model. It begins with the input layer, which defines the input shape to ensure it matches the initial data. The encoding layer follows, where the first parameter specifies the encoding dimensions, forming a fully connected neuron layer. ReLU activation is applied to capture nonlinear relationships, and regularization is used to prevent overfitting.

The next layer, known as the latent space, uses the second encoding dimension to further compress the data, creating a bottleneck. The sigmoid activation function is then applied, ensuring that output values are between 0 and 1. This is important when the output needs to represent a probability, such as in binary classification tasks, where it expresses the likelihood of an input belonging to a particular class. By constraining the output to this range, the results are easier to interpret and use in decision-making.

After the latent space, the decoding stage begins, where the model gradually expands back to the original encoding dimensions, reconstructing the data step by step. Finally, the output layer fully reconstructs the data to match the original input size. The model is compiled using the Adam optimizer, which is widely used for training deep learning models due to its adaptability with learning rates. The model's performance is measured using Mean Squared Error (MSE) as the loss function.

After the model is trained using model.fit, it adjusts its weights based on the input data to minimize the loss function, improving the model's reconstruction performance. During the training process, a callback called "Early Stopping" is often used. Early stopping halts the training when the model's performance stops improving, preventing it from running unnecessary epochs that could lead to overfitting. This means that instead of manually adjusting the epoch count for large datasets, you can set a high number of epochs and rely on early stopping to stop the training at the optimal point when the model no longer benefits from further epochs. This approach helps avoid overfitting by ensuring that training is halted at the right time.

The model is then saved in a file labeled `model_new.h5`. With the model created and saved, it can be reloaded and further trained if needed. More importantly, it can be used on new data without requiring retraining from scratch, while maintaining the same layout and using the best parameters found.

```
def train_autoencoder_sequential(normalized_inputs, encoding_dim, encoding_dim_2,
↪  learning_rate, batch_size, epochs=2000):
    # Define the Sequential model
    model = keras.Sequential()

    # Add layers to the model
    model.add(layers.InputLayer(input_shape=(normalized_inputs.shape[1],)))  # Define the
    ↪  input layer
    model.add(layers.Dense(encoding_dim, activation='relu',
    ↪  activity_regularizer=tf.keras.regularizers.l2(learning_rate)))  # Encoder layer
    model.add(layers.Dense(encoding_dim_2, activation='sigmoid'))  # Latent space layer
    model.add(layers.Dense(encoding_dim, activation='sigmoid'))  # Decoder layer
    model.add(layers.Dense(normalized_inputs.shape[1], activation='sigmoid'))  # Output
    ↪  layer

    # Compile the model with Adam optimizer and mean squared error loss
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate), loss='mse')
    model.summary()

    # Define early stopping to prevent overfitting
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,
    ↪  restore_best_weights=True)

    # Train the autoencoder model
    history = model.fit(normalized_inputs, normalized_inputs, epochs=epochs,
    ↪  batch_size=batch_size, verbose=1, validation_split=0.2, callbacks=[early_stopping])
    model.save('model.h5')

    # Return the trained model
    return model, history
```

Now that the model is saved, two new functions come into play. The first function is used to train the autoencoder, and it is called within a larger function known as `good_training`. Within these functions, the model is loaded with its layers and trained on the RUNs that are inputted. As mentioned earlier, the model is now trained exclusively on GOOD data labeled from the CMS Run Registry.

The `good_training` function can train either one run or multiple runs, depending on the user input. If only one run is trained, the model will go through the process as usual. If multiple runs are inputted into the model, the data from these runs will be extracted, concatenated to match the same shape, and then trained as normal.

Finally, the Mean Squared Error (MSE) is calculated by comparing the MSE of all normalized inputs and their reconstructions. The resulting data is then saved in a variable labeled `mse_data`.

Plotting the `mse_data` allows us to visualize the initial shape of the run's data, how it evolves, and ultimately provides a graph where each lumisection's loss is displayed across the entire run.If a spike is shown in a anomalous run, the user can input the lumisection to look at closer and see the reconstruction error.

To run and reconstruct the model(rebuild it the same as an input) a function was created to evaluate, called `evaluate_autoencoder_and_plot`. The function allows the user to enter the run number they would want to check after training on good run, and see the reconstruction to detect anomalies.

