Project report in DAT255 – Deep Learning Engineering

Anomaly Detection in LHC Olympics 2020 Dataset using Deep Learning

Date 24.04.2025

Carla Miquel Blasco, (189020)

Jolanta Rudus, (189942)

Ralf Stockreiter, (189955)

Problem description

The Large Hadron Collider (LHC) is the most powerful particle accelerator in the world, designed to study fundamental physics by colliding particles at nearly the speed of light. One of its main goals is to search for Beyond Standard Model (BSM) Physics, which could reveal new discoveries beyond our current understanding.

The LHC Olympics 2020 dataset is a simulated dataset that mimics real LHC collision events. Some events contain only standard physics interactions (background), while others may hide signals of new physics. The challenge is to detect these unknown signals using deep learning.

This project aims to develop a Deep Learning-based anomaly detection model to identify rare or unexpected signals that deviate from the learned background distribution. Specifically, Autoencoder-type models will be trained to reconstruct normal (background) event patterns and hopefully detect unusual events based on high reconstruction errors. This approach will help distinguish possible new physics signals from standard QCD background in the events_anomalydetection.h5 dataset [7].

Deep learning and autoencoder architectures are well-suited for this task because they can model complex, high-dimensional data distributions without explicit labelling. Existing solutions in High Energy Physics (HEP) include statistical tests, isolation-based models or supervised classifiers trained on specific BSM signatures. However, they either require prior knowledge of the signal or do not generalise well. Our approach aims to improve on these by using unsupervised learning to detect any deviation from the background, making it more flexible and potentially sensitive to a wider range of new physics scenarios.

The scope of the current project is limited to training and evaluating the model in a controlled, offline setting. However, the methodology could be adapted and integrated into real-time data monitoring systems at the LHC, assisting physicists in filtering interesting events for further analysis.

Data

The dataset used in this project consists of simulated LHC (Large Hadron Collider) collision events, with each event containing up to 700 particles. The properties recorded for each particle are:

- **pT (Transverse Momentum):** The momentum of the particle perpendicular to the beam axis.
- **η (Pseudorapidity)**: A measure of how far the particle moves in the detector.
- **φ (Azimuthal Angle)**: The direction in which the particle moves around the beam axis.
- **Label (if available)**: Indicates whether the event is part of the background (0) or an anomaly (1).

Each event is represented as a fixed-size array of shape (2100,), as it consists of 700 particles, each with 3 features (pT, η, φ), ensuring uniformity across all events, making it ideal for deep learning architectures.

Dataset Structure

The dataset is divided into two main categories:

- Background Sample (QCD Dijet Events): Represents normal LHC events.
- Black Box Data: May contain unknown new physics signals or anomalies.

These labelled categories provide a benchmark for unsupervised anomaly detection. The goal is to train models to learn the normal patterns (background) and identify deviations (anomalies) within new data.

Pros and Cons of the Dataset

The LHC Olympics 2020 dataset is well-suited for deep learning due to its large size, consistent structure, and informative physical features, which are ideal for training models capable of capturing spatial and temporal patterns. These characteristics allow for effective training of deep models, especially those like convolutional autoencoders (CNNs) or recurrent neural networks (RNNs).

However, there are challenges. One limitation is the scarcity of labelled anomalies, which makes it difficult to evaluate the performance of the model on rare events. Additionally, because the data is synthetic, it may not fully capture the noise and imperfections that would be present in real-world particle collision data, limiting its generalizability.

Model Comparison

In this project, we did not compare our models to pre-trained ones. All models, including Convolutional Autoencoders (CNN), Recurrent Neural Networks (RNNs), and Variational Autoencoders (VAEs), were trained from scratch using the background-only data. This

approach was chosen to focus on learning the normal patterns within the data without the bias of pretrained models.

Data Preprocessing and Reformatting

We first loaded the data from the file *events_anomalydetection.h5* [1].

To make the training process more stable and help the models learn better, we apply normalization to each feature (pT, η, φ) before training. This means we scale the values so they are in a similar range, which helps the model process them more effectively.

Initially, we tried to normalize all three features using MinMaxScaler, which scales values to the range [0, 1]. However, we noticed that pT (transverse momentum) had a much larger range than η and φ. Using MinMaxScaler on pT resulted in potentially important high-magnitude variations being compressed into a narrow range, possibly leading to loss of information. Based on this observation and feedback from our teacher, we switched to using StandardScaler exclusively for pT, which standardizes the values by removing the mean and scaling to unit variance. This change led to improved model performance.

For η and φ, we continued using MinMaxScaler, since their values already lie in reasonably bounded ranges and don't suffer from the same extreme scaling issues.

These scalers are fitted only on the real, non-padded particle data from the training set, so the scaling is based on meaningful input and not influenced by zeros added for padding. The fitted scalers are saved so we can use the exact same transformation when validating or testing the model later.

To prevent the model from learning from artificial input (i.e., the zero-padded values used to standardize event length), we implemented masking. Each event is accompanied by a binary mask, where valid particles are assigned a value of 1.0, and zero-padded particles are assigned 0.0. This ensures the model focuses only on real data during training and evaluation, improving learning efficiency and robustness.

For efficient data handling:

- The training set is loaded lazily via tf.data.Dataset generators, allowing streaming from disk and minimizing memory usage.
- The validation and test sets are fully loaded into memory and wrapped in TensorFlow datasets for rapid batch access.
- All datasets are batched and prefetched using tf.data.AUTOTUNE, optimizing data throughput and ensuring efficient model training.

Finally, we reshaped the data to fit the input requirements of the deep learning models:

A. **Convolutional Autoencoder (CNN)**: Each event was reshaped into a 2D image-like structure of shape `(N_events, 700, 3, 1)`. This structure allows the model to

capture spatial relationships, making it more efficient than using fully connected layers.

B. **Recurrent Neural Networks (RNNs, LSTMs)**: The data was reshaped into a sequence format of shape `(N_events, 700, 3)`, treating each particle as a sequential input. This approach captures both spatial and temporal dependencies, but it requires more computational resources for training.
C. **Variational Autoencoder (VAE)**: The data was reshaped into the same sequence format as the RNN model `(N_events, 700, 3)`. The VAE learns a probabilistic latent space and models sequential dependencies using RNN or LSTM layers in the encoder and decoder, with training based on both reconstruction loss and KL divergence.

