ZDC Neutron Energy Regression with XGBoost and GNN

I m not an expert so, dont ask hard questions

Satyajit Puhan

June 25, 2026

Project Objective

Main task

Predict true neutron energy in the Zero Degree Calorimeter (ZDC) from ECAL and HCAL hit information stored in a ROOT file, using an XGBoost regression model.

- Input: `myTree_20251002_100k_50_250GeV_neutron_penci.root`
- Target: neutron Monte Carlo energy, selected with PDG code 2112
- Output: a fast regression model saved as `outputs/zdc_xgb.json`
- Goal: convert detector hits into useful features, train a stable model, and verify that the predictions are physically meaningful

Presenter message

I am not only showing a machine-learning result; I am showing the full path from raw detector hits to a validated neutron-energy estimator.

Why This Is an Issue

Data difficulty

ROOT event data is not a simple rectangular table. Each event can contain a different number of ECAL and HCAL hits, so the raw arrays are jagged.

- Hit count changes event by event
- Positions and energies must stay aligned
- Loading everything at once can use too much memory

Modeling difficulty

The model needs fixed-size inputs while the detector gives variable-size hit lists.

- The model needs one consistent input shape
- The detector response is nonlinear
- The method must remain understandable and easy to verify

What is Our Goal : the detector sees deposited energy and hit positions; the model learns how those observations map back to true neutron energy.

ROOT File Contents

Branches used by `features.py`

Branch group	Variables
ECAL positions	`ecal_posX`, `ecal_posY`, `ecal_posZ`
ECAL energy	`ecal_energy`
HCAL positions	`hcal_posX`, `hcal_posY`, `hcal_posZ`
HCAL energy	`hcal_energy`
MC truth	`mcPar_PDG`, `mcPar_energy`

- Tree name as are in the root: `myTree`
- Dataset name: 100k neutron events from 50–250 GeV
- Only the listed branches are read, which avoids unnecessary memory use
- Events without a neutron truth label are skipped to keep the target clean

How the ROOT File Is Handled

Memory-safe reading

`features.py` uses `uproot` and `awkward` to read the ROOT tree in chunks instead of loading the full file into RAM.

1. Open the ROOT file and select `myTree`
2. Iterate over only the required branches using `tree.iterate(...)`
3. For each event, convert ECAL and HCAL jagged arrays into awkward
4. Compute shower summary values, then append one fixed 29-value row
5. Delete each chunk after processing to release memory immediately
6. Return `X` with shape $(N, 29)$ and `y` with shape $(N,)$

Default chunk size: 2000 events; it can be lowered for low-RAM machines.

Feature Engineering: Main Idea

Why features are needed

XGBoost expects fixed-length vectors, so each variable-length event is summarized into 29 physics-informed features. The features preserve total energy, shower position, shower spread, and ECAL/HCAL balance.

- 13 ECAL features describe the electromagnetic calorimeter response
- 13 HCAL features describe the hadronic calorimeter response
- 3 combined features describe total visible energy and detector sharing

$$\text{event} = [\text{ECAL}_{13},\ \text{HCAL}_{13},\ \text{combined}_3] \Rightarrow 29 \text{ features}$$

Feature Terms and Meanings

Term	Meaning
`total_E`	Sum of deposited hit energy; the first estimate of visible shower energy
`max_E`	Largest single hit energy; tells whether energy is concentrated in one cell
`n_hits`	Number of recorded hits; measures shower activity and detector occupancy
`mean_x,y,z`	Average hit position; locates the shower in detector coordinates
`std_x,y,z`	Position spread; measures how wide the shower is in each direction
`ew_x,y,z`	Energy-weighted centroid, $\sum x_i E_i / \sum E_i$; emphasizes high-energy hits
`rms_r`	Lateral shower radius around the energy-weighted center
`ecal_frac`	ECAL share of total energy; separates ECAL-heavy and HCAL-heavy events
`log_total_E`	Log-scaled total energy; reduces scale dominance from large-energy events

Target Label and Preprocessing

Truth energy

The target label is taken from `mcPar_energy` for the particle where `mcPar_PDG == 2112`.

