Ludwig-Maximilians-Universität München

Faculty of Physics
AI in Physics Research Group

PP

Group A

Vyron Arvanitis
Emanuele Poggi
Paul Hofmann
Moritz Heidtmann
Angelos Aslanidis

Supervisor

Nikolai Krug

August 4, 2025

Contents

1 Introduction

In this lab course, we work with simulated data from the Belle II experiment to develop a fast and efficient simulation pipeline. The goal is to train and compare machine learning models that can predict early in the simulation chain whether an event is likely to survive later selection stages. This allows us to discard unimportant background events early, saving computational resources typically spent on full detector simulation and reconstruction (processess which are far more comptutaionaly expensive). The strategy is illustrated in Fig. 1, where a neural network acts as a pre-filter to speed up the overall simulation process.

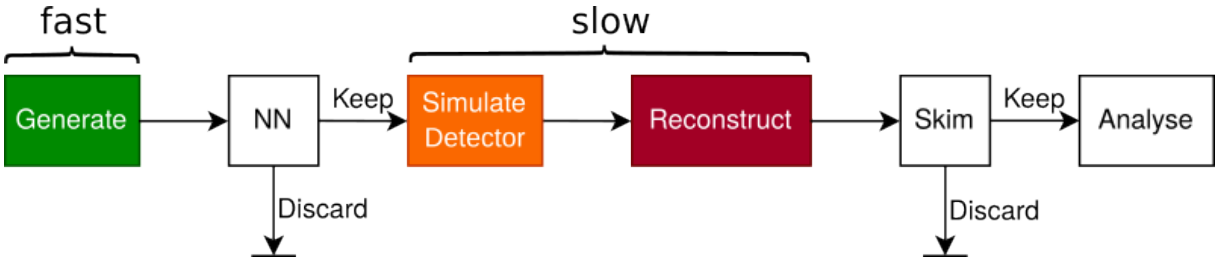

Figure 1: Smart simulation strategy: a neural network filters generated events before expensive detector simulation. Figure adapted from the official LMU AI Lab repository [6].

2 The Belle experiment

The Belle experiment is a so-called *B-Factory*, a type of particle accelerator specifically designed to produce large numbers of B *mesons*. These factories operate with a center-of-mass energy tuned to the $\Upsilon(4S)$ *resonance*, a state that decays almost exclusively into pairs of B and $\bar{B}$ mesons. This production mechanism enables detailed studies of B meson decays, allowing physicists to investigate phenomena such as *CP violation*, rare decays, and physics beyond the Standard Model. The Belle II experiment, located at the *SuperKEKB collider in Tsukuba, Japan*, is the successor to the original Belle experiment and significantly improves on its precision and data collection capabilities.

Unlike proton-proton colliders (such as the LHC), Belle II uses electron-positron (e^+e^-) collisions. Electrons and positrons are elementary particles with no internal structure, this offers several advantages. First, their interactions are much cleaner, and the background is easier to interpret—there is a notable absence of hadronic activity such as jets. Moreover, the absence of internal substructure allows us to precisely know the initial momenta of the colliding particles, enabling accurate reconstruction of the event kinematics. This precision is crucial for tuning the aforementioned center-of-mass energy to specific resonances, such as the $\Upsilon(4S)$.

B mesons are unstable particles that decay via the weak interaction, and their decays can be classified into three categories: hadronic, semileptonic, and rare decays. In *hadronic decays*, the B meson decays into only hadrons, such as combinations of kaons (K), pions (π), and other mesons. *Semileptonic decays* involve both hadrons and leptons, typically resulting in a final state containing a charged lepton ($e^{\pm}$ or $\mu^{\pm}$), a neutrino, and hadrons. *Rare decays* may involve photons, multiple leptons, or other suppressed processes that are especially sensitive to

effects beyond the Standard Model. These decay modes provide a rich ground for studying CP violation, flavor physics, and potential new physics.

The Belle II detector is structured as a layered system of subdetectors (like an onion) built around the interaction point. Closest to the interaction region is the Vertex Detector (VXD), composed of the PiXel Detector (PXD) and Silicon Vertex Detector (SVD), which provide precise tracking information for reconstructing decay vertices. Surrounding the VXD is the Central Drift Chamber (CDC), responsible for tracking charged particles and measuring their momenta and energy loss. Particle identification is handled by the Time Of Propagation (TOP) detector in the barrel region and the Aerogel Ring Imaging Cherenkov (ARICH) detector in the forward region.Photons and electrons are measured in the Electromagnetic Calorimeter (ECL) via the production of showers, while muons and long-lived neutral kaons (K_L^0) are identified in the outermost KLong and Muon (KLM) detector. The entire setup is enclosed in a 1.5 T magnetic field provided by a superconducting solenoid, which not only alligns the beams but also enables accurate momentum measurements. An overview of the collider (figure 2) and the particles that can be identified in each subdetector can be seen in table 1

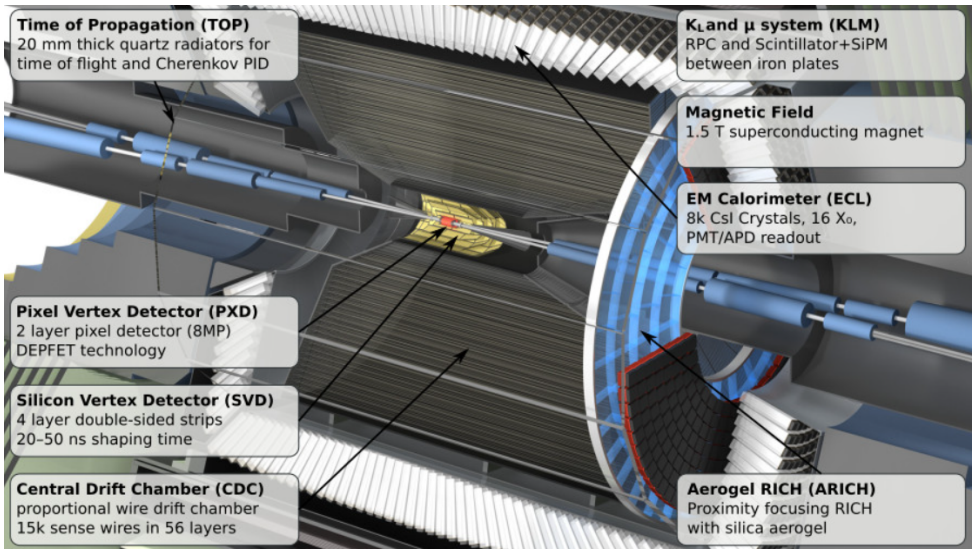

Figure 2: Closeup of the Belle II detector indicating all the different subdetectors [2].

Subdetector	Main Purpose
PXD + SVD (VXD)	Track reconstruction near the IP, measures short-lived particles
CDC	Tracking of charged particles
TOP / ARICH	Particle identification (e.g. distinguish π, K, p) via Cherenkov radiation
ECL	Detection of electrons and photons (electromagnetic showers)
KLM	μ identification and detection of K_L^0

Table 1: Belle II subdetectors and the particles they primarily detect.