Data Augmentation

No data augmentation techniques were applied to the dataset. Given the structured nature of the data and the meaningfulness of the features (pT, η, φ), applying artificial alterations could have introduced noise, potentially undermining the ability of the model to learn meaningful patterns.

Model implementation

We explored three different architectures: Variational Autoencoder (VAE), Recurrent Neural Network (RNN), and Convolutional Neural Network (CNN).

Variational Autoencoder (VAE)

A popular use of variational autoencoders is to generate new data similar to the training data, for example, images or text. Since they learn the distribution of data, they can also be used in the context of anomaly detection to identify outliers that do not fit. The autoencoder gets trained on normal (non-anomalous) data so that it is able to reconstruct any normal data. When it gets anomalous data as input, it is not able to properly reconstruct it. This reconstruction error can be used to identify anomalies. If the error is higher than a set threshold, the data is flagged as anomalous. If the autoencoder was able to make a good reconstruction, and thus the error is low, the data is flagged as non-anomalous. To define the threshold, we used the 95th percentile from the validation dataset. This means 95% of the events with lower error are considered ‘normal’, and those above the threshold are predicted as ‘anomaly’.

Variational autoencoders are optimal for structured, high-dimensional data of fixed length. They compress the high-dimensional data into a lower-dimensional latent space while preserving a meaningful structure. This compression forces the model to learn underlying patterns, perfect for our particle events, where structures like particle distributions and their three values matter. The features of our data have a physical meaning and relationships. VAEs are good at modelling such structured dependencies, especially when the

relationships are nonlinear. We are using a convolutional layers which expect fixed-length inputs. Our dataset already provides that (700 particles x 3 features), so there is no need for complex preprocessing like dynamic padding or attention masking. VAEs also do not just encode to a single point, they encode to a distribution (mean and variance). Encoding into a distribution instead of a fixed point allows the model to learn a range of possible representations, rather than just one. This adds an element of randomness to the encoding process. Thus, our model becomes robust to small variations (e.g. detector noise if there is any) and nearby points in latent space correspond to similar physical events, which is good for anomaly detection and interpretation [8].

Architecture

We were experimenting with different approaches for the encoder.

The first option is to use **one-dimensional convolutional layers**. Although Conv1D typically handles sequences, using a kernel size of 1 means that each particle's features are treated independently, without considering the order of particles. Even with kernel size 1, convolutional layers can learn important features from the particles' individual characteristics. Conv1D uses shared weights, which means that we need fewer parameters, can train faster and are less likely to run into the problem of overfitting. This is especially helpful when working with high-dimensional data like our 2100 features.

The second option is to use **TimeDistributed dense layers**, which apply the same dense (fully connected) transformation to each particle individually. This approach treats each particle independently, like in the Conv1D case, but without using convolutions. It gives the model flexibility to learn complex transformations for each particle without sharing weights across them. This can increase expressiveness but comes at the cost of more parameters and potentially higher risk of overfitting.

The third option is using **fully connected (dense) layers** after flattening the input. This means that the particle-wise structure gets discarded entirely and the whole event gets treated as a single vector of 2100 features. This means the model no longer understands that the input is made up of separate particles. Instead, it just sees one long vector of numbers. As a result, it loses the helpful assumption that each group of three numbers (pT, η, φ) belongs to an individual particle. However, it allows the model to learn global interactions between all features. This can be powerful but also increases the risk of overfitting.

We apply **batch normalization** for the training to converge faster and to prevent exploding/vanishing gradients. This normalizes the data for each mini-batch during training so that they have a mean of 0 and a standard deviation of 1.

Dropout is used to regularize the model and prevent overfitting. It randomly turns off a fraction of the neurons in a layer. This forces the model not to rely too heavily on any one neuron.

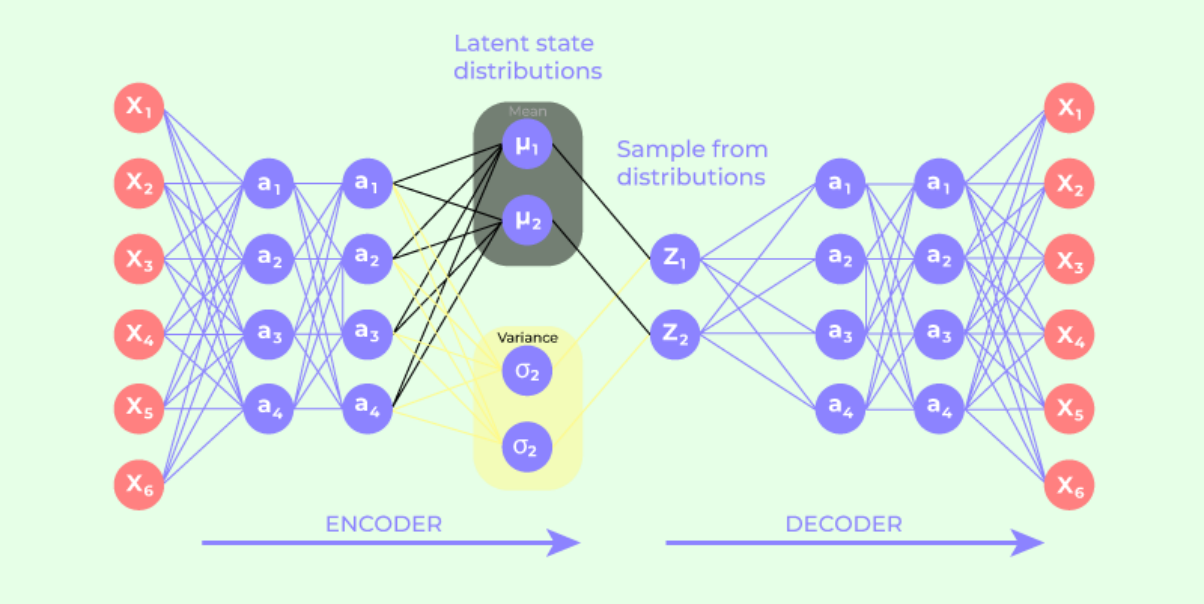
After the encoder has extracted meaningful features from the input, it maps the event into a **latent space**. As already mentioned, a variational autoencoder encodes each input as a distribution. It is defined by a mean vector (z_mean) and a log-variance vector (z_log_var).