- PDG 2112 means neutron
- The first matched neutron energy is used as the regression target
- Events without this target are not useful for supervised training
- Data is stored as `float32` to reduce memory use

Validation data

Saved validation file:
`outputs/val_results.npz`

- Validation events: 20,000
- Energy range: 50.01–249.99 GeV
- Mean validation energy: 149.83 GeV

How I Validate Using 20% Data

80/20 split

In `train.py`, the dataset is split into 80% training data and 20% validation data using `train_test_split`. The model learns only from the 80% training sample.

- Training set: used to fit the XGBoost trees
- Validation set: held out during fitting and used to test generalization
- Validation events saved in this project: 20,000
- Validation energy range: 50.01–249.99 GeV
- The validation set represents unseen events, so good validation metrics mean the model is learning a reusable detector-energy relationship

Why 20% Validation Is Important

Without validation

A model can memorize training events and still look good on the same data. That does not prove it will work on new ROOT files.

- Training metrics alone can be misleading
- Overfitting is common in powerful models
- Detector response must generalize across energy

With validation

The 20% held-out sample checks whether predictions remain accurate on unseen events.

- Val $R^2 = 0.9795$
- Val MAE $= 4.5888$ GeV
- Train/Val R^2 gap $= 0.0090$
- Small gap means low overfitting

Other Models I Could Use

Classical baselines

- Linear regression: simple, but too limited for nonlinear calorimeter response
- KNN regression: easy local averaging, but slow for large validation and inference sets
- SVR: nonlinear, but usually scales poorly with many events

Stronger alternatives

- Random forest: useful nonlinear baseline, but often less sharp than boosting
- Neural network: flexible, but needs more tuning and is less directly interpretable
- KNN graph method: preserves hit neighborhoods, but adds graph-building cost

Why XGBoost: strong accuracy, fast training, low memory use, and named feature importance.

Why XGBoost Is Better Here Than a KNN Graph

KNN graph idea

A KNN graph connects each detector hit to nearby hits in position space. It can describe local shower topology, but every event graph has to be built and processed.

- Needs a choice of k
- Sensitive to coordinate scaling
- Variable event sizes add overhead
- Harder to explain every prediction

XGBoost advantage

This project already compresses each event into meaningful shower features, so XGBoost can learn the energy mapping directly.

- Fixed 29-feature input
- Low memory with `tree_method=hist`
- Clear feature importance
- Strong validation: $R^2 = 0.9795$

Why Choose XGBoost

Reason for model choice

XGBoost is a strong fit because this project converts detector events into compact, tabular, physics-informed features. Once the event is represented by 29 numbers, boosted trees are a natural and efficient choice.

- Handles nonlinear feature interactions without needing hand-written calibration formulas
- Works well on medium-size tabular datasets
- Uses histogram tree building for speed and lower memory use
- Needs a simple Python stack: `uproot`, `awkward`, `xgboost`, `sklearn`, and plotting tools
- Gives feature importance for physics interpretation
- In this project, it handles 100k events in under 1 GB RAM according to the README

How XGBoost Works

Boosted trees

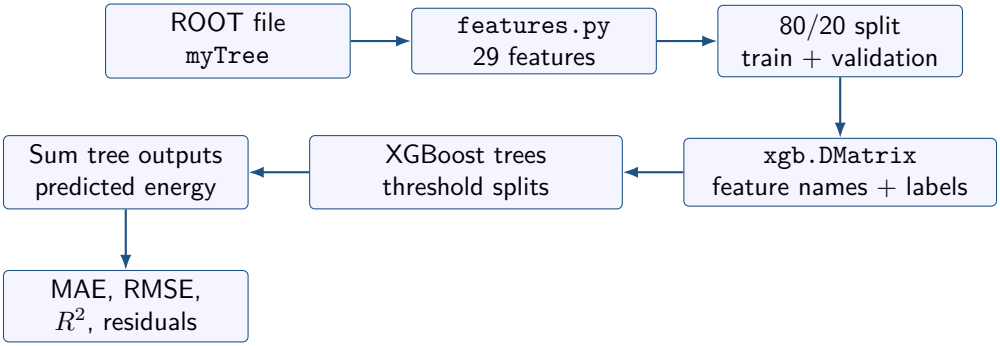
XGBoost builds many decision trees one after another. Each new tree corrects remaining error.