2.1 Problem Description

The Belle II experiment collects an enormous amount of data, with its center-of-mass energy tuned to the $\Upsilon(4S)$ resonance *corresponding to a value of* $\sqrt{s} = 10.58$ *GeV*. While these "reso-

nant" events are the primary source for B physics, Belle II also records other types of data.

To achieve this center-of-mass energy the two beams colliding have respective energies of 7 GeV and 4 GeV. This assymetric beam configuration introduces a forward boost to the CoM system, forcing the produced B (and $\bar{B}$) mesons to travel measurable distances in the detector before they decay.

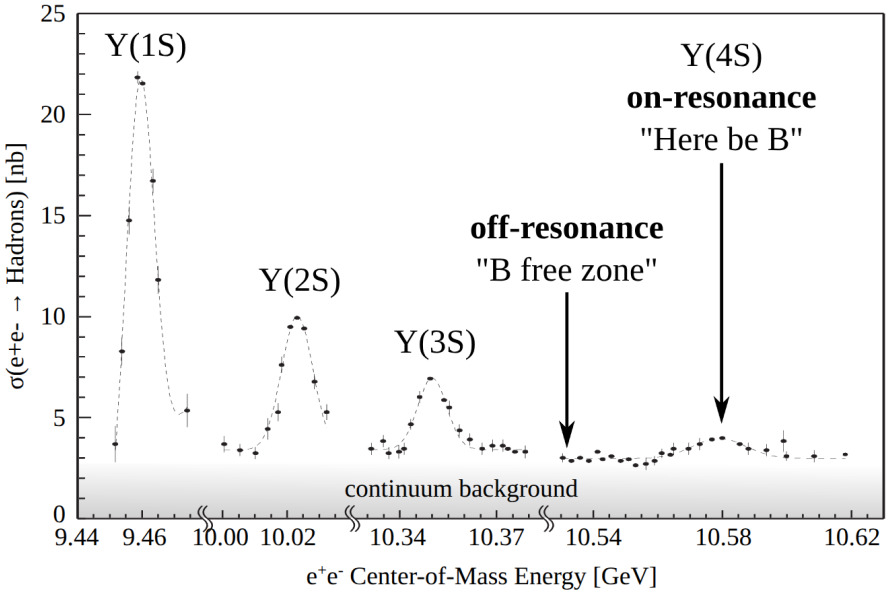

Figure 3: Cross section of e^+e^- with respect to the center-of-mass energy [GeV] [3].

To reduce the data volume and focus on interesting events, Belle II employs a two-level filtering system: the Level-1 hardware trigger (TRG) and the High Level Trigger (HLT), which together reduce the event rate before data is stored. Even after this, the number of events retained is still too large for direct analysis, so further selection — known as skimming — is performed to extract smaller datasets.

In parallel, Monte Carlo (MC) simulations are used to model the detector response and to train and validate analysis strategies. These simulated events must pass through all stages of detector simulation, digitization, reconstruction, and selection, mirroring the real data processing pipeline. However,this process is computationally expensive, especially when simulating large background samples, most of which are eventually discarded by skims.

To address this issue, the goal of this lab course is to explore a technique known as *Smart Background Simulation*. Instead of fully simulating every event, we aim to train a neural network that can predict early on whether a given event will survive a later skim. This skim selects events in which at least one B^0 meson can be reconstructed through hadronic decays. After building several neural networks we evaluate their performance on an independent test set and calculate the speedup offered by each network.

3 Theory

3.1 Deep Sets

In many physical systems, such as collections of particles or events in high-energy physics, the data can be naturally represented as unordered sets. Traditional neural networks, however, are not inherently permutation-invariant. This means that they assume an order in their input. To address this, Deep Sets provide a principled way to process such set-structured data while maintaining invariance to the order of elements. Figure 4 visualizes the structure of a Deep Set model.

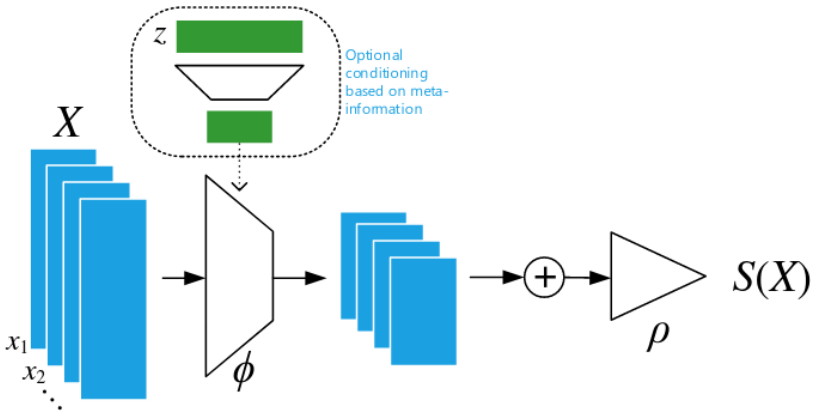

Figure 4: Schematic illustration of a Deep Set model.

A Deep Set model operates by applying a shared function ϕ independently to each element in the set. This produces a new set of embeddings, which are then aggregated using a permutation-invariant operation such as sum, mean, or max. The aggregated result is passed through a second function ρ, typically implemented as a multilayer perceptron (MLP), to produce the final output:

$$f(X) = \rho\left(\sum_{x \in X} \phi(x)\right) \tag{1}$$

This structure ensures that the model respects the symmetry of sets. The theoretical foundation of Deep Sets is discussed in [9], where it is shown that any function on a set that is invariant to permutations can be decomposed in this form.

In our project, we use this architecture to model collections of particles, where the order of particles is physically irrelevant. The Deep Set model thus provides a natural and efficient baseline for learning from such data.

3.2 Graph Convolutional Neural Networks

In contrast to Deep Set models, which treat the input as an unordered collection, Graph Neural Networks (GNNs) explicitly leverage relational information between data points. This is partic-

ularly important in our context, where particles are connected through a decay tree structure that naturally forms a graph. Each node in the graph represents a particle, and edges represent physical relationships (e.g., parent-child connections in the decay chain).

In terms of Deep Sets, graph convolutions can be seen as permutation equivariant operations. As long as the aggregation function (e.g., sum, mean, or max) is symmetric, the update remains invariant to the ordering of neighbors.