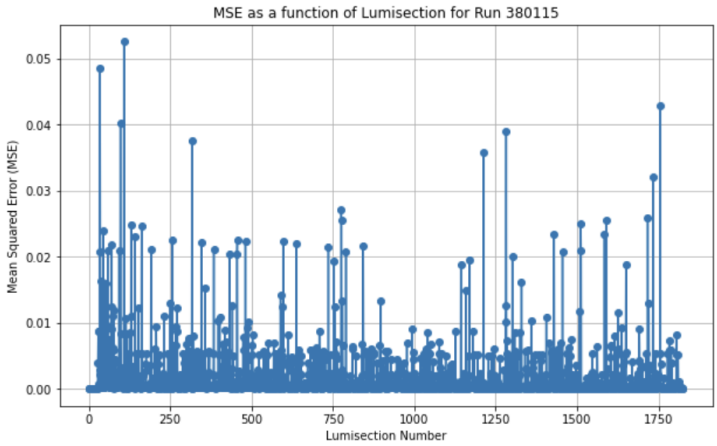
```
def evaluate_autoencoder_and_plot():
    # Step 1: Input the run number
    run = input("Enter Run Number:")

    # Step 2: Load the trained autoencoder model
    autoencoder_final_model = load_model('model_new.h5')

    # Step 3: Fetch test data using the harvest_DIALS function
    test_data_array, ls_numbers, sorted_results = harvest_DIALS(run,
    ↪  'JetMET/MET/pfMETT1/Cleaned/METSig')

    # Step 4: Normalize the input data
    normalized_input = preprocess(test_data_array)

    # Step 5: Generate predictions from the autoencoder
    auto_output = autoencoder_final_model.predict(normalized_input)

    # Step 6: Calculate Mean Squared Error (MSE) for each lumisection
    mse_data = tf.math.reduce_mean(tf.math.pow(auto_output - normalized_input, 2), axis=1)

    # Step 7: Plot MSE as a function of Lumisection number
    plt.figure(figsize=(10, 6))
    plt.plot(ls_numbers, mse_data, marker='o')
    plt.xlabel('Lumisection Number')
    plt.ylabel('Mean Squared Error (MSE)')
    plt.title(f'MSE as a function of Lumisection for Run {run}')
    plt.grid(True)
    plt.show()

    # Step 8: Ask the user for the specific Lumisection number to plot, with an option to
    ↪  stop
    while True:
        try:
            ls = input("Enter Lumisection Number to plot (or type 'X' to exit): ")
            if ls.upper() == 'X':
                break

            ls = int(ls)
            if ls in ls_numbers:
                ls_index = ls_numbers.index(ls)
```

```
                plt.figure(figsize=(10, 6))
                plt.plot(normalized_input[ls_index], label='Input', color='blue')
                plt.plot(auto_output[ls_index], label='Output', color='red',
                ↪  linestyle='dashed')
                plt.xlabel('Bin')
                plt.ylabel('Value')
                plt.title(f'Autoencoder Output vs. Input for Lumisection {ls}')
                plt.legend()
                plt.grid(True)
                plt.show()
            else:
                print(f"Lumisection {ls} not found in the data. Please enter a valid
                ↪  lumisection number.")
        except ValueError:
            print("Invalid input. Please enter a valid number or 'X' to exit.")

    # Step 9: After user exits, plot the average initial input vs. the average reconstructed
    ↪  output
    avg_input = np.mean(normalized_input, axis=0)
    avg_output = np.mean(auto_output, axis=0)

    plt.figure(figsize=(10, 6))
    plt.plot(avg_input, label='Average Input', color='blue')
    plt.plot(avg_output, label='Average Output', color='red', linestyle='dashed')
    plt.xlabel('Bin')
    plt.ylabel('Value')
    plt.title('Comparison of Average Initial Input vs. Reconstructed Output')
    plt.legend()
    plt.grid(True)
    plt.show()

# Run the function
evaluate_autoencoder_and_plot()
```

The following figure is an example run in detecting anomalies in a GOOD run.

Following in figure 15, the model was then tested on runs that are flagged as BAD by experts from the CMS run registry. The model will attempt to reconstruct the distribution, and if an anomaly is found it will show a spike in the lumisection graph.

```
Enter Run Number:380115
```




```
Enter Lumisection Number to plot (or type 'X' to exit): 177
```

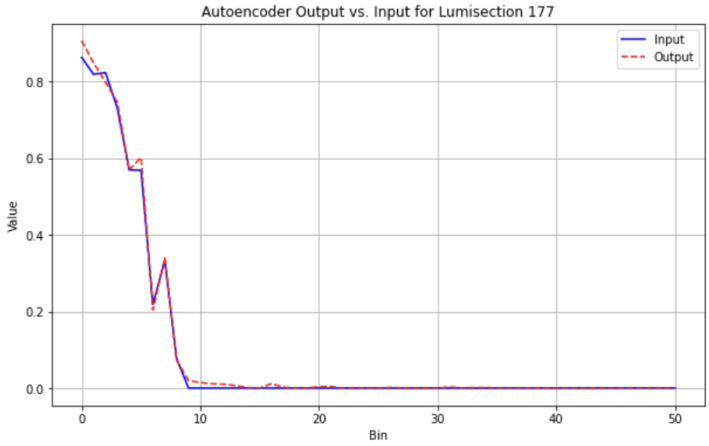
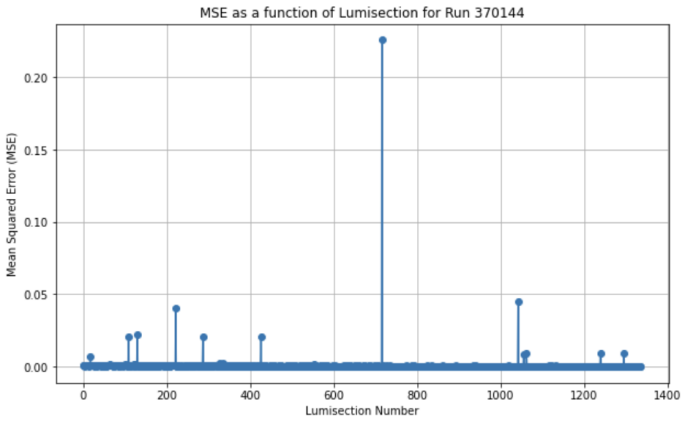


Figure 13: Results of running RUN 390115 through the model and evaluating results. The model shows the lumisections losses and then allows the user to view the reconstruction on whichever lumisection they enter. The second graph shows lumisection 177.

```
In [20]: 1 evaluate_autoencoder_and_plot()
Enter Run Number:370144
```




```
Enter Lumisection Number to plot (or type 'X' to exit): 1000
```

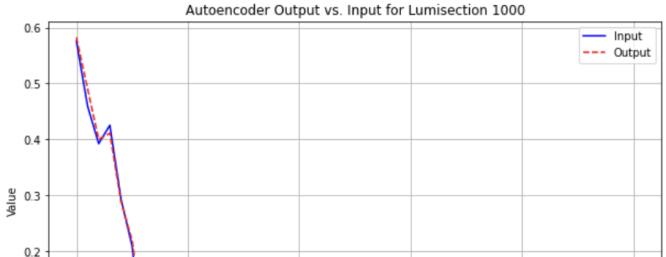


Figure 15 clearly shows the spike on 716, to ensure there is a difference multiple lumisections were checked to see where the anomaly is.