$$\hat{y}_i = \sum_{k=1}^{K} f_k(x_i), \qquad f_k \in \mathcal{F}$$

- Each tree splits on features such as `total_E`, `rms_r`, or `ecal_frac`
- Objective used here: `reg:squarederror`
- Evaluation metrics during training: RMSE and MAE
- Early stopping keeps the best validation round and reduces over-training risk
- Final prediction is the sum of all learned tree corrections

Presentation point: explain it as a sequence of corrections: rough estimate first, then more trees reduce the remaining residuals.

XGBoost Structure From the Code



Important correction for presenting

This code does not train a neural network, so there are no hidden layers or activation functions. The model uses tree split rules such as `feature < threshold`.

XGBoost Activation and Loss

Activation

Neural networks use activations such as ReLU or sigmoid. XGBoost does not use those activations.

- A tree node asks a yes/no question
- Example: is `hcal_total_E` below a threshold?
- The event follows left or right branches
- Final prediction is the sum of many tree corrections

Loss

The training objective in `train.py` is `reg:squarederror`.

$$\text{Loss} \approx \sum_i (E_i - \hat{E}_i)^2$$

- E_i = true neutron energy
- $\hat{E}_i$ = predicted energy
- RMSE is the square-root average error

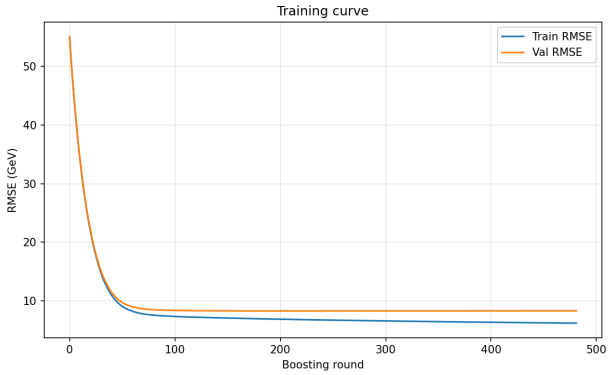
Training Configuration

Setting	Value in `train.py`
Train/validation split	80% / 20%
Number of boosting rounds	up to 1000
Early stopping	30 rounds
Learning rate	`0.05`
Max tree depth	6
Subsample	`0.8`
Column sample by tree	`0.8`
Min child weight	5
Tree method	`hist`
Random seed	42
Best round saved in metrics	451

Why these settings matter

Lower learning rate and early stopping reduce overfitting; row and column subsampling make trees more robust.

Figure: Training Loss Curve



This figure

The plot shows RMSE during boosting rounds for training and validation data.

- Training RMSE shows fit on seen events
- Validation RMSE shows performance on unseen events
- If validation error rises, the model is overfitting
- Early stopping keeps the best round, here round 451

Project File Map

File or folder	Role
`README.md`	Project summary, installation, run commands, output list, metric targets
`features.py`	ROOT reader and feature builder; creates one 29-value row per neutron event
`train.py`	Train/validation split, XGBoost training, metric calculation, model saving, and plot generation
`verify.py`	Independent quality report with six pass/warn/fail checks and energy-slice plots
`predict.py`	Loads `zdc_xgb.json` and applies the trained model to a new ROOT file
`tr.py`	Event-level hit inspector showing raw hits, prediction, truth, and residual
`outputs/`	The only figure/model folder used by this presentation: metrics, plots, validation predictions, and saved model

End-to-End Workflow

Training command

```
python train.py -root myTree_20251002_100k_50_250GeV_neutron_penci.root
```

1. Read ROOT branches in chunks
2. Extract ECAL, HCAL, and combined features using `features.py`
3. Split into 80% training and 20% validation samples
4. Train XGBoost with early stopping on validation performance
5. Save `zdc_xgb.json`, `metrics.json`, and `val_results.npz`
6. Generate diagnostic plots in `outputs/`
7. Run `python verify.py` to check accuracy, resolution, bias, uniformity, and overfit