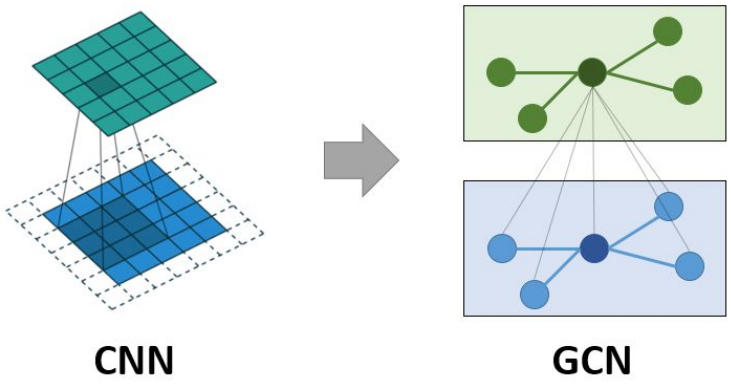

Figure 5: Analogy between CNNs and GCNs. In CNNs, pixels are updated based on spatial neighborhoods; in GCNs, nodes are updated based on graph neighborhoods.

Figure 5 illustrates the analogy between Convolutional Neural Networks (CNNs) and Graph Convolutional Networks (GCNs). In CNNs, pixels are updated based on their spatial neighborhoods, while in GCNs, nodes are updated based on their graph neighborhoods. This allows GNNs to capture complex relationships and dependencies in the data, making them particularly suitable for tasks involving structured data like particle decays.

3.3 Multi Layer Perceptron

The Multi Layer architecture is as

3.4 Transformer

The Transformer architecture was introduced by Vaswani [8]. The key ingredient in the success of the Transformer is the *self-attention* mechanism, which enables the model to capture long-range dependencies between elements of the input.

The encoder takes the input sequence (in our case, the set of particles in an event), maps tokens such as PDG identifiers to dense embedding vectors, adds positional encodings, and processes them through multiple layers of multi-head self-attention and feedforward sublayers. Each encoder layer contains connections and layer normalization, allowing information to propagate effectively through the network. In our architecture, only the encoder component of the Transformer is used, as our task is classification rather than sequence generation.

The outputs of the decoder are generated by attending both to the encoder outputs and to its

own previously generated tokens. It introduces a *masked* self-attention mechanism to prevent the model from attending to future positions in the output sequence. Since our problem does not involve generating sequences, the decoder component is omitted in our implementation.

The attention mechanism computes a weighted sum of vectors, where the weights are determined by the similarity between a query vector and a set of key vectors. In the multi-head self-attention setting, several independent attention "heads" are computed in parallel, allowing the model to focus on different types of relationships simultaneously. Self-attention enables each particle to directly incorporate information from all others in the event. The aforementioned property makes the transformer architecture highly suitable for particle physics tasks!

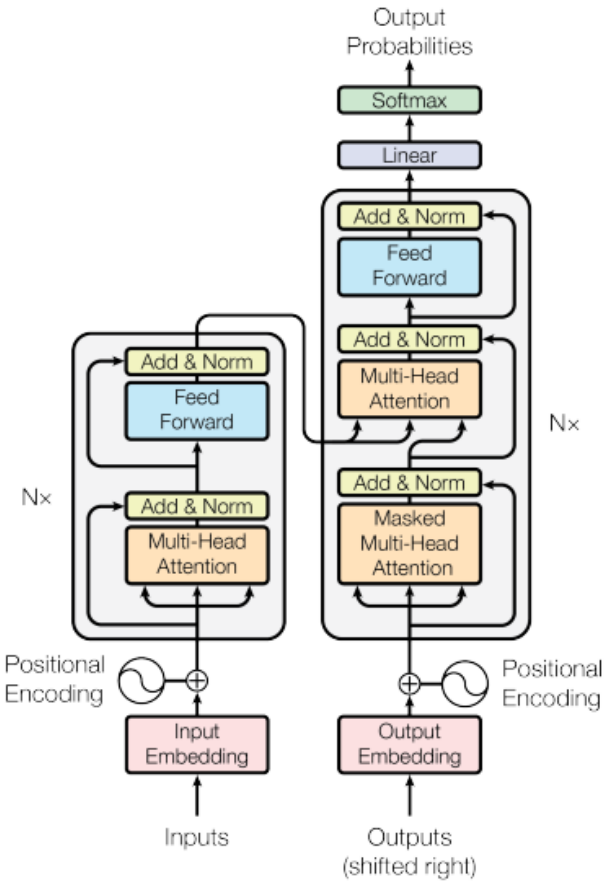

Figure 6: Transformer-model architecture [8]

4 Methodology

To identify the optimal model architecture, each group member conducted small-scale investigations targeting specific aspects of the model design. These investigations included the following:

- Does it help to increase the number of linear layers? Which sequence of GCN and Linear layers works best? (subsection 4.2)
- Could one make use of some (approximate) symmetries in the data? (subsection 4.3)

- Which activation function performs best? (subsection 4.4)
- Do we need to transform the values for the particle features? (subsection 4.5)
- Are there any redundant features in the dataset that could be removed? (subsection 4.6)
- Methods to combat overfitting (subsection 4.7)
- Ideas for different model architectures (section 3.4 and 5.7)
- Is it possible to use a standard MLP as well? If yes how?

4.1 The Dataset

The dataset of the experiment consists of simulated particle collision events, with each event containing multiple decay particles. These events contain a variable number of final state and intermediate particles. Hence, they can be structured as a graph, where each particle is a node and the edges represent the relationships between particles (e.g., parent-child relationships in decays). The graph can be seen in figure 7.

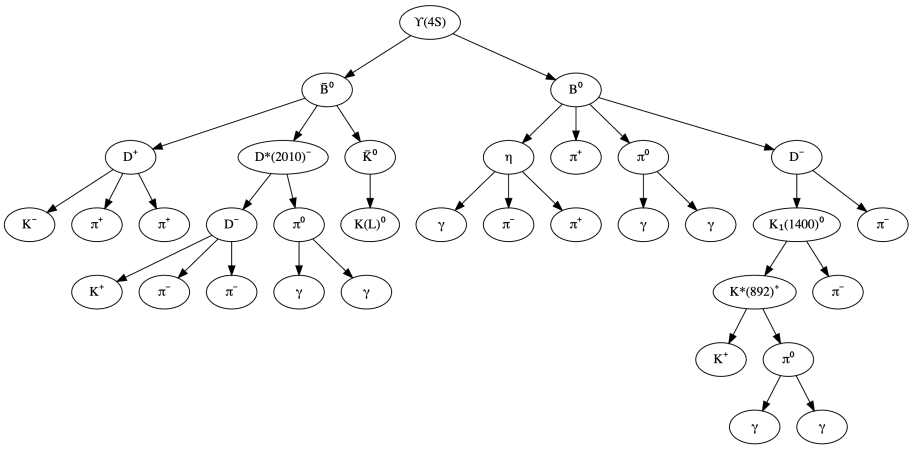

Figure 7: Example of a particle decay graph. The nodes represent particles, and the edges represent the decay relationships between them.

Each particle is described by a fixed number of features, which include:

- `prodTime` — the production time of the particle,
- `energy` — the energy of the particle,
- `x`, `y`, `z` — the spatial coordinates of the particle at production,
- `px`, `py`, `pz` — the momentum components of the particle,
- `pdg` — the PDG identifier of the particle, indicating its type,
- `index`, `mother_index` — the particle's index and the index of its mother particle (if any).
- `label` — this indicates weather or not the particle has passed the downstream event selection or not, it takes binary values 0 or 1

Each event is represented as a tensor of shape `(num_particles, num_features)`. Since the number of particles can vary between events, padding is applied to ensure consistent input dimensions across batches. These padded values are properly masked during training to prevent them from affecting the model.