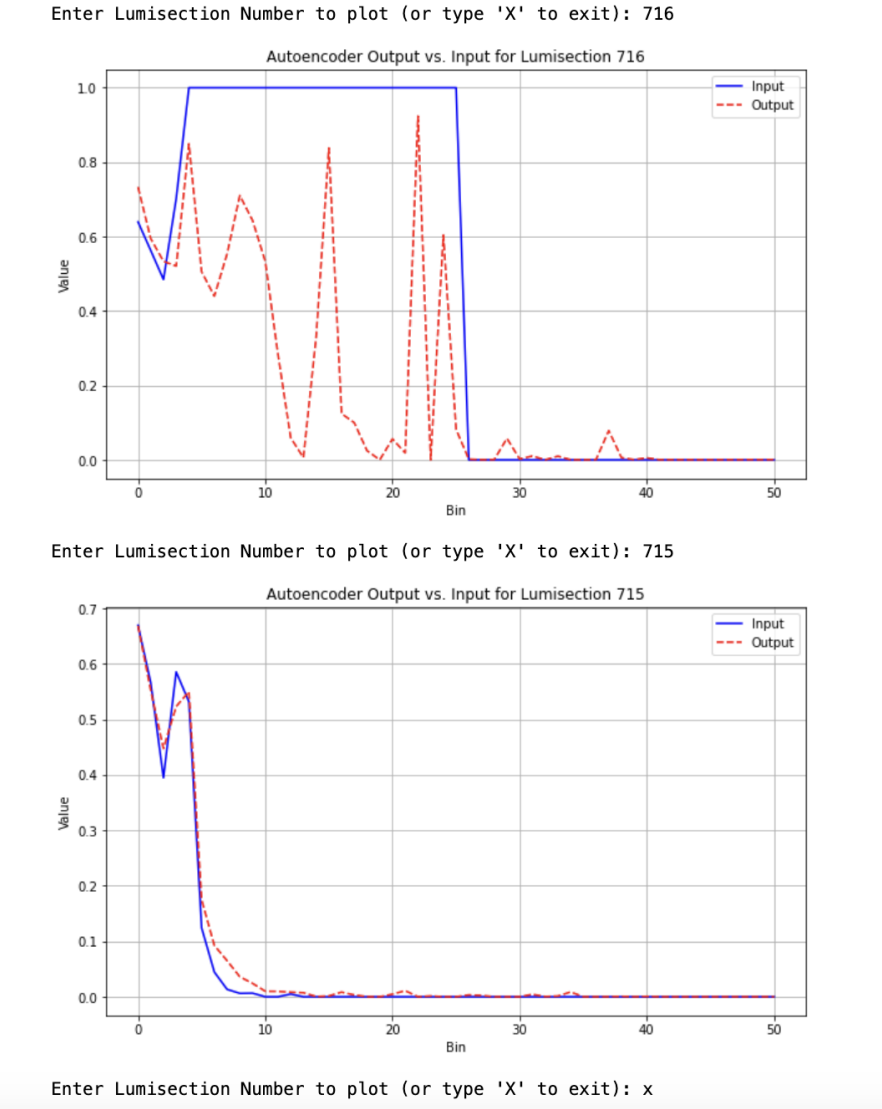


Figure 16: The figure shows a reconstruction plot of lumisection 716 and 715, a clear difference as an anomaly is found in 716. The error of reconstruction being high and obvious.

To ensure that the anomaly is in lumisection 716, the model was cross checked with the run registry, as it is already labeled where the said anomaly is in the lumisections. The following figure , shows a screenshot of the run registry showing that RUN 370144 has an anomaly in lumisection 716.

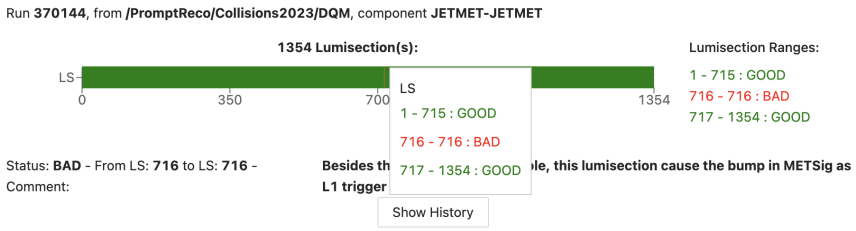


Figure 17: Screenshot from CMS run registry for RUN 370144 showing the anomalous lumisection is 716