Evaluation Terms

Metric	Formula or check	Meaning
MAE	$\frac{1}{N}\sum \lvert y-\hat{y}\rvert$	Average absolute GeV error; easy to explain physically
RMSE	$\sqrt{\frac{1}{N}\sum (y-\hat{y})^2}$	Penalizes larger mistakes more strongly than MAE
R^2	Variance explained	Closer to 1 means better regression quality
Residual	$(\hat{y}-y)/y$	Relative prediction error for each event
Bias μ	Mean residual	Systematic over- or under-prediction
Resolution σ	Std. of residuals	Energy resolution spread
Overfit gap	Train R^2 - Val R^2	Checks memorization

During presentation, I use R^2 for overall regression quality, MAE for practical GeV error, and residual σ for detector-resolution behavior.

Why R^2 Is Important

Simple meaning

R^2 tells us how much of the true energy variation is explained by the model.

- $R^2 = 1$: perfect prediction
- $R^2 = 0$: no better than predicting the mean energy
- Higher R^2 means the predicted trend follows the true energy trend
- In this project, validation $R^2 = 0.9795$

$$R^2 = 1 - \frac{\sum_i (E_i - \hat{E}_i)^2}{\sum_i (E_i - \bar{E})^2}$$

Presenter line

I use R^2 to show that the model is not only close in GeV, but also follows the correct energy ordering across 50–250 GeV.

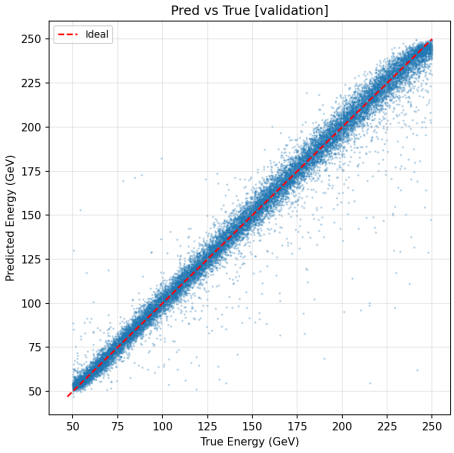
Main Numeric Results

Metric	Train	Validation
MAE (GeV)	3.9398	4.5888
RMSE (GeV)	6.1724	8.2846
R^2	0.9886	0.9795
Residual mean μ	0.0023	0.0030
Resolution σ	0.0415	0.0590
Resolution RMS	0.0416	0.0591

Interpretation

Validation MAE is 4.59 GeV, about 2.3% of the 200 GeV validation energy span. The validation R^2 of 0.9795 shows that the model explains most of the energy variation, while the small train–validation gap shows that the model is not simply memorizing the training sample.

Figure 1: Predicted vs True Energy

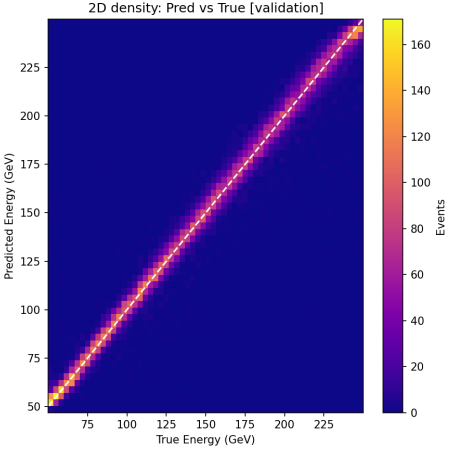


Why

Each point is one validation event. The x-axis is the true neutron energy and the y-axis is the XGBoost prediction.