To actually sample a point z from this distribution, we use the reparameterization trick (z = z_mean + z_log_var * random noise).

```
z_mean = Dense(self.latent_dim, name="z_mean")(x) # mean of the latent distribution
z_log_var = Dense(self.latent_dim, name="z_log_var")(x) # log variance
z = Lambda(sampling, output_shape=(self.latent_dim,), name="z")([z_mean, z_log_var])
```

The **decoder** takes the sampled latent vector z and attempts to reconstruct the original event from it. It is built using fully connected (dense) layers, with optional batch normalization and dropout for stability and regularization. It first expands the low-dimensional latent vector into a flat vector of size 2100 (700 particles × 3 features) using a dense layer. Then, this vector is reshaped into the original format of shape (700, 3).

We use a **sigmoid activation** in the last layer to ensure the output values are between 0 and 1. Since the input features are scaled to fall within the range [0, 1], using a sigmoid ensures that the decoder's outputs match this scale. This helps us to maintain consistency between the inputs and the reconstructed outputs, allowing the model to learn a good representation of the original data.



In this picture you can see a summary of the rough architecture for our variational autoencoder, combining the encoder part with the latent space and the decoder.

Recurrent Neural Network (RNN)

Recurrent neural networks can also be used in an encoder-decoder architecture. Therefore the basic process is similar to the variational autoencoder. The main difference is that a VAE is probabilistic, as it encodes inputs into a distribution. Basic RNNs on the other hand are deterministic, encoding to a single vector which is then used for reconstruction.

RNNs are well-suited for sequential or ordered data, where the order of elements carries meaning. In our case, even though the particles in an event aren't naturally sequential like words in a sentence, we can treat them as a sequence to explore whether the model can learn dependencies or patterns across particle positions. This approach could reveal useful structures or correlations. One possible scenario we could think of is early particles in an event influencing later ones.

Unlike convolutional or dense layers, RNNs (like LSTMs and GRUs) maintain a memory of previous inputs while processing the current one. This gives them the potential to capture longer ranging relationships across the particle list [9].

Architecture

We explored RNN-based autoencoders using both LSTM (Long Short-Term Memory) and GRU (Gated Recurrent Unit) layers. These types of layers are designed to deal with sequences and learn dependencies over time — or in our case, over the particle order.

Layer (type)	Output Shape	Param #	Connected to
input_layer (InputLayer)	(None, 700, 3)	0	-
not_equal (NotEqual)	(None, 700, 3)	0	input_layer[0][0]
masking (Masking)	(None, 700, 3)	0	input_layer[0][0]
any (Any)	(None, 700)	0	not_equal[0][0]
gru (GRU)	(None, 700, 128)	51,072	masking[0][0], any[0][0]
gru_1 (GRU)	(None, 700, 64)	37,248	gru[0][0], any[0][0]
gru_2 (GRU)	(None, 64)	24,960	gru_1[0][0], any[0][0]
dense (Dense)	(None, 2100)	136,500	gru_2[0][0]
reshape (Reshape)	(None, 700, 3)	0	dense[0][0]

Here is one sample architecture we tried. In the following we will talk about the most important parts of our autoencoders' architecture.

The **encoder** begins with a masking layer, which instructs the model to ignore padded timesteps — in our case, rows where all values are zero. This is important because in variable-length sequences, zero-padding is used to make all inputs the same shape, but these padded particles carry no useful information. The masking ensures that the model does not mistakenly learn patterns from this padding.

The sequence is then passed through one or more **LSTM or GRU layers**. These layers can learn to capture interactions between particles across the sequence. Intermediate layers use return_sequences=True to preserve the full sequence at each step, while the final layer reduces the sequence to a single latent vector (return_sequences=False), which summarizes the whole event into a compressed representation.

```
for units in encoder_layers:
    if rnn_type == 'LSTM':
        encoded = LSTM(units, activation=activation_encoder, return_sequences=True)(encoded)
    elif rnn_type == 'GRU':
        encoded = GRU(units, activation=activation_encoder, return_sequences=True)(encoded)

if rnn_type == 'LSTM':
    encoded = LSTM(latent_dim, activation=activation_encoder, return_sequences=False)(encoded)
elif rnn_type == 'GRU':
    encoded = GRU(latent_dim, activation=activation_encoder, return_sequences=False)(encoded)
```

This compressed vector has a dimensionality defined by latent_dim, which can be tuned. It acts as the core representation of the particle event, where we hope the model captures the most meaningful patterns in a lower-dimensional space.

The **decoder** takes this latent vector and tries to reconstruct the original particle event. It first maps the latent vector into a larger vector that has the same total size as the input: 700 × 3 = 2100. This is done using a dense layer.

After that, we reshape the flat vector into the original event format of shape (700, 3) — so each reconstructed particle has three values again. This mirrors the format of the original input.

```
decoded = Dense(input_shape[0] * input_shape[1], activation=activation_decoder)(encoded)
decoded = Reshape((input_shape[0], input_shape[1]))(decoded)
```

As in the variational autoencoder, we use a **sigmoid activation** in the final layer. This ensures that the output values are between 0 and 1 — matching the scale of the input features, which have been normalized. Sigmoid is ideal here because it constrains the output range, avoids extreme values, and fits naturally with our loss functions mean squared error when working with normalized data.

Convolutional Neural Network (CNN)

Convolutional neural networks are powerful tools for learning spatial hierarchies in structured data. While CNNs are traditionally used for image processing, they are well-suited for our dataset because the data has a fixed two-dimensional shape: 700 particles × 3 features. By thinking of the data as something like a simple image with one color channel, we can use convolutional layers to find useful patterns in nearby values and make use of the parameter-sharing benefits of convolutional layers.

