DESY Summer Student Project Report

Energy Regression with Graph Neural Networks on EM-showers in the HGCAL

Sena Durgut, Bogazici University

Supervised by: M. Scham, I. Melzer-Pellmann, D. Kruecker

September 22, 2022

Abstract

The CMS Collaboration will change the current end-cap calorimeter with the new High Granularity Calorimeter (HGCAL) in order to meet the challenges of the upcoming High Luminosity (HL) phase. The HGCAL is designed to be a high granularity imaging calorimeter with greatly improved spatial resolution and timing, resulting in better discrimination if pile-up events, better shower seperation and higher precision compared to the current end-cap calorimeter. However data analysis challenges arise due to the HGCAL's irregular geometry, it is difficult to find a grid-based representation for the HGCAL. Without a grid-based representation, many of the frequently used machine learning techniques like Convolutional Neural Networks (CNN) cannot be applied to the HGCAL data. On the other hand, Graph Neural Networks have received increasing interest lately by researchers for their capability to offer exceptionally powerful representations of complex systems and they allow one to represent the new irregular geometry of the detector. In this report the energy regression with Graph Neural Network on EM-showers in the HGCAL generated by photons (Energy $\in$ [50,100] GeV, η=2) is presented. Various graph operators and their optimization were studied.

Contents

1 Introduction

1.1 The CMS HGCAL

In the context of high energy physics (HEP), the ratio of the event rate to the total cross section is defined as luminosity. It serves as a crucial indicator of how many interactions an accelerator is capable of producing.

As part of the High Luminosity (HL) phase of the CMS Experiment, the High Luminosity (HL) upgrade of the existing Large Hadron Collider (LHC) faces numerous obstacles. The HL-LHC will integrate ten times higher luminosity than the LHC, creating considerable issues for radiation tolerance and event pile-up on detectors, particularly for in the areas close to the beampipe. The existing calorimeter end-caps composed of electromagnetic (CE-E) and hadronic (CE- H) sections were not designed to deal with such luminosity and its effects. As part of the HL-LHC upgrade, the CMS Collaboration will replace the present endcap calorimeters with a high granularity calorimeter (HGCAL) (Figure 1) [1]. The HGCAL has been designed to be a high granularity image calorimeter with vastly enhanced spatial resolution and timing, resulting in superior differentiation of pile-up events, improved shower separation, and subsequently more precise measurements.