- Red dashed line is perfect prediction
- Points close to the line mean low error
- The cloud follows the diagonal across 50–250 GeV
- This supports the high validation R^2

Figure 2: Density Map

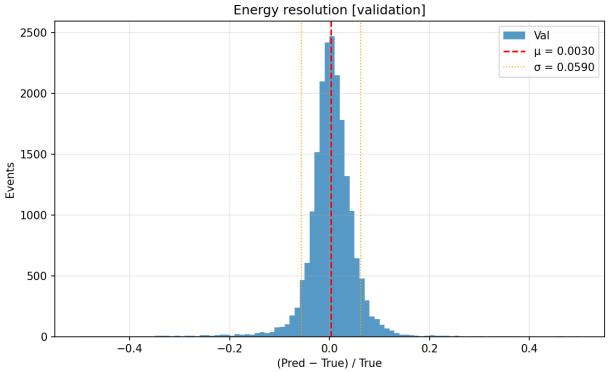


this figure

The density plot shows where many validation events are located. Brighter regions mean more events.

- Dense band near diagonal = stable predictions
- White dashed line = ideal behavior
- Useful because scatter plots can hide overlap
- Confirms most events are not extreme outliers

Figure 3: Resolution Histogram

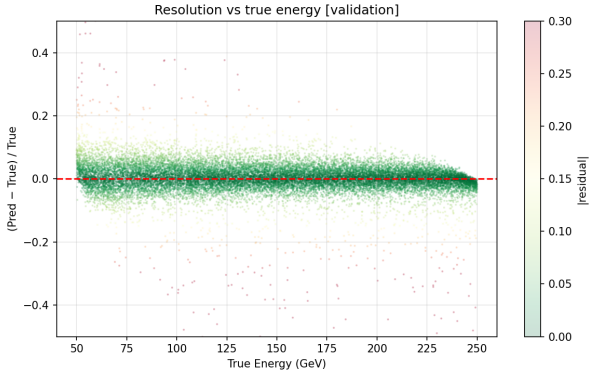


This figure

The histogram shows relative error: $(\hat{E} - E)/E$. It tells us whether the model is biased and how wide the error distribution is.

- Mean $\mu = 0.0030$: almost no systematic bias
- Sigma $\sigma = 0.0590$: about 5.9% resolution
- Target was $\sigma < 0.10$, so this passes

Figure 4: Resolution vs Energy



This figure

This plot checks whether errors depend on true neutron energy. A good model should not work only at low energy or only at high energy.

- Horizontal red line is zero residual
- Residuals remain centered around zero
- Color shows absolute error size
- Confirms performance across 50–250 GeV

Why HCAL Total Energy Has High Gain

What the saved model shows

Parsing `outputs/zdc_xgb.json` shows `hcal_total_E` has the largest total gain: it gives the biggest error reduction in tree splits.

Physics reason

Neutrons are neutral hadrons, so much of their measurable shower energy appears in HCAL.

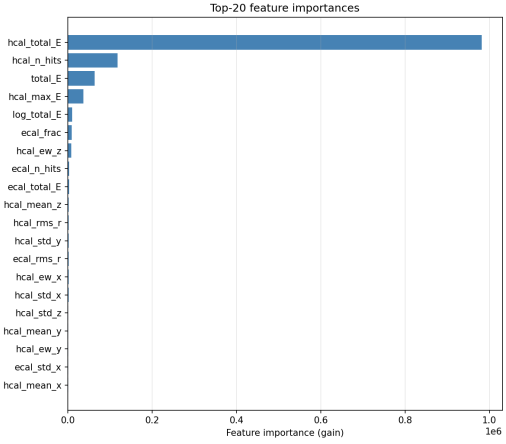
- HCAL is designed for hadronic showers
- Total HCAL energy tracks the neutron energy scale
- More neutron energy usually means more visible HCAL energy

Machine-learning reason

XGBoost gain is high when a split strongly reduces prediction error.

- `hcal_total_E`: top gain feature
- `hcal_n_hits`: second strongest in parsed model
- `total_E`: also important, but less specific than HCAL

Feature Importance and Physics Reading

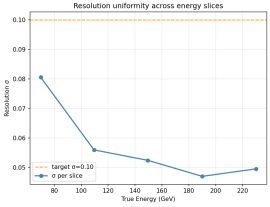


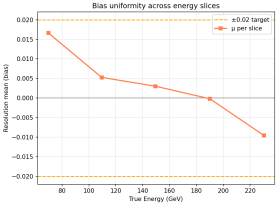
Why

XGBoost gain estimates how much a feature improves tree splits.