Autoencoders built with CNNs can capture local patterns in the data — for example, correlations between a particle's features or nearby particles. These models tend to be robust and computationally efficient due to the reduced number of parameters and shared kernel weights. Just like with the other architectures, CNN-based autoencoders learn to compress the input into a lower-dimensional latent representation and then reconstruct it back to the original form.

This architecture is deterministic, like RNNs. Each event gets encoded into a single latent vector (not a distribution). This means it cannot generate new samples by sampling from a

learned latent space. However, its deterministic nature simplifies training and can have benefits on reconstruction performance.

Architecture

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 700, 3, 1)	0
conv2d (Conv2D)	(None, 700, 3, 64)	1,024
global_average_pooling2d (GlobalAveragePooling2D)	(None, 64)	0
dense (Dense)	(None, 64)	4,160
dense_1 (Dense)	(None, 2100)	136,500
reshape (Reshape)	(None, 700, 3, 1)	0
conv2d_transpose (Conv2DTranspose)	(None, 700, 3, 1)	10

The encoder starts with a series of **2D convolutional layers**. These layers apply learnable filters to extract increasingly abstract features from the input, using various filter sizes and depths. Each layer captures different aspects of local patterns within the particle data. We used multiple combinations of filter sizes and depths (e.g. [64, 32, 16] or [128, 64, 32]) to explore the optimal feature extraction strategy. The convolutional layers use padding to preserve spatial dimensions and are followed by activation functions like ReLU, Tanh, or Sigmoid.

To reduce the spatial dimensionality and transition from feature maps to a latent vector, we apply **global average pooling**. This operation computes the average of each feature map, effectively summarizing the entire spatial structure into a compact form. A final **dense layer** then projects this representation into the latent space of a specified dimension (e.g. 32, 64, or 128), which serves as the compressed encoding of the event.

```
for filters, kernel_size in zip(encoder_filters, kernel_sizes):
    encoded = Conv2D(filters, kernel_size, activation=params["activation_encoder"], padding="same")(encoded)

encoded = GlobalAveragePooling2D()(encoded)

encoded = Dense(latent_dim, activation=params["activation_encoder"])(encoded)
```

The decoder mirrors the encoder, but in reverse. It first uses a dense layer to expand the latent vector into a flat vector of the original input size. This vector is then reshaped into the input shape (700, 3, 1), restoring the structure of particles and their features. Finally, a **Conv2DTranspose** layer is applied. This layer is the inverse of a convolutional layer — it "upsamples" the data, allowing the network to reconstruct a smooth and detailed version of the original input.

```
decoded = Dense(input_shape[0] * input_shape[1], activation=params["activation_decoder"])(encoded)
decoded = Reshape(input_shape)(decoded)
decoded = Conv2DTranspose(input_shape[2], (3, 3), activation="sigmoid", padding="same")(decoded)
```

We use a **sigmoid activation function** in the last layer to ensure that the output values lie within the [0, 1] range. Since the input features are scaled accordingly, this maintains consistency between inputs and outputs and enables effective learning.

Just like in the other architectures, we optionally include **batch normalization** and **dropout** layers during training to improve stability, accelerate convergence, and reduce overfitting.

Hyperparameters and Optimization

To maximize model performance, we performed systematic hyperparameter tuning using **Optuna**, an optimization framework based on the **Tree-structured Parzen Estimator (TPE)** algorithm by default - a Bayesian optimization method that models the probability distribution of hyperparameters based on previous trial results. Rather than performing an exhaustive or random search, TPE builds two probabilistic models: one for parameter configurations associated with high-performing outcomes, and another for all others. It then chooses new hyperparameters by sampling from areas of the search space where the likelihood of improvement is highest. This saves us the time consuming work of trying out endless combinations of hyperparameters in order to find the best combination. During an optimization run, we can specify a maximum number of trials, and in each trial, Optuna evaluates a different combination of hyperparameters, updating its internal models based on the validation loss. This feedback loop enables Optuna to gradually converge toward an optimal or near-optimal configuration [6].

We let Optuna optimize the following hyperparameters [6]:

- Latent Dimensionality
  Controls the size of the compressed representation in the latent space.
  Range explored: latent_dim_min=32 to latent_dim_max=128

- Learning Rate
  Determines how quickly the model updates its weights during training. Affects convergence speed and stability.
  Range explored: learning_rate_min=0.00001 to learning_rate_max=0.01

- Activation Functions
  Introduce non-linearity into the network, enabling it to learn complex patterns.
  We intended to explore relu, tanh, sigmoid. For simplicity reasons we decided to stick to the most suitable one though, being sigmoid.

- Number of Layers and Units
  Defines the depth and capacity of the encoder and decoder.
  Optimized for RNN and CNN models: encoder_layers, decoder_layers

The tuning process is defined in hyperparameter_tuning.py [1]

Optimizer

We used the **Adam optimizer** for all models due to its adaptive learning rate, efficient convergence, and robustness to sparse gradients. It combines the benefits of AdaGrad and RMSProp, making it well-suited for our deep learning task [10].

Loss Functions

- **Variational Autoencoder (VAE):**
  A custom loss function is employed, combining two components:
    - **Reconstruction loss**: Calculated using Mean Squared Error (MSE), which measures how well the model reconstructs the input features (pT, η, φ).
      To reflect the varying importance of features in anomaly detection, we apply feature-wise weights, e.g., [3, 1, 1], giving more weight to pT due to its higher physical significance.
      These weights are configurable as a hyperparameter and can be tuned via the config file (e.g., config.yaml) to optimize model performance.
      A mask is also applied to exclude zero-padded particles from the loss calculation, ensuring the model learns only from meaningful input.
    - **KL divergence**: This term encourages the latent space to follow a standard normal distribution, improving the robustness and structure of the learned representation.
- **RNN and CNN Autoencoders:**
  These models use **Mean Squared Error (MSE)** as the primary reconstruction loss, measuring the difference between the input and the reconstructed output.

Cross-Validation

To monitor generalization performance, a **validation set** is created by splitting a fixed proportion of the training data. This allows for consistent and fair model evaluation across experiments.

Callbacks

A suite of **callbacks** is used to improve training efficiency and prevent overfitting:

- **Early Stopping:**
  Halts training when the validation loss does not improve for a predefined number of epochs (PATIENCE).