4.2 Varying Layers in the DeepSet model

The architecture was developed to explore the optimal number and arrangement of GCN and linear layers. To support this flexibility, an additional script (deepset_gcn_variator.py) was introduced. This script allows dynamic model construction and naming based on the layer configuration.

The naming convention follows the pattern:

`DS_{TotalLayers}_GCN_{GCNIndices}`.

For example, the model `DS_6_GCN_135` consists of six layers (including input and output), with the 1st, 3rd, and 5th layers implemented as GCN layers. All other layers default to linear.

This models introduces two additional configuration arguments:

- `hidden_layers` (int): Number of hidden layers. The total number of layers is `hidden_layers` + 3 (one input and two output layers)
- `gcn_layers` (list): A list of layer indices to be implemented as GCN layers. For example `gcn_layers` = [2,4] replaces the 2nd and 4th layers with GCNs.

Layer behaviour is defined as follows:

- If index 1 is included in `gcn_layers`, the input layer is implemented as a GCN; otherwise, it is a linear layer.
- Hidden layers not specified in `gcn_layers` are standard linear layers.
- After all hidden layers, aggregation is performed using masked mean pooling (or mean over N if no mask is provided).
- The final output module consits of two linear layers speparated by a ReLU activation (i.e., a small MLP).

The complete architecture is summarized in table 2, where h is the number of hidden layers. u is the number of hidden units and is always set to $u = 32$. B denotes the batch size, N the number of particles, and f the number of features.

Table 2: Architecture of the variable DeepSet model with configurable GCN and Linear layers

Layer #	Description	Output shape	Parameters
1	Input layer (GCN or Linear with ReLU)	$[B, N, u]$	$u \times f + u$
2 - (h+1)	Hidden layers (GCN or Linear with ReLU)	$[B, N, u]$	$u \times u + u$ (per layer)
–	Masked mean pooling (or mean over N)	$[B, u]$	–
h+2	Linear + ReLU (global MLP)	$[B, u]$	$u \times u + u$
h+3	Linear (output layer)	$[B, 1]$	$1 \times u + 1$

After performing multiple calculations with this architecture (see folder saved_models/Layers_investigation), the best model was `DS_4_GCN_123`: 4 hidden layers, with GCNs at the input, 2nd and 3rd layer.

4.3 Exploiting Rotational Symmetry

Ideally, the collision of the electrons occurs parallel to the walls of the detector. This setup allows one to introduce cylindrical coordinates and exploit rotational symmetry around the z axis. In this coordinate system, the angular component can be neglected, and the transformation is defined as

$$r = \sqrt{x^2 + y^2}, \quad \text{and} \quad p_{xy} = \sqrt{p_x^2 + p_y^2}. \tag{2}$$

Here, r denotes the radial position, and p_{xy} is the transverse momentum in the xy plane. This transformation reduces the feature vector by two dimensions, thereby lowering the risk of overfitting without discarding relevant information about the particle collision.

The implementation requires an additional function, `transform_to_cylindrical`, located in `utils.py`. This function takes a DataFrame as input, which must contain at least the features `'x'`, `'y'`, `'px'`, and `'py'`. The output is a modified DataFrame in which the original features are replaced by `'r'`, and `'p_xy'`.

To enabale coordinate transformation, the function `preprocess` in `utils.py` takes an additional argument, `"coordinates"`, which can be set to `"cylindrical"`. If this argument is not specified, the data is processed using the original Cartesian coordinates.

Theoretically, one would expect improved performance by reducing the number of parameters by compressing essential information into a more compact format using cylindrical coordinates, while discarding unnecessary information by neglecting the angular coordinate. To test this hypothesis, the models `deepset`, `deepset_gcn`, and `deepset_combined` were trained using both Cartesian and cylindrical coordinate representations.

Training was performed on all four row groups of the training set over 10 epochs, using a 75%/25% train-validation split, a batch size of 256, and early stopping with a patience of 5.

Table 3: Validation metrics for Cartesian vs. cylindrical coordinates

	Cartesian		Cylindrical	
Model	**Loss**	**Acc. (%)**	**Loss**	**Acc. (%)**
`deepset`	0.633	63.6	0.624	64.5 %
`deepset_gcn`	0.614	66.0	0.610	66.3 %
`deepset_combined`	0.478	77.5	0.4727	77.6 %

Table 3 presents the validation loss and accuracy for each model trained using both Cartesian and cylindrical coordinates. Training on cylindrical coordinates consistently leads to improved performance across all models. Specifically, the validation accuracy increases by 0.9% for `deepset`, 0.3% for `deepset_gcn`, and 0.1%. for `deepset_combined`.

4.4 Activation functions

Activation functions introduce non-linearities into neural networks, enabling them to approximate complex functions beyond simple linear mappings.

Rectified Linear Unit (ReLU)

The ReLU is a piecewise linear activation function widely adopted for its simplicity and efficiency, given by

$$\mathrm{ReLU}(x) = \max(0, x). \tag{3}$$

It introduces sparsity by zeroing out negative values. Also, it is efficient to compute and generally accelerates convergence in deep networks.

Leaky ReLU

Leaky ReLU modifies the ReLU function by allowing a small, non-zero gradient in the negative domain:

$$\mathrm{LeakyReLU}(x) = \begin{cases} x & \text{if } x \geq 0, \\ \alpha x & \text{otherwise}, \end{cases} \tag{4}$$

with $\alpha \in (0, 1)$, here set to 0.01. LeakyReLU prevents complete inactivity of neurons by allowing gradient flow for negative inputs. It is slightly more flexible than ReLU with minimal additional computational cost.

Exponential Linear Unit (ELU)

ELU is a smoother alternative to ReLU, designed to improve gradient flow and convergence, given by

$$\mathrm{ELU}(x) = \begin{cases} x & \text{if } x \geq 0, \\ \alpha(\exp(x) - 1) & \text{otherwise}, \end{cases} \tag{5}$$

with $\alpha > 0$ (here set to 1). It produces negative outputs that push mean activations closer to zero, improving learning dynamics. The non-zero gradient for negative inputs enhances backpropagation compared to ReLU.

Investigation results

`deepset`, `deepset_combined`, and `deepset_gcn` were trained using ReLU, Leaky ReLU and ELU to find a suitable activation function for the optimal model. The results are presented in table 4. ELU consistently underperforms across all models. Leaky ReLU yields the best results

Table 4: Validation metrics for different activation functions

	ReLU		Leaky ReLU		ELU	
Model	**Loss**	**Acc. (%)**	**Loss**	**Acc. (%)**	**Loss**	**Acc. (%)**
`deepset`	0.633	63.6	0.628	64.1	0.639	63.1
`deepset_combined`	0.478	77.5	0.478	77.3	0.487	77.0
`deepset_gcn`	0.614	66.0	0.614	66.1	0.6312	64.0

for both `deepset` and `deepset_gcn`, while `deepset_combined` achieves its highest performance with ReLU.