- Energy sums provide the main scale information
- Shower shape terms correct for spread, position, and leakage effects
- ECAL/HCAL balance helps separate different detector response patterns
- Importance makes the model easier to explain because each split is tied to a named feature

Verification Summary





Check	Result
Val $R^2 > 0.90$	PASS
MAE $< 5\%$ of range	PASS
Resolution $\sigma < 0.10$	PASS
Bias $\lvert\mu\rvert < 0.02$	PASS
Slice spread < 0.05	PASS
Overfit gap < 0.05	PASS

Energy-slice resolution ranges from about 0.047 to 0.081, with total spread 0.0335. This means the model remains reasonably uniform from low to high neutron energy.

ZDCgNN Section: Why Add It?

Purpose of this section

After validating XGBoost, we can discuss the graph neural network approach stored in `ZDCgNN/`. This helps explain what changes when we model each event as a graph of detector hits.

- XGBoost input: one fixed 29-feature vector per event
- ZDCgNN input: one graph per event, with hits as nodes
- XGBoost learns from engineered shower summaries
- ZDCgNN learns from local hit neighborhoods made by a kNN graph
- This section uses `ZDCgNN/model.py`, `dataset.py`, and `train.py`

ZDCgNN Graph Construction

From hits to graph

`dataset.py` converts each ROOT event into a PyTorch-Geometric `Data` object.

- Node = one ECAL or HCAL hit
- Node position = (x, y, z)
- Edge = connection to nearby hits
- Default graph rule: kNN with $k = 8$

Why kNN matters

kNN connects each hit to its nearest neighbors in detector space, so the model can learn local shower shape.

- Small k: fewer local messages
- Large k: more edges and more memory
- Here $k = 8$ balances locality and cost

ZDCgNN Node Features

Six features per detector hit

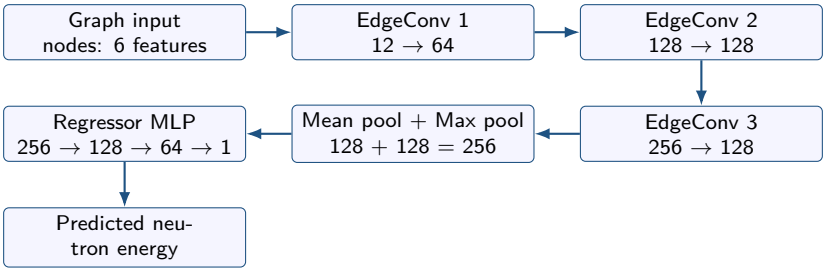
Each node feature vector is:

$$[x,\ y,\ z,\ E_{\text{dep}},\ \text{detector\_id},\ \text{layer\_id}]$$

Feature	Meaning for students
x, y, z	Where the hit happened in detector space
E_{dep}	Energy deposited in that hit
detector_id	0 for ECAL, 1 for HCAL
layer_id	Approximate detector depth from normalized z bin

Target label is the same idea as before: true neutron energy from `mcPar_energy`.

ZDCgNN Architecture From Code



Default hidden size is 64. The final output is one number: predicted neutron energy in GeV.

EdgeConv and Message Passing

Main idea

EdgeConv updates each hit by looking at nearby hits connected through the kNN graph. This is called message passing.

- Each edge compares a hit with a neighbor
- EdgeConv uses $(x_i \parallel x_j - x_i)$
- The symbol $\parallel$ means concatenate
- Aggregation is `max`

why

Think of each hit asking: "What do nearby hits tell me about the shower?" After several EdgeConv layers, each node contains richer local shower information.

Activation Functions in ZDCgNN

Activation used in the code

`model.py` uses ReLU activations inside every EdgeConv MLP and inside the final regressor.

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

- Negative values become 0
- Positive values pass through
- Adds nonlinearity to the model
- Helps learn complex detector response

Where ReLU appears

- EdgeConv MLP: Linear, BatchNorm, ReLU
- Second Linear block again uses BatchNorm and ReLU
- Regressor MLP: ReLU after 256 to 128 and 128 to 64 layers

Pooling, Dropout, and Output Layer

Global pooling

After EdgeConv layers, node embeddings must become one event-level vector.