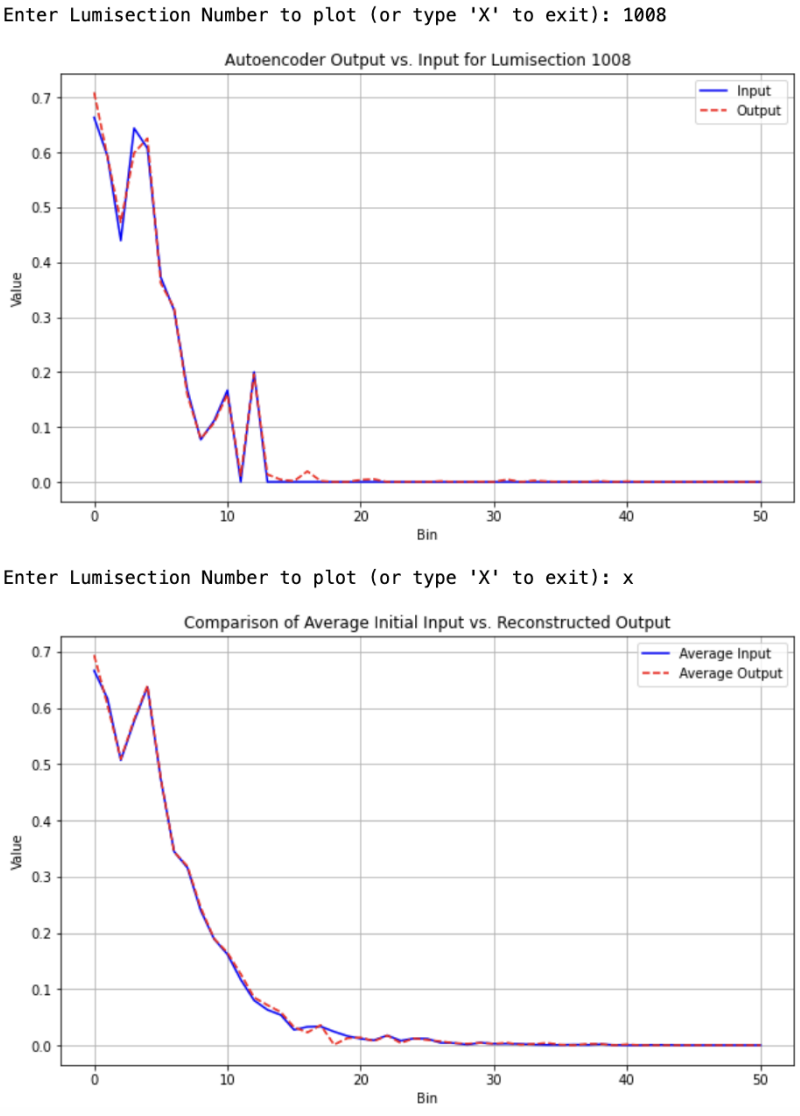


Figure 14: The results are the second part for RUN 380115, which is marked as a GOOD run in the CMS run registry. It shows another lumisection being reconstructed similarly to the initial input, followed by the full reconstruction of the observable distributions, with the Mean Squared Error (MSE) used as a metric to evaluate the accuracy of this reconstruction.

8 Conclusion and Future Works

8.1 Conclusion and Results

To conclude this project and report, a model for anomaly detection was successfully built, trained, and tested. The project began with thorough research within the CMS framework to fully understand the data being used. Following this, the role of machine learning models in anomaly detection was explored, leading to the selection of an Autoencoder as the most suitable approach.

Using Python and libraries such as TensorFlow's Keras, a simple sequential model was developed. The model's hyperparameters were optimized using Optuna, which helped identify the best configuration for the Autoencoder. The model was trained on multiple datasets, ensuring that the reconstruction error was minimized.

The final model effectively detects anomalies by training on "GOOD" data and then reconstructing the data. Anomalies are detected when a spike appears in the plot of Mean Squared Error (MSE) per lumisection, as demonstrated in previous sections. This approach enables the model to accurately identify deviations in the data, making it a valuable tool for anomaly detection in CMS data quality monitoring.

8.2 Future Work

Anomaly detection is crucial for ensuring that anomalous data is removed before it is sent for further analysis. Currently, the code is being modified and further developed to automatically remove any anomalous lumisections, enabling the data to be cleared and prepared for physics analysis.

The primary goal of integrating anomaly detection machine learning models within the CMS DQM is to improve the reliability of the monitoring process by supporting shifters with these ML tools. Once these tools are in place, they can then be applied to study multiple monitored elements simultaneously, enhancing the overall efficiency and accuracy of the monitoring system. Currently, the model is trained to detect anomalies within the MET Significance of runs. In the future, the aim is to expand the model's capabilities to analyze multiple dependencies from runs, identifying the source of anomalies more effectively.

For example, the model could take input data from MET Significance, MET Phi, and jets, allowing it to study the entire run comprehensively. If an anomaly is detected, it would be identified within a single model rather than requiring multiple machine learning models, each designated to a specific monitored element. This integrated approach would streamline the process and improve the accuracy of anomaly detection by leveraging correlations between multiple observables across various aspects of the CMS data.

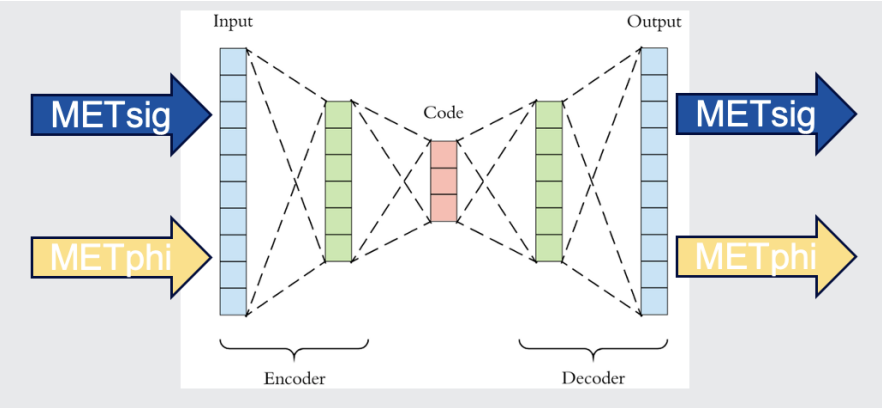


Figure 18: The figure illustrates an autoencoder that takes in two dependencies, METPhi and METSig, and reconstructs them to detect anomalies. This approach enables anomaly detection within a single model, eliminating the need for separate models for each dependency.