4.5 Transforming the feature values

For this investigation, we used our best model obtained after the first lab day — that is **CombinedModel with GCN** model — and focused on normalizing the inputs. The implementation details can be found in Appendix A.1.

Essentially, subtracting the mean and dividing by the standard deviation of each feature, calculated across both the batch and particle dimensions.

4.6 Unecessary Features

To investigate whether or not some features are unnecessary in our analysis, we used again the same model (**CombinedModel with GCN**) and evaluated it, subtracting features via physical arguments. We performed six tests.

First of all, we removed the momentum vector ($\vec{p}$). The argument for this was that the energy and momenta of a particle are associated with the well-known relationship:

$$E^2 = p^2 + m^2$$

so, in principle, the model could reconstruct the momenta from the energy (or vice versa), assuming the mass is constant or implicitly learned.

For the same reason mentioned above, we explored removing the energy of the particles.

Afterwards, we removed the position vectors x, y, and z. This started as a simple test, however the following argument was made: in our problem, there exists a cylindrical symmetry — the x and y axes can be arbitrarily chosen (as the beams are considered to be colliding along the z axis). Therefore, we considered removing only the x and y coordinates.

However, after consideration we came to the conclusion that this is physically wrong because removing both x and y would eliminate any information about the transverse plane. Therefore, we opted to remove only the x position coordinate, as a compromise to test sensitivity to symmetry arguments.

In the end, we removed the `prodTime` feature just for further analysis, since its physical importance was unclear and we wanted to check if it had any significant impact on performance.

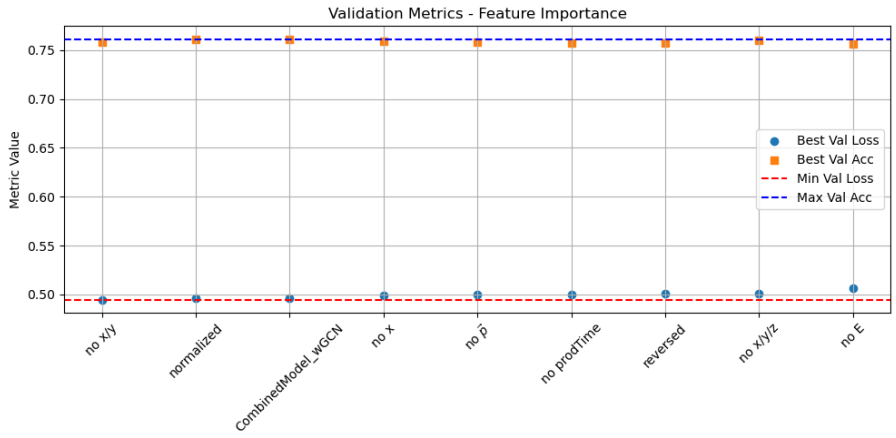

Figure 8: Validation accuracy and loss for different experiments. The horizontal dashed lines indicate the best performance achieved with the baseline model. Note: the **reversed** model refers to the configuration where the GCN and DeepSet layers were swapped.

From figure 8 it is evident that no significant improvement in model performance was observed when removing features. In all cases, the validation accuracy and loss remained close to the baseline values. The only slight performance gain was observed when applying normalization to the input features, confirming its importance as a preprocessing step.

4.7 Overfitting and Mitigation Techniques

Overfitting occurs when a model learns not only the underlying patterns in the training data but also noise or statistical fluctuations that do not generalize to new, unseen data. This results in low training loss but poor performance on validation or test sets. In high-energy physics tasks, overfitting can severely undermine the model's ability to detect meaningful physical signatures, especially in the presence of detector noise and variable event topologies.

To combat overfitting, we employed several regularization strategies that improve generalization:

Early Stopping

Early stopping halts training when the validation loss stops improving, even if the training loss continues to decrease [4]. This prevents the model from over-optimizing on the training data, offering a simple yet effective guard against overfitting.

Batch Normalization

Batch normalization normalizes the activations within each mini-batch to have zero mean and unit variance [5]. This reduces internal covariate shift and stabilizes learning, which allows for higher learning rates and often acts as an implicit regularizer, discouraging overfitting.

Dropout

Dropout randomly sets a subset of neurons to zero during training [7], forcing the network to develop redundant representations. This reduces co-adaptation of neurons and enhances robustness, as the model cannot rely on any single pathway to make predictions.

Regularization (L1/L2)

Regularization adds a penalty term to the loss function based on the magnitude of the model's weights. L2 regularization (weight decay) discourages overly large weights by penalizing their squared magnitude. This constrains model complexity and promotes simpler, more generalizable solutions [4].

Increased Training Data

Expanding the training set—either through real data or augmentation—exposes the model to a broader range of examples, making it harder to memorize specific samples. More diverse data reduces variance and improves the model's ability to generalize. In physics applications, techniques such as event reweighting or simulation-based augmentation can help increase effective training diversity.

Collectively, these methods reduce model variance and promote generalization, which is essential for ensuring reliable performance on physical inference tasks.

Hyperparameter Optimization

Hyperparameter optimization involves empirically testing different configurations that influence the learning behavior of a neural network. Key hyperparameters include the number of layers, neurons per layer, activation function, dropout rate, L2-regularization parameter (λ), and learning rate.

For this task, we employed the Optuna library [1], which uses the Tree-structured Parzen Estimator (TPE)—a Bayesian optimization algorithm. Unlike random search, TPE models the relationship between hyperparameters and the objective function (in our case, validation loss), and strategically explores regions of the hyperparameter space that are more likely to yield better results.

Search intervals were selected based on common literature values. After multiple trials, the following configuration produced the best performance:

- Dropout rate: 0.179
- Embedding dimension: 25
- Weight decay: 2.18×10^{-5}

This optimization process helped fine-tune the model's capacity and regularization, ultimately improving generalization and performance on the validation set.

4.8 Speedup

To evaluate the efficiency of using a NN as a filter before detector simulation, we define the speedup as the ratio of time needed to process events *without* and *with* the NN, for the same number of positively selected (skimmed) events.