- **Model Checkpoint:**
  Automatically saves the best-performing model based on validation loss.

- **Learning Rate Scheduler:**
  Reduces the learning rate by 5% after each epoch, helping the model converge more smoothly over time.

Data Loading and Preprocessing

The dataset is divided into training, validation, and test sets using configurable proportions. To ensure a clean learning signal for the models, the training and validation sets include only background events (label = 0), while the test set contains a mix of background and signal (anomalous) events, with a fixed anomaly ratio to simulate realistic detection scenarios.

To make the training process more stable and help the models learn better, we apply **normalization** to each feature (pT, η, φ) before training. This means we scale the values so they are in a similar range, which helps the model process them more effectively.

- For transverse momentum (pT), we use a StandardScaler. This method centers the data around zero and scales it based on how much it varies. We use this because pT can have a wide range of values (e.g., some particles can have very high momentum while others have low), and it's important to make it easier for the model to handle.

- For pseudorapidity (η) and azimuthal angle (φ), we use a MinMaxScaler. This method transforms the values so they fall between 0 and 1. Since η and φ already lie in fairly bounded ranges, this simple scaling is enough.

These scalers are fitted only on the real, non-padded particle data from the training set, so the scaling is based on meaningful input and not influenced by zeros added for padding. The fitted scalers are saved so we can use the exact same transformation when validating or testing the model later.

To help the model distinguish between real and padded inputs, a binary mask is generated for each event. Each valid particle receives a mask value of 1.0, while zero-padded rows are assigned a value of 0.0. This ensures the model focuses only on meaningful data during training and evaluation.

For efficient data handling:

- The training set is loaded lazily via tf.data.Dataset generators, which allows streaming from disk and minimizes memory usage.

- The validation and test sets are fully loaded into memory and wrapped in TensorFlow [5] datasets for rapid batch access.

All datasets are batched and prefetched using tf.data.AUTOTUNE, optimizing data throughput and ensuring efficient model training.

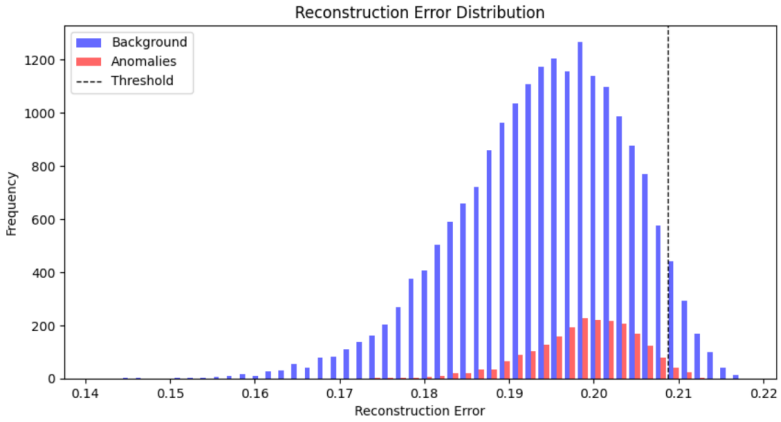
Evaluation

Evaluation Metrics and Interpretation

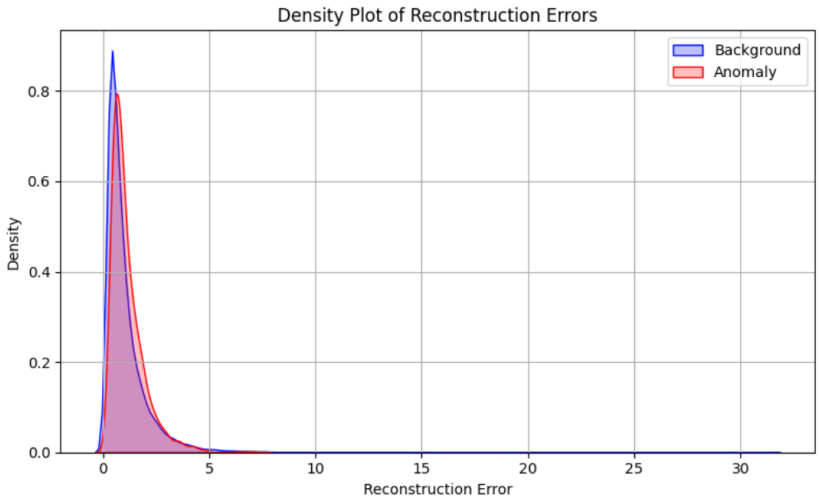
We used various different metrics and plots to verify the performance of our models and for being able to interpret their behaviour. The following plots are those of our VAE.

Reconstruction Error

The core metric for evaluating our autoencoders is the reconstruction error, which quantifies how well the model can reproduce the input data from its compressed latent representation. A higher reconstruction error typically indicates that the input is an anomaly, as it is poorly represented by the learned latent distribution. In the plot for the reconstruction error distribution, there is almost no visible distinction between background events and anomalies. This means our model is still performing somewhat poorly.



When observing the density plot of reconstruction errors, though, you can clearly see that the anomalies are overall getting a higher reconstruction error. This is our goal, so the training is definitely heading in the right direction.

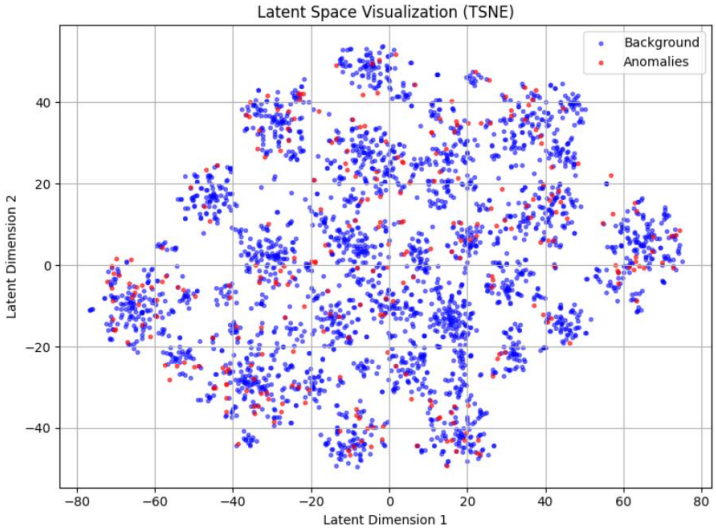


Training vs Validation Loss Curve

This plot helps assess the model's learning progress over epochs. The observed gap between training and validation loss suggests that the model has not yet converged and may benefit from further training to better generalize to unseen data. We can also observe that our model is not overfitting because the validation loss keeps being higher than the training loss.