9 Acknowledgments

I would first like to thank my university, the American University of the Middle East, for giving me the opportunity to apply for and attend the CERN Summer Student Programme. With no prior background in physics but a strong admiration for learning new things and developing myself, I was extremely nervous. However, I'm so glad I stepped out of my comfort zone and explored a whole new world of science that I never imagined. I'd like to express my gratitude to Dr. Bilal Jabakanji and Dr. Bilel Neji for their support and for selecting me as one of the AUM applicants this year. I also want to thank my fellow peer and AUM researcher, Ghanima Al Asfour, for helping me integrate into CERN life and the physics community, having been in my shoes two years ago. Without her guidance, none of this would have been possible.

A huge thank you goes to CERN for providing so many students with the opportunity to work hands-on and implement what they're studying in real-life environments. My biggest thanks go to the DQM team for having me with them, especially to my supervisor, Roberto Seidita, and co-supervisor, Antonio Vagnerini. Roberto was incredibly supportive and understanding when giving me the project. It was something I had never done before, but with his guidance, support, and help, I was able to build a machine learning model and understand the math and the behind-the-scenes aspects of the code I was writing. Special thanks to my colleague Gabriel Moreira, not only for letting me access the API of DIALS but also for taking time out of his workday to help me with errors in my model and advising me on improving my programming format.

Finally, I want to thank all the summer students I met, as well as those I didn't get a chance to meet. Sharing this experience with so many people from different cultures and backgrounds was the highlight of this journey. Not only was I able to work on and create a report on a topic I'm passionate about, but I also met people with similar interests, as well as others who introduced me to new studies and interests. So, thank you to the 2024 Summer Students as a whole!

10 Bibliography

[21] "Tracker DQM Machine Learning studies for data certification". In: (2021). URL: https://cds.cern.ch/record/2799472.

[Aba+24] D. Abadjiev et al. "Autoencoder-Based Anomaly Detection System for Online Data Quality Monitoring of the CMS Electromagnetic Calorimeter". In: *Computing and Software for Big Science* 8.1 (June 2024). ISSN: 2510-2044. DOI: 10.1007/s41781-024-00118-z. URL: http://dx.doi.org/10.1007/s41781-024-00118-z.

[Bor16] Laura Borrello. *Offline DQM Shifts*. June 2016. URL: https://twiki.cern.ch/twiki/bin/view/CMS/OfflineDQMShifts.

[CER96] CERN. LHC Experiments Committee. *Technical proposal for CMS computing*. Tech. rep. Geneva, Switzerland: EP, 1996.

[CMS06] CMS Collaboration. *Detector performance and software*. CERN, 2006.

[CMS23] CMS Collaboration. *An AutoEncoder-based Anomaly Detection Tool with a Per-LS Granularity*. Mar. 2023.

[Gér19] Aurélien Géron. *Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow*. Sebastopol, CA, USA: O'Reilly Media, Inc., 2019.

[Ten23] TensorFlow. *The Sequential model*. 2023. URL: https://www.tensorflow.org/guide/keras/sequential_model.