Let:

- t_g: time for event generation
- t_s: time for detector simulation and reconstruction
- $r = 0.05$: fraction of events passing the original skim (without NN)
- f_1: true positive rate (TPR)
- f_0: false positive rate (FPR)
- X: number of events generated

Further, we define:

$$\begin{aligned} X_i &= r \cdot X \quad \text{(number of events passing skim)} \\ X_i &= f_1 \cdot X' \quad \text{(true positives from NN)} \\ \Rightarrow X' &= \frac{r}{f_1} \cdot X \quad \text{(number of events to be generated with NN)} \\ P &= f_1 \cdot X' + f_0 \cdot (1 - r) \cdot X \quad \text{(events passing NN)} \end{aligned}$$

The speedup is the ratio of total time without and with NN:

$$\text{Speedup} = \frac{X \cdot (t_g + t_s)}{X' \cdot t_g + P \cdot t_s}$$

Substituting X' and P into the equation:

$$\text{Speedup} = \frac{f_1 \cdot (t_g + t_s)}{t_g + t_s \cdot \left(f_1 + f_0 \cdot \frac{1-r}{r}\right)}$$

Finally, assuming $t_s = 100 \cdot t_g$, we simplify to:

$$\text{Speedup} = \frac{101 \cdot f_1}{1 + 100 \cdot \left(f_1 + f_0 \cdot \left(\frac{1-r}{r}\right)\right)}$$

4.9 ROC Curve and AUC Evaluation

To evaluate the performance of each model, we use the Receiver Operating Characteristic (ROC) curve, a widely adopted metric for binary classification tasks. The ROC curve plots the **True Positive Rate (TPR)** against the **False Positive Rate (FPR)**.

Definitions

- **True Positive Rate (TPR):** It measures the proportion of actual positives that are correctly identified:
$$\text{TPR} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$
- **False Positive Rate (FPR):** It measures the proportion of actual negatives that are incorrectly identified as positives:
$$\text{FPR} = \frac{\text{FP}}{\text{FP} + \text{TN}}$$

The ideal model achieves a high TPR while maintaining a low FPR. The performance of a model can be quantified using the **Area Under the Curve (AUC)**. A higher AUC indicates better separability between the two classes.

Interpretation

- AUC = 1.0: Perfect classifier.
- AUC = 0.5: Random guessing.
- AUC $<$ 0.5: Model performs worse than random (potentially inverted labels).

5 Summary of Evaluated Architectures

Based on our cumulative research, we implemented and evaluated a variety of model architectures to better understand which components and design choices contribute most to performance. Rather than being pre-selected as the most promising, these models served as investigative tools to explore the effects of architectural elements such as embeddings, graph structure, and normalization. Simpler baseline models were also included to isolate and quantify the impact of these additions.

5.1 General setup

All models described below take at least the following arguments:

- `num_features` (int): Number of input features
- `units` (int): Number of hidden units

To describe the shapes of the model layers, we adopt the following notation:

- B: Batch size
- N: Number of particles per event
- f: Number of input features
- u: Number of hidden units (= 32 in every model)

The input to every model is given by B batches, each containing N particles, each having f features. The input shape is therefore $[B, N, f]$. Any deviation will be further specified for each model individually.

5.2 Deep Set Model

This model follows the Deep Sets framework (as introduced in section 3.1), in which the same transformation is applied independently to each element of the input set, followed by a permutation-invariant aggregation. All other models are built upon this base architecture.

The architecture of the model is summarized in table 5.

Table 5: Deep Set Model Architecture

Layer #	Description	Output shape	Parameters
1	Linear + ReLU (shared across particles)	$[B, N, u]$	$u \times f + u$
–	Masked mean pooling (or mean over N)	$[B, u]$	–
2	Linear + ReLU (global MLP)	$[B, u]$	$u \times u + u$
3	Linear (Output layer)	$[B, 1]$	$1 \times u + 1$

5.3 Combined Model

Each PDG ID is mapped to an embedding vector, which is then concatenated with the corresponding particle features. The resulting embeddings are concatenated with the original particle features before being processed by the rest of the network.

The input to this model includes both the feature vector of each particle and the corresponding PDG ID vector.

The additional arguments introduced by this model are:

- `embed_dim:` Embedding dimension ($= e$)
- `num_pdg_ids:` Number of PDG ID categories ($= n_{\text{PDG}}$)

The architecture of the model is summarized in table 6.

Table 6: Architecture of the Combined Model with PDG embedding and Deep Set structure

Layer #	Description	Output shape	Parameters
1	Embedding lookup (PDG code)	$[B, N, e]$	$(n_{\text{PDG}} + 1) \times e$
–	Concatenation of features and embeddings	$[B, N, f + e]$	–
2	Linear + ReLU (shared across particles)	$[B, N, u]$	$u \times (f + e) + u$
–	Masked mean pooling (or mean over N)	$[B, u]$	–
3	Linear + ReLU (global MLP)	$[B, u]$	$u \times u + u$
4	Linear (output layer)	$[B, 1]$	$1 \times u + 1$

5.4 DeepSet with GCN

This model combines a Graph Convolutional Network (GCN) layer with the Deep Set architecture. The GCN captures local structure between particles as defined by an adjacency matrix

(typically based on mother-daughter relations).

The additional input this model takes is an adjacency matrix that encodes the particles pairwise relationships. The adjacency matrix is a symmetric binary matrix that encodes

- mother-daughter and daughter-mother relationships between particles,
- as well as self-connections (i.e., diagonal entries are 1).

This structure ensures that information can flow both directions along the graph edges, while also preserving each particle's self-information during message passing in the GCN. The models architecture is summarized in table 7.

Table 7: Architecture of the Deep Set structure with additional GCN layer

Layer #	Description	Output shape	Parameters
1	Graph Convolution (GCN)	$[B, N, u]$	$u \times f + u$
2	Linear + ReLU (shared across particles)	$[B, N, u]$	$u \times u + u$
–	Masked mean pooling (or mean over N)	$[B, u]$	–
3	Linear + ReLU (global MLP)	$[B, u]$	$u \times u + u$
4	Linear (output layer)	$[B, 1]$	$1 \times u + 1$

5.5 Combined Deep Set with GCN

This model combines the architectures of the "Deep Set with GCN" (section 5.4) and the "Combined Model" (section 5.3): The PDG ID of each particle is mapped to an embedding vector and concatenated with its feature vector. The resulting enriched representation is then processed by the DeepSet-GCN architecture.

Additionally the input requires the PDG codes, and the adjacency matrix as mentioned in section 5.4. The architecture of the model is summarized in table 8.

Table 8: Architecture of the Combined Model with PDG embedding, GCN and Deep Set structure

Layer #	Description	Output shape	Parameters
1	Embedding lookup (PDG code)	$[B, N, e]$	$(n_{\text{PDG}} + 1) \times e$
–	Concatenation of features and embeddings	$[B, N, f + e]$	–
2	Graph Convolution (GCN)	$[B, N, u]$	$u \times (e + f) + u$
3	Linear + ReLU (shared across particles)	$[B, N, u]$	$u \times u + u$
–	Masked mean pooling (or mean over N)	$[B, u]$	–
4	Linear + ReLU (global MLP)	$[B, u]$	$u \times u + u$
5	Linear (output layer)	$[B, 1]$	$1 \times u + 1$

5.6 Combined Deep Set with GCN Normalized

This model extends the Combined Deep Set with GCN architecture (model 5.5) by introducing a normalization of the inputs according to section 4.5. The architecture is the same as in the case of 5.5 with an additional normalization layer.