- Global mean pool: average shower information
- Global max pool: strongest local response
- Concatenated vector size: 256

Regressor

The MLP maps the graph summary to one energy prediction.

- Linear 256 to 128, ReLU
- Dropout 0.2 to reduce overfitting
- Linear 128 to 64, ReLU
- Linear 64 to 1 output

ZDCgNN Training Setup and Loss

Split and optimizer

`train.py` uses a 70/20/10 split.

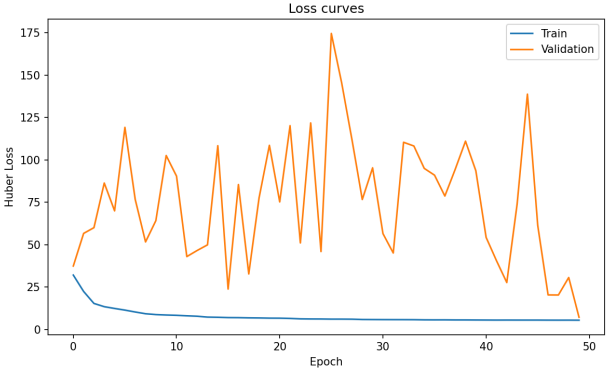
- 70% train
- 20% validation
- 10% test
- Optimizer: Adam
- Learning rate: 10^{-3}
- Scheduler: ReduceLROnPlateau

Loss

The GNN uses Huber loss with $\delta = 1.0$.

- Small errors behave like squared error
- Large errors behave more like absolute error
- Useful when outlier events exist
- Training tracks loss and MAE

ZDCgNN Loss Curves

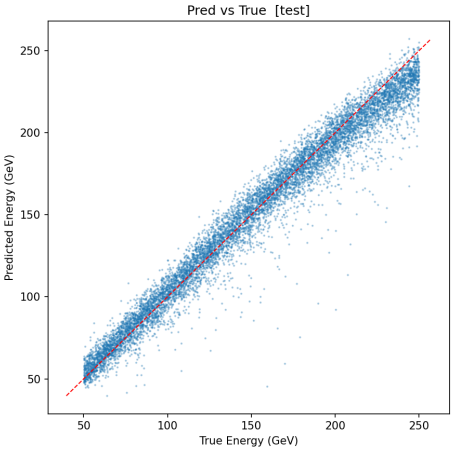


This figure

This plot shows train and validation Huber loss over epochs.

- Training loss shows learning on seen graphs
- Validation loss checks unseen graphs
- A widening gap suggests overfitting
- Stable validation loss means better generalization

ZDCgNN Prediction Plot



This figure

This is the GNN test-set predicted energy versus true neutron energy.

- Each point is one graph/event
- Red dashed line is ideal prediction
- Points close to the line mean better regression
- Test set is the final unseen 10%

ZDCgNN Strengths and Tradeoffs

Strengths

- Uses individual detector hits directly
- Learns local shower topology through kNN edges
- EdgeConv can learn spatial relationships
- Mean and max pooling capture whole-event behavior

Tradeoffs

- Graph construction costs time and memory
- More dependencies: PyTorch and PyTorch Geometric
- Harder to interpret than named XGBoost features
- Needs careful tuning of k, hidden size, epochs, and dropout

Presentation line: ZDCgNN is more expressive, while XGBoost is simpler, faster, and easier to explain.

Conclusion and Next Steps

Conclusion

The project now presents two approaches: XGBoost with engineered features and ZDCgNN with hit-level graph learning.

- XGBoost was chosen because the engineered features are tabular, compact, nonlinear, and interpretable
- The validation result is strong: $R^2 = 0.9795$, MAE = 4.59 GeV, resolution $\sigma = 0.0590$
- ZDCgNN uses kNN graphs, EdgeConv layers, ReLU activations, Huber loss, and graph pooling
- For future work: compare XGBoost and ZDCgNN on the same held-out test set and study outlier events