11 Appendix: Full Code Listing

```
#!/usr/bin/env python
# coding: utf-8

# In[1]:

get_ipython().system('pip install cmsdials==1.3.0')

# In[2]:

import cmsdials

# In[3]:

get_ipython().system('pip install optuna')

# In[4]:

import os
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.utils import plot_model
from keras.models import Model
from tensorflow.keras import layers, models, optimizers, callbacks, regularizers
from cmsdials.auth.client import AuthClient
from cmsdials import Dials
from cmsdials.auth.bearer import Credentials
from cmsdials.filters import LumisectionHistogram1DFilters
import pandas as pd
import pickle
import optuna
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.models import load_model
from sklearn.metrics import mean_squared_error

# In[5]:

creds = Credentials.from_creds_file()
dials = Dials(creds, workspace="jetmet")

# In[6]:

pip install numpy tensorflow keras matplotlib pandas pickle-mixin optuna

# In[7]:

import matplotlib.pyplot as plt

# In[8]:

def harvest_DIALS(run, ME = 'JetMET/MET/pfMETT1/Cleaned/METSig'):
     # Authenticate and initialize Dials client
    credentials = Credentials.from_creds_file()  # Assuming you have a credentials file
    dials = Dials(credentials, workspace="jetmet")

    # Fetch data using the specified filters
    data_ = dials.h1d.list_all(LumisectionHistogram1DFilters(me=str(ME), run_number=run))
```

```
59
60      # Process the results
61      results_ = [{'ls_number': result.ls_number, 'data': result.data, 'dataset':
        ↪  result.dataset} for result in data_.results]
62
63      # Filter the results to only include those with 'PromptReco' in the dataset
64      filtered_results = [result for result in results_ if 'PromptReco' in result['dataset']]
65
66      # Sort the filtered results by lumisection number
67      sorted_ = sorted(filtered_results, key=lambda x: x['ls_number'])
68
69      # Extract lumisection numbers from the sorted results
70      ls_numbers = [histogram['ls_number'] for histogram in sorted_]
71
72      # Create a 2D array by vertically stacking the 'data' arrays from the sorted results
73      data_array = np.vstack([histogram['data'] for histogram in sorted_])
74
75      # Create the result as a tuple containing the data array, lumisection numbers, and
        ↪  sorted results
76      result = (data_array, ls_numbers, sorted_)
77
78      # Return the result
79      return result
80
81  def format_dials_output(data_):
82      results_ = [{'ls_number': result.ls_number, 'data': result.data, 'dataset':
        ↪  result.dataset} for result in data_.results]
83
84      # Filter the results to only include those with 'PromptReco' in the dataset
85      filtered_results = [result for result in results_ if 'PromptReco' in result['dataset']]
86
87      # Sort the filtered results by lumisection number
88      sorted_ = sorted(filtered_results, key=lambda x: x['ls_number'])
89
90      # Extract lumisection numbers from the sorted results
91      ls_numbers = [histogram['ls_number'] for histogram in sorted_]
92
93      # Create a 2D array by vertically stacking the 'data' arrays from the sorted results
94      data_array = np.vstack([histogram['data'] for histogram in sorted_])
95
96      # Create the result as a tuple containing the data array, lumisection numbers, and
        ↪  sorted results
97      result = (data_array, ls_numbers, sorted_)
98
99      # Return the resul
100     return result
101
102 def preprocess(output):
103     min_val = tf.reduce_min(output, axis=0)
104     max_val = tf.reduce_max(output, axis=0)
105     normalized_inputs = tf.where(max_val - min_val != 0, (output - min_val) / (max_val -
        ↪  min_val + 1e-8), tf.zeros_like(output))
106     return normalized_inputs
107
108 def calculate_metric(pred, true, anomaly_indices):
109     mse = tf.math.reduce_mean(tf.math.pow(pred - true, 2), axis=1)
110     mse_no_anom = mse[~anomaly_indices]
111     mse_anom = mse[anomaly_indices]
112
113     mean_mse_no_anom = tf.reduce_mean(mse_no_anom)
114     std_mse_no_anom = tf.math.reduce_std(mse_no_anom)
```

```
    mean_mse_anom = tf.reduce_mean(mse_anom)

    metric = np.abs(mean_mse_anom - mean_mse_no_anom) / std_mse_no_anom
    return metric.numpy()

def objective(trial):
    # Suggest values for hyperparameters using Optuna
    batch_size = trial.suggest_categorical('batch_size', [32, 46, 64, 128])  # Batch size to try
    encoding_dim = trial.suggest_int('encoding_dim', 10, normalized_inputs_good.shape[1])  # First encoding layer size
    encoding_dim_2 = trial.suggest_int('encoding_dim_2', 2, encoding_dim)  # Second encoding layer size, must be <= encoding_dim

    # Fixed learning rate (if you change batch size, learning rate should ideally be adjusted, but it's fixed here)
    learning_rate = 1e-4

    # Train the autoencoder model with the suggested hyperparameters
    model, history= train_autoencoder_sequential(normalized_inputs_good, encoding_dim, encoding_dim_2, learning_rate, batch_size, epochs=2000)

    # Train the model on the good data, using itself as the target (autoencoder task)
    model.fit(normalized_inputs_good, normalized_inputs_good, epochs=2000, batch_size=batch_size, verbose=0)

    # Print the shape of the normalized input data for verification
    print(normalized_inputs_good.shape)

    # Predict using the trained model on the anomaly data
    pred = model.predict(normalized_inputs_anomaly)

    # Create a boolean array to mark the anomalous LS (Luminosity Sections)
    anomaly_indices = np.zeros(normalized_inputs_anomaly.shape[0], dtype=bool)
    anomaly_indices[anomaly_ls_index] = True  # Marking the known anomalous indices

    # Calculate the metric based on the predictions and true anomaly data
    metric = calculate_metric(pred, normalized_inputs_anomaly, anomaly_indices)

    # Print the type of the metric for debugging purposes
    print(metric.dtype)

    # Return the calculated metric (to be maximized by Optuna)
    return metric

def train_autoencoder_sequential(normalized_inputs, encoding_dim, encoding_dim_2, learning_rate, batch_size, epochs=2000):
    # Define the Sequential model
    model = keras.Sequential()

    # Add layers to the model
    model.add(layers.InputLayer(input_shape=(normalized_inputs.shape[1],)))  # Define the input layer