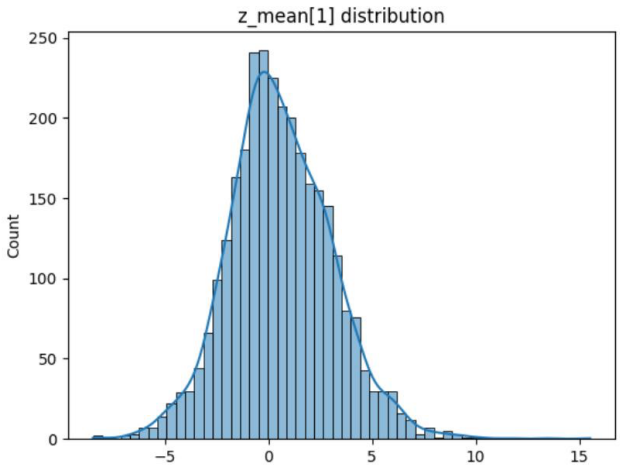
Latent Space Visualization

We visualize the compressed latent space to assess the structure the model has learned. So far, there are no visible distinctions between background and anomalous events that would enable us to tell that the latent features encode meaningful distinctions.



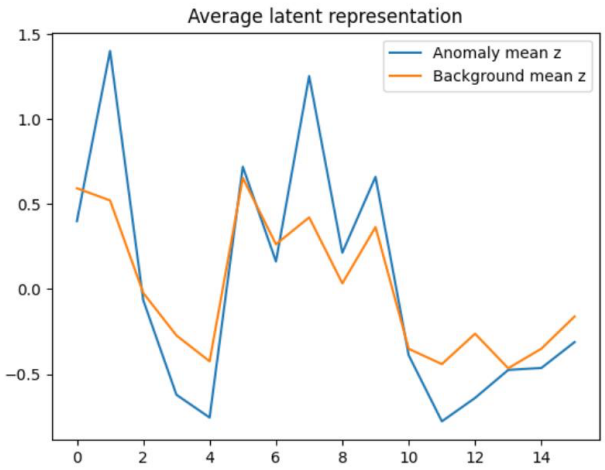
Distribution of z_mean Components

To verify that the latent variables follow a standard normal distribution, we plot the distribution of z_mean. This plot for one of the z_mean components shows a near-zero mean and unit variance, indicating that the KL divergence term, guided by the β parameter, is working as expected to enforce a Gaussian prior.



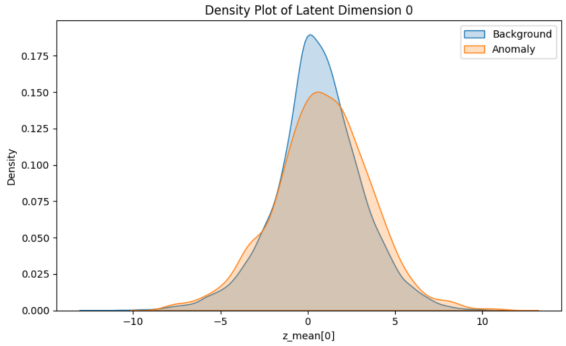
Average Latent Representation by Class

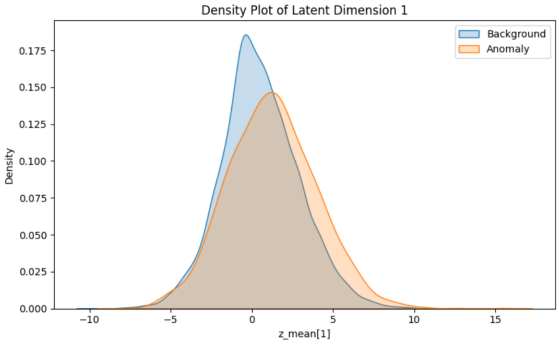
By plotting the average latent representation for background and anomaly events, we observe that the model begins to separate the two classes in latent space, though further training could enhance this separation.



Density Plot of Latent Dimensions

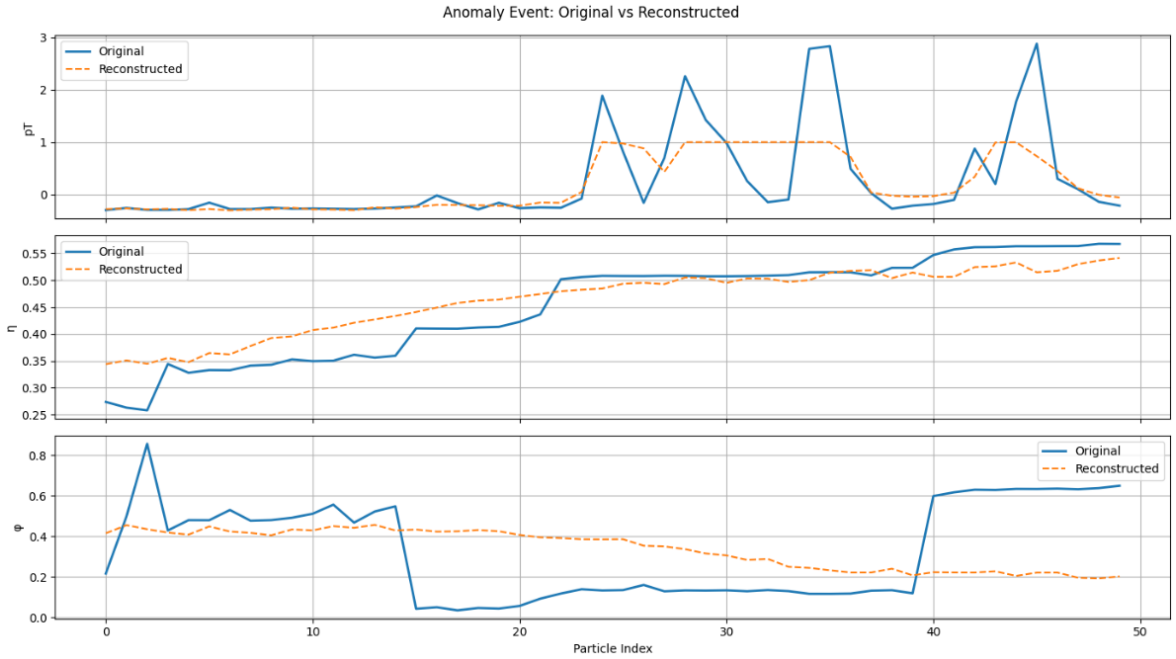
These plots help us understand which latent dimensions are most useful for distinguishing between background and anomalies. If some dimensions show strong separation, while others are not, this could point to potential areas for model refinement.