5.7 Transformer Model

This architecture is quite different from the Graph Convolution and Deep Set approaches explored earlier. The Transformer leverages self-attention, a mechanism that enables the model to learn long-range interactions between particles — a capability particularly valuable in particle physics tasks.

The input to the Transformer model consists of both the particle feature vectors and their corresponding PDG codes. The PDG codes are first mapped to embedding vectors, which are concatenated with the original particle features. The combined representation is then projected to the Transformer's internal dimension before being passed through a stack of Transformer encoder layers.

The additional arguments introduced by this model are:

- `embed_dim` (int): Embedding dimension ($= e$)
- `num_pdg_ids` (int): Number of PDG ID categories ($= n_{\text{PDG}}$)
- `num_heads` (int): Number of attention heads in each Transformer layer
- `num_layers` (int): Number of Transformer encoder layers
- `dropout_rate` (float): Dropout probability applied in the Transformer layers

The architecture of the Transformer model is summarized in Table 9.

Table 9: Architecture of the Transformer Model with PDG embedding

Layer #	Description	Output shape	Parameters
1	Embedding lookup (PDG code)	$[B, N, e]$	$(n_{\text{PDG}} + 1) \times e$
–	Concatenation of features and embeddings	$[B, N, f + e]$	–
2	Linear projection to Transformer units	$[B, N, u]$	$u \times (f + e) + u$
3	Transformer encoder stack	$[B, N, u]$	per-layer: MHSA + FFN
–	Mean pooling over particle dimension	$[B, u]$	–
4	Linear output layer	$[B, 1]$	$1 \times u + 1$

5.8 Performance Comparison

All models were trained on the full training dataset contained in `"smartbkg_dataset_4k_training.parquet"`, using a 75%-25% train-validation split. Training was conducted over 10 epochs using the Adam optimizer with a weight decay of 10^{-4}. Furthermore, the models were trained on Cartesian coordinates with features `"prodTime"`, `"x"`, `"y"`, `"z"`, `"energy"`, `"px"`, `"py"`, and `"pz"`.

Evaluation was performed on the full test dataset contained in `"smartbkg_dataset_4k_testing.parquet"`, using a batch size of 256.

Table 10 shows the validation loss and accuracy for every model presented in section 5.

Comparing the different models leads to the following conclusions: Incorporating PDG IDs through learned embeddings improves performance by enriching the particle representation. Similarly, adding GCN layers to the DeepSet architecture enhances performance by capturing

Table 10: Train and validation metrics for every model

	Train		Validation	
Model	**Loss**	**Acc. (%)**	**Loss**	**Acc. (%)**
`deepset`	0.629	64.0	0.633	63.6
`deepset_combined`	0.480	77.2	0.478	77.5
`deepset_gcn`	0.617	65.8	0.614	66.0
`deepset_combined_gcn`	0.463	78.2	0.467	78.0
`transformer`	0.446	79.2	0.443	79.4
`deepset_combined_gcn_normalized`	0.460	78.0	0.468	77.7

structural information from particle relationships. Combining both techniques, embeddings and GCNs, yields further improvement.

In contrast, normalizing the input features slightly reduces the performance of the `deepset_combined_gcn` model by 0.3% in validation accuracy.

Among the implemented models, the Transformer currently achieves the highest accuracy at 79.4%.

6 Results

6.1 Optimal model

Section 4 discusses strategies for regularization, feature processing, and hyperparameter optimization, while Section 5 investigates architectural modifications to identify the most effective model design. Based on the insights gained, we combined the most effective components into a single architecture, referred to as the Optimal Model.

The model builds upon a base architecture in which the input features are augmented with PDG ID embeddings, increasing the feature dimensionality by 25. The network consists of four sequential layers: the input layer, second, and third layer are implemented as GCNs, followed by one linear layer. This is followed by mean pooling across particles and a final two-layer MLP to produce the output.

Each trainable layer is followed by dropout (with a rate of 0.179), a LeakyReLU activation (slope 0.01), and Batch Normalization. Training was conducted using cylindrical coordinates, while all other settings remained consistent with previous experiments.

Table 11 presents the architecture of the optimal model. Achieving a training loss of 0.451 with a training accuracy of 79.0%, and a validation loss of 0.430 with a accuracy of 80.1%, it was identified as the best-performing model.

Table 11: Architecture of the Optimal Model

Layer #	Description	Output shape	Parameters
1	Embedding lookup (PDG code)	$[B, N, e]$	$(n_{\text{PDG}} + 1) \times e$
–	Concatenation of features and embeddings	$[B, N, f + e]$	–
2	GCN layer + BatchNorm, LeakyReLU, Dropout	$[B, N, u]$	$u \times (f + e) + u$
3-4	GCN layer + BatchNorm, LeakyReLU, Dropout	$[B, N, u]$	$u \times u + u$
5	Linear layer + BatchNorm, LeakyReLU, Dropout	$[B, N, u]$	$u \times u + u$
–	Masked mean pooling (or mean over N)	$[B, u]$	–
6	Linear + BatchNorm, LeakyReLU, Dropout	$[B, u]$	$u \times u + u$
7	Linear (output layer)	$[B, 1]$	$1 \times u + 1$

6.2 ROC curves and speedup results

Figures 9 and 10 present the ROC curves and speedup plots for our most promising models. We began with a simple Deep Set model (3.1) and, through a series of investigations, developed more complex architectures such as the Deep Set Combined model and the Deep Set Combined with GCN model. Our exploration culminated in a Transformer model and an "Optimal Model" configuration.

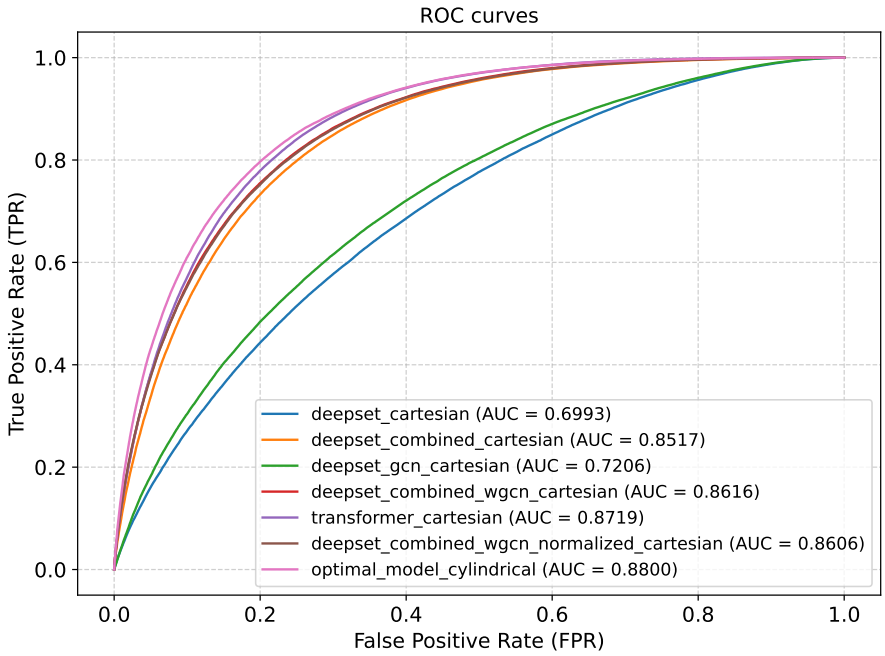

Figure 9: The resulting ROC curves of our different models

From figure 9, it is evident that all but the simplest baseline models achieve strong results, with most AUC scores exceeding 0.85, indicating strong classification performance. The Transformer and Optimal Model stand out with the highest AUC values.