    model.add(layers.Dense(encoding_dim, activation='relu', activity_regularizer=tf.keras.regularizers.l2(learning_rate)))  # Encoder layer
    model.add(layers.Dense(encoding_dim_2, activation='sigmoid'))  # Latent space layer
    model.add(layers.Dense(encoding_dim, activation='sigmoid'))  # Decoder layer
    model.add(layers.Dense(normalized_inputs.shape[1], activation='sigmoid'))  # Output layer

    # Compile the model with Adam optimizer and mean squared error loss
```

```
    model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate), loss='mse')
    model.summary()

    # Define early stopping to prevent overfitting
    early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5,
    ↪  restore_best_weights=True)

    # Train the autoencoder model
    history = model.fit(normalized_inputs, normalized_inputs, epochs=epochs,
    ↪  batch_size=batch_size, verbose=1, validation_split=0.2, callbacks=[early_stopping])
    model.save('model.h5')
    # Return the trained model
    return model, history

def good_training():
    # Ask for run numbers input (comma-separated if multiple)
    input_runs = input("Enter run number(s) separated by commas: ")
    run_list = [item.strip() for item in input_runs.split(',')]

    # Print the resulting list
    print("Your run list is:", run_list)
    outputs = []
    histories = []
    sorted_results_all = []
    ls_numbers_all = []

    if len(run_list) > 1:
        print("Your runs will be concatenated")
        for run in run_list:
            output, ls_numbers, sorted_results = harvest_DIALS(run,
            ↪  ME='JetMET/MET/pfMETT1/Cleaned/METSig')
            outputs.append(output)
            sorted_results_all.extend(sorted_results)
            ls_numbers_all.extend(ls_numbers)
        print("Appended output list:", outputs)
        output_finals = np.concatenate(outputs, axis=0)
        print("Concatenated array:", output_finals)
    else:
        run = run_list[0]
        output, ls_numbers, sorted_results = harvest_DIALS(run,
        ↪  ME='JetMET/MET/pfMETT1/Cleaned/METSig')
        outputs.append(output)
        sorted_results_all.extend(sorted_results)
        ls_numbers_all.extend(ls_numbers)
        print("Single output:", outputs)
        output_finals = np.array(outputs[0])

    normalized_inputs = preprocess(output_finals)
    autoencoder_final_model = load_model('model_new.h5')
    history = autoencoder_final_model.fit(normalized_inputs, normalized_inputs, epochs=2000,
    ↪  batch_size=batch_size, verbose=1, validation_split=0.2, callbacks=[early_stopping])
    autoencoder_final_model.save('model_new.h5')


    # Plot the training history
    plt.figure()
    plt.plot(history.history['loss'], label='loss')
    plt.plot(history.history['val_loss'], label='val_loss')
    plt.ylabel('Loss')
    plt.xlabel('Epoch')
    plt.legend(loc='upper right')
```

```
    plt.show()

    # Evaluate the model
    mse_data = tf.math.reduce_mean(tf.math.pow(model.predict(normalized_inputs)-normalized_inputs, 2), axis=1)

    # Print the best parameters used
    print("The best parameters for the model that were used are:")
    print("Encoding Dimension:", best_params['encoding_dim'])
    print("Encoding Dimension 2:", best_params['encoding_dim_2'])
    print("The batch size used:", best_params['batch_size'])
    return model, sorted_results_all, normalized_inputs, ls_numbers_all

# In[17]:

learning_rate = 1e-4
best_params = {'batch_size': 46, 'encoding_dim': 115, 'encoding_dim_2': 70}

# In[10]:

output,ls_numbers,sorted_data=harvest_DIALS(380310) ##good run
normalized_inputs_good = preprocess(output)
print ("good inputs:",normalized_inputs_good)

output2,ls_numbers,sorted_data=harvest_DIALS(370144) ##bad run
normalized_inputs_anomaly = preprocess(output2)
anomaly_ls_index = 716
print ("anomoloy inputs:",normalized_inputs_anomaly)

# In[11]:

# Create and run the Optuna study
study = optuna.create_study(direction='maximize')  # maximizing the metric
study.optimize(objective, n_trials=15)

# Print the best parameters
print("Best hyperparameters: ", study.best_params)
#367696

# In[14]:

batch_size = best_params['batch_size']
encoding_dim = best_params['encoding_dim']
encoding_dim_2 = best_params['encoding_dim_2']
learning_rate = 1e-4

print(best_params)
print(f"BEST PARAMS :\nbatch size: {batch_size}\nencoding dimension: {encoding_dim}\nencoding dimension 2: {encoding_dim_2}\n")

# Build the autoencoder model
input_data = keras.Input(shape=(normalized_inputs_good.shape[1],))
encoded = layers.Dense(encoding_dim, activation='relu', activity_regularizer=tf.keras.regularizers.l2(learning_rate))(input_data)
decoded = layers.Dense(encoding_dim_2, activation='sigmoid')(encoded)
decoded = layers.Dense(encoding_dim, activation='sigmoid')(decoded)
decoded = layers.Dense(normalized_inputs_good.shape[1], activation='sigmoid')(decoded)

model = keras.Model(input_data, decoded)
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate), loss='mse')
```

```
early_stopping = tf.keras.callbacks.EarlyStopping(monitor='val_loss', patience=5, restore_best_weights=True)

# Train the model with the correct variable
history = model.fit(normalized_inputs_good, normalized_inputs_good, epochs=2000, batch_size=batch_size, verbose=1, validation_split=0.2, callbacks=[early_stopping])

# Save the trained model
model.save('model_new.h5')

# In[29]:

good_training() #380126,380127,380128

# In[18]:

good_training()#380349,380348,380346,380309

# In[38]:

good_training() #380308,380307,380306,380238,380236,380235,380197

# In[39]:

good_training() #380074,380066

# In[42]:

good_training()#380043,380056,380051,380052,380053

# In[17]:

good_training()#380033,380030,380029,380066

# In[25]:
good_training() #379866,379774,379765

# In[15]:

good_training() #379729,379661,379660

# In[ ]:

good_training() #370169,370172,370175,370129,370102

# In[13]:

import numpy as np
from keras.models import load_model
import matplotlib.pyplot as plt
from sklearn.metrics import mean_squared_error

run = input("Enter Run Number:")
# Load your trained model
autoencoder_final_model = load_model('model_new.h5')

# Fetch test data using the harvest_DIALS function
test_data_array, ls_numbers, sorted_results = harvest_DIALS(run, 'JetMET/MET/pfMETT1/Cleaned/METSig')
normalized_input = preprocess(test_data_array)