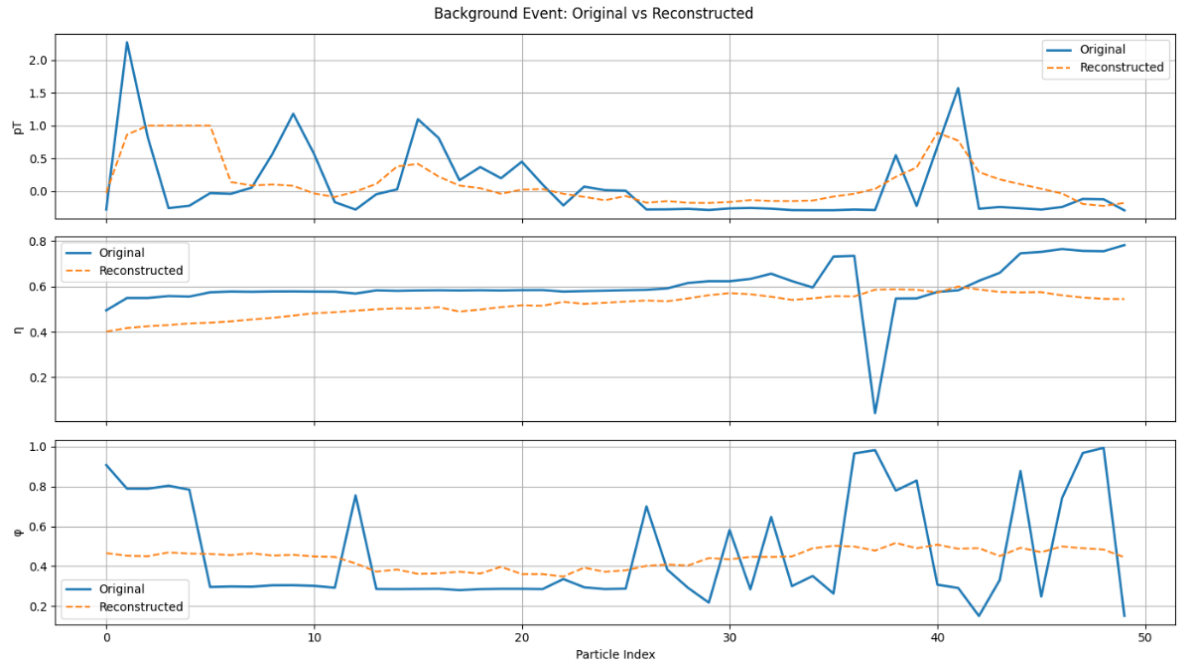




Reconstruction Performance on Background vs. Anomalies

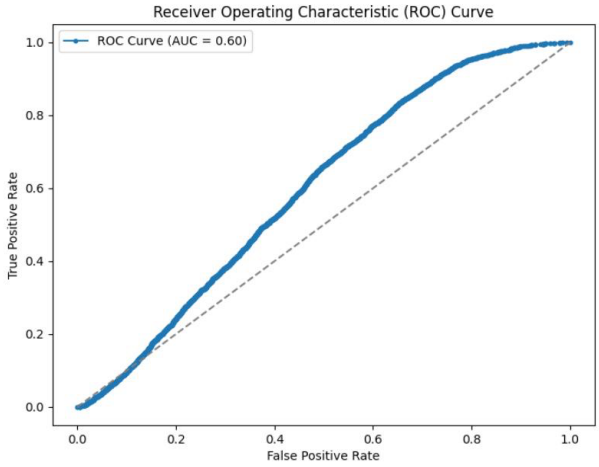
For both background and anomalous events, we visualize the original input features (the particle values pT, η and φ) against the reconstructed outputs. We can then see how good the model is able to reconstruct each of those features separately. The model performs a little better on background events but struggles more with anomalies, which is exactly what we want to see.





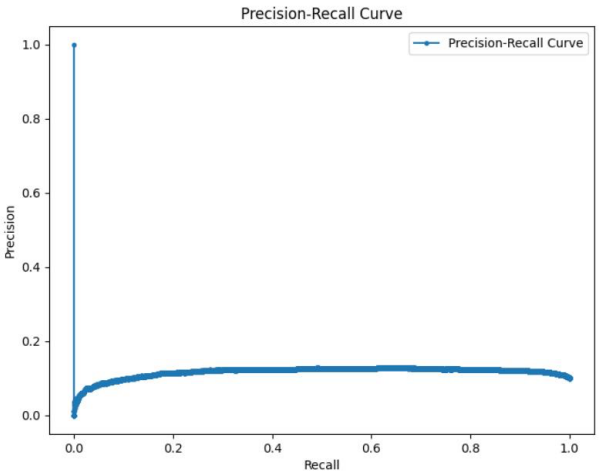
Receiver Operating Characteristic (ROC) curve

We further evaluate the model using the ROC curve, which plots the true positive rate against the false positive rate. This metric provides a threshold-independent assessment of the model's ability to distinguish between classes. The Area under the ROC Curve (AUC) summarizes this performance in a single number, with values closer to 1 indicating excellent separation between background and anomalies, and values near 0.5 suggesting performance no better than random guessing. Ours being at 0.6 means that our model is far from perfect, but better than random guessing [2].