While ROC curves indicate classification quality, they do not reflect entirely the impact on computational efficiency. To address this, we also evaluate the speedup—(4.8) if events are

filtered using the model predictions.

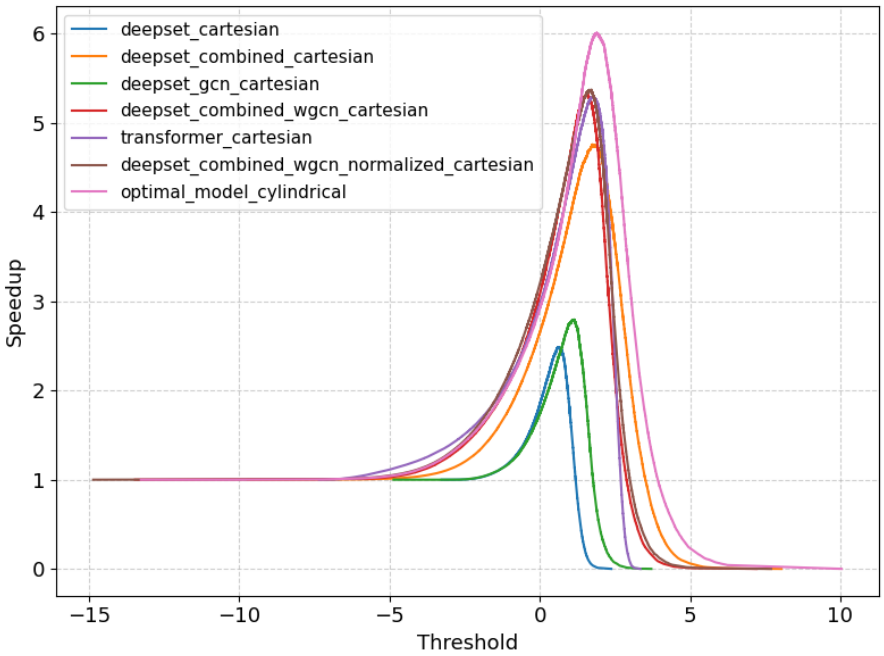

Figure 10: Speedup gained by our seperate models

From figure 9, we observe that the more advanced models are indistinguishable as far as their ROC performance is concerned, however they diverge in the achievable speedup. The Optimal Model attains the highest maximum speedup, followed closely by the Transformer and Deep Set Combined with GCN variants.

Table 12: Evaluation metrics on the test dataset for all models

Model	**Loss**	**Acc. (%)**	**AUC**	**Max. Speedup**	**Best Threshold**
DS	0.633	63.6	0.699	2.485	0.618
DS_combined	0.476	77.5	0.852	4.758	1.757
DS_gcn	0.614	66.0	0.721	2.795	1.149
DS_combined_wgcn	0.464	78.0	0.862	5.323	1.586
transformer	0.441	79.5	0.872	5.301	1.774
DS_combined_wgcn_normalized	0.467	77.6	0.860	5.373	1.719
optimal_model	0.432	80.1	0.880	6.009	1.899

The "Best Threshold" column is obtained from the set of decision thresholds returned by the `roc_curve` function. These thresholds correspond to decreasing cut values on the model's decision function that are used to compute the true positive rate (TPR) and false positive rate (FPR) pairs. The first threshold is always set to `np.inf`, which by definition yields zero predicted positives.

Among these thresholds, the one shown in the table is the value at which the model achieves its maximum estimated speedup. For example, the Optimal Model reaches a maximum speedup of 5.934 at a decision threshold of 1.901.

7 Conclusion

In this project, we successfully developed and evaluated several nerual network architectures aimed at identifying events likely to survive downstream selection in the Belle II simulation pipeline. Starting from a simple Deep Set baseline, we incrementally introduced architectural innovations such as PDG embeddings, graph convolutional layers, normalization, and coordinate transformations to improve predictive performance.

Out results demonstrate that incorporating domain knowledge into the model design significantly enhances classification accuracy and computational efficency. Specifically, the Optimal Model, which combines PDG embeddings, multiple GCN layers, and cylindrical coordinate inputs with stron regularization techniques, achieved the best performance acoss all metric. It reached an AUC of 0.880, test accuracy of 80.1%, and maximum speedup of 6.01×, enabling a substantial reduction in computational cost for event simulation.

While the Transformer model also showed strong performance (AUC = 0.872, speedup = 5.30), it was slightly outperformed by the Optimal Model, suggesting that tailored architectures that incorporate physical symmetries and relational structures may be better suited for this specific task than more general-purpose models.

Overall, this work confirms the feasibility of using machine learning models to intelligently filter unimportant events eraly in the simulation chain, allowing for more efficient resource usage in high-energy physics experiments.

Appendix

Note: LLMs like DeepSeek and ChatGPT were used for spell check and debugging of the code. They also assisted with creating docstrings.

A Code Listings

A.1 Normalization function

```
def normalize_inputs(inputs):
    x = inputs["feat"]  # position, momentum, production time, energy
    # x.shape = (batch_size, num_particles, num_features)
    mean = x.mean(dim=(0,1), keepdim=True)
    std = x.std(dim=(0,1), keepdim=True) + 1e-8  # avoid divide-by-zero
    x_norm = (x - mean) / std
    return {**inputs, "feat": x_norm}
```

References

[1] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[2] Belle II Collaboration. Belle ii detector overview. https://software.belle2.org/development/sphinx/_images/detector_labeled.jpg, 2023. Accessed: 2025-07-12.

[3] Belle II Collaboration. Belle ii detector overview. https://software.belle2.org/development/sphinx/_images/Uresonances.svg, 2023. Accessed: 2025-07-12.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[5] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *International Conference on Machine Learning*, 2015.

[6] Nikolai Krug. Preparation materials for particle physics lab group a. https://gitlab.physik.uni-muenchen.de/ai-lab/ss25/pp-lab-group-a/-/blob/main/preparation.md, 2025. Accessed: July 8, 2025.

[7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[8] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *CoRR*, abs/1706.03762, 2017.

[9] Manzil Zaheer, Satwik Kottur, Siamak Ravanbakhsh, Barnabas Poczos, Ruslan Salakhutdinov, and Alexander Smola. Deep sets. *Advances in neural information processing systems*, 30, 2017.