auto_output = autoencoder_final_model.predict(normalized_input)
```

```
# print(normalized_input)
# mse = tf.keras.losses.MSE(normalized_input, auto_output)
# print('MSE on test set:', mse.numpy())

mean_squared_error(normalized_input, auto_output)

# In[21]:

mse_data = tf.math.reduce_mean(tf.math.pow(model.predict(normalized_input)-normalized_input, 2), axis=1)

# In[22]:

mse_data

# In[23]:

plt.plot(mse_data)

# In[27]:

ls = 75
plt.plot(auto_output[ls])
plt.plot(normalized_input[ls])
plt.xlabel('Bin')
plt.ylabel('Value')
plt.title(f'Autoencoder Output for Lumisection {ls}')
plt.show()

# In[28]:

run = input("Enter Run Number:")
# Load your trained model
autoencoder_final_model = load_model('model_new.h5')

# Fetch test data using the harvest_DIALS function
test_data_array, ls_numbers, sorted_results = harvest_DIALS(run, 'JetMET/MET/pfMETT1/Cleaned/METSig')
normalized_input = preprocess(test_data_array)
auto_output = autoencoder_final_model.predict(normalized_input)
# print(normalized_input)
# mse = tf.keras.losses.MSE(normalized_input, auto_output)
# print('MSE on test set:', mse.numpy())

mean_squared_error(normalized_input, auto_output)
mse_data = tf.math.reduce_mean(tf.math.pow(model.predict(normalized_input)-normalized_input, 2), axis=1)
print(mse_data)
ls = 715
plt.plot(auto_output[ls])
plt.plot(normalized_input[ls])
plt.xlabel('Bin')
plt.ylabel('Value')
plt.title(f'Autoencoder Output for Lumisection {ls}')
plt.show()

# Plot MSE as a function of Lumisection number
plt.figure(figsize=(10, 6))
plt.plot(ls_numbers, mse_data, marker='o')
plt.xlabel('Lumisection Number')
plt.ylabel('Mean Squared Error (MSE)')
```

```
plt.title(f'MSE as a function of Lumisection for Run {run}')
plt.grid(True)
plt.show()

# In[29]:

run = input("Enter Run Number:")
autoencoder_final_model = load_model('model_new.h5')
test_data_array, ls_numbers, sorted_results = harvest_DIALS(run,
↪  'JetMET/MET/pfMETT1/Cleaned/METSig')
normalized_input = preprocess(test_data_array)

# Predict the autoencoder's output
auto_output = autoencoder_final_model.predict(normalized_input)

# Compute the overall Mean Squared Error (MSE)
overall_mse = mean_squared_error(normalized_input, auto_output)
print(f"Overall Mean Squared Error: {overall_mse}")

# Optionally, compute MSE using TensorFlow for verification
overall_mse_tf = tf.reduce_mean(tf.square(normalized_input - auto_output)).numpy()
print(f"Overall Mean Squared Error (TensorFlow): {overall_mse_tf}")

# Plot the average initial input vs. the average reconstructed output across all samples
avg_input = np.mean(normalized_input, axis=0)
avg_output = np.mean(auto_output, axis=0)

plt.figure(figsize=(10, 6))
plt.plot(avg_input, label='Average Input', color='blue')
plt.plot(avg_output, label='Average Output', color='red', linestyle='dashed')
plt.xlabel('Bin')
plt.ylabel('Value')
plt.title('Comparison of Average Initial Input vs. Reconstructed Output')
plt.legend()
plt.grid(True)
plt.show()

# Optionally, plot individual sample comparison
sample_index = 0  # Change this to plot a different sample
plt.figure(figsize=(10, 6))
plt.plot(normalized_input[sample_index], label='Input', color='blue')
plt.plot(auto_output[sample_index], label='Output', color='red', linestyle='dashed')
plt.xlabel('Bin')
plt.ylabel('Value')
plt.title(f'Comparison of Initial Input vs. Reconstructed Output for Sample {sample_index}')
plt.legend()
plt.grid(True)
plt.show()

# In[31]:

run = np.sum(normalized_input,axis=0)
plt.plot(run)

# In[19]:

def evaluate_autoencoder_and_plot():
    # Step 1: Input the run number
    run = input("Enter Run Number:")

    # Step 2: Load the trained autoencoder model
```

```
    autoencoder_final_model = load_model('model_new.h5')

    # Step 3: Fetch test data using the harvest_DIALS function
    test_data_array, ls_numbers, sorted_results = harvest_DIALS(run,
    ↪  'JetMET/MET/pfMETT1/Cleaned/METSig')

    # Step 4: Normalize the input data
    normalized_input = preprocess(test_data_array)

    # Step 5: Generate predictions from the autoencoder
    auto_output = autoencoder_final_model.predict(normalized_input)

    # Step 6: Calculate Mean Squared Error (MSE) for each lumisection
    mse_data = tf.math.reduce_mean(tf.math.pow(auto_output - normalized_input, 2), axis=1)

    # Step 7: Plot MSE as a function of Lumisection number
    plt.figure(figsize=(10, 6))
    plt.plot(ls_numbers, mse_data, marker='o')
    plt.xlabel('Lumisection Number')
    plt.ylabel('Mean Squared Error (MSE)')
    plt.title(f'MSE as a function of Lumisection for Run {run}')
    plt.grid(True)
    plt.show()

    # Step 8: Ask the user for the specific Lumisection number to plot, with an option to
    ↪  stop
    while True:
        try:
            ls = input("Enter Lumisection Number to plot (or type 'X' to exit): ")
            if ls.upper() == 'X':
                break

            ls = int(ls)
            if ls in ls_numbers:
                ls_index = ls_numbers.index(ls)
                plt.figure(figsize=(10, 6))
                plt.plot(normalized_input[ls_index], label='Input', color='blue')
                plt.plot(auto_output[ls_index], label='Output', color='red',
                ↪  linestyle='dashed')
                plt.xlabel('Bin')
                plt.ylabel('Value')
                plt.title(f'Autoencoder Output vs. Input for Lumisection {ls}')
                plt.legend()
                plt.grid(True)
                plt.show()
            else:
                print(f"Lumisection {ls} not found in the data. Please enter a valid
                ↪  lumisection number.")
        except ValueError:
            print("Invalid input. Please enter a valid number or 'X' to exit.")

    # Step 9: After user exits, plot the average initial input vs. the average reconstructed
    ↪  output
    avg_input = np.mean(normalized_input, axis=0)
    avg_output = np.mean(auto_output, axis=0)

    plt.figure(figsize=(10, 6))
    plt.plot(avg_input, label='Average Input', color='blue')
    plt.plot(avg_output, label='Average Output', color='red', linestyle='dashed')
    plt.xlabel('Bin')
    plt.ylabel('Value')
```

```
    plt.title('Comparison of Average Initial Input vs. Reconstructed Output')
    plt.legend()
    plt.grid(True)
    plt.show()

# Run the function
evaluate_autoencoder_and_plot()

# In[20]:

evaluate_autoencoder_and_plot()

```