Precision-Recall (PR) curve

This one is useful to assess performance under class imbalance, like it is the case for our anomaly to background events ratio. The PR curve plots precision (the proportion of predicted anomalies that are correct) against recall (the proportion of true anomalies that are detected). High values for both precision and recall across thresholds indicate robust model performance. This is not yet the case for us [4].



Baseline Performance

To estimate baseline performance, we first trained our model using a basic setup: no hyperparameter tuning (without Optuna [6]), fewer training samples, and fewer epochs. This gave us a starting point to compare how much our model improved after using better settings and more data.

Deployment

Our project focused on research and exploration. We wanted to test how well different deep learning models like CNNs, RNNs, and VAEs, can spot unusual patterns in particle collision data from the LHC. We did not actually deploy the models in a real-world system, but in the future, this kind of approach could be very useful in physics, for example, to help automatically detect rare or unknown events in particle experiments.

Potential Deployment Scenario

If we were to deploy our models, they could be added to the system that processes data from particle detectors like those at the LHC. After some basic preparation, each collision event could be run through our anomaly detection model. The model would then check how much the event differs from what it has learned to be "normal." If the difference is big, it could mean the event is unusual and worth looking into—maybe even something related to new physics.

To make the system fast and able to handle lots of data, we could use cloud services or powerful computers. The models could be made available using tools like TensorFlow Serving or PyTorch's TorchServe, depending on which framework we used.

Framework Suitability

We mainly used TensorFlow and Keras, which gave us everything we needed to build, train, and prepare our models for possible use. It also has tools to save trained models, make them available through APIs, and keep track of how well they perform. For more advanced use cases, we could use something like TensorFlow Extended (TFX) to manage the whole machine learning workflow from start to finish.

Monitoring and Maintenance

To keep the system working well in the future, a few things would need to be monitored and maintained:

- **Changing data:** The LHC might produce slightly different data over time because of updates or changes in how it is run. So, it is important to regularly check how well the model is doing and retrain it with new data when needed.
- **Correct input format:** The model expects the data to be in a specific format and properly scaled. We need to make sure the input data always follows this structure to avoid errors or wrong results.
- **Model versioning**: Every time we train a new version of the model, we should keep track of which data was used and what changes were made. This helps with troubleshooting and makes sure others can reproduce the results later.

Opportunities for Expansion

To make the project more useful and user-friendly, there are a few ways it could be improved:

- **Visual interface**: Creating a simple dashboard would help physicists look at strange events, compare them to known ones, and better understand what might be happening.
- **Using physics knowledge**: Adding known physics rules into the model could make the results more trustworthy and easier to explain.
- **Learning from labelled data**: If we get some examples of labelled unusual events, we could improve the model by letting it learn from both labelled and unlabeled data.
- **Sharing with others**: Making our code and models public would let other researchers test them, use them on real data, and improve them even further.

Pretrained Models and Testing

Some pretrained models are available in the .saved_models/{model_type} directory. These models can be useful for testing and benchmarking.

Note: Some models do not include a history.pkl file. This is because the training jobs were interrupted or aborted before completion, likely due to resource limitations or timeouts. While they may not represent optimal performance, they can still be used for inference or experimentation.

To test a saved model, you can run the following command:

—

python main.py --mode test --model_type vae --data_path your_data_path --model_path ./saved_models/vae/16042025

—

Replace vae with your model type (e.g., cnn, rnn), and make sure to point to the correct saved model path and your dataset.

Conclusion

Looking back, the metrics we chose - particularly the reconstruction error, training vs validation loss, and the reconstruction performance on background vs. anomalies - proved to be good indicators of the model's performance. They gave us insight not only into how well the model was learning, but also into how effectively it could distinguish background from anomalies, which was our primary goal.

While we initially considered training all three of our deep learning models, in the end we mainly focused on training the Variational Autoencoder (VAE), largely due to limited access to computing resources. Only one of us had the GPU power to train the full model on the complete dataset, while the rest worked with subsets and only very few epochs and Optuna trials. Despite that, we managed to get valuable results and established a good starting point for further training.

Overall, deep learning turned out to be a solid approach for the problem, especially because of its ability to learn meaningful representations without labels. If we had more time (and more GPUs), we could have explored additional architectures or fine-tuned hyperparameters further. Nonetheless, the project was fun, it is always better to apply your knowledge in a nice coding project than solely focus on theory. It gave us all a good hands-on experience with applying deep learning in a practical way.

References

[1] C. M. Blasco, R. Stockreiter, J. Rudus, *Anomaly_Detection*, GitHub repository. [Online]. Available: https://github.com/CarlaMiquelBlasco/Anomaly_Detection

[2] Coursera, “What Is a ROC Curve?,” *Coursera Articles*, [Online]. Available: https://www.coursera.org/articles/what-is-roc-curve

[3] OpenAI, *ChatGPT*. [Online]. Available: https://chatgpt.com

[4] Scikit-learn, “Precision, Recall and F-measure Metrics,” Scikit-learn Documentation. [Online]. Available: https://scikit-learn.org/stable/modules/model_evaluation.html#precision-recall-f-measure-metrics

[5] TensorFlow, *An end-to-end open-source machine learning platform*. [Online]. Available: https://www.tensorflow.org

[6] Optuna, *A hyperparameter optimization framework*. [Online]. Available: https://optuna.org/

[7] G. Kasieczka, B. Nachman, D. Shih, “LHC Olympics 2020: Anomaly Detection Dataset,” *Zenodo*, 2021. [Online]. Available: https://zenodo.org/record/4536377

[8] GeeksforGeeks, “Variational Autoencoders,” [Online]. Available: https://www.geeksforgeeks.org/variational-autoencoders/

[9] IBM, “Recurrent Neural Networks,” *IBM Think Blog*. [Online]. Available: https://www.ibm.com/think/topics/recurrent-neural-networks

[10] J. Brownlee, “Adam Optimization Algorithm for Deep Learning,” *Machine Learning Mastery*, Jan. 2017. [Online]. Available: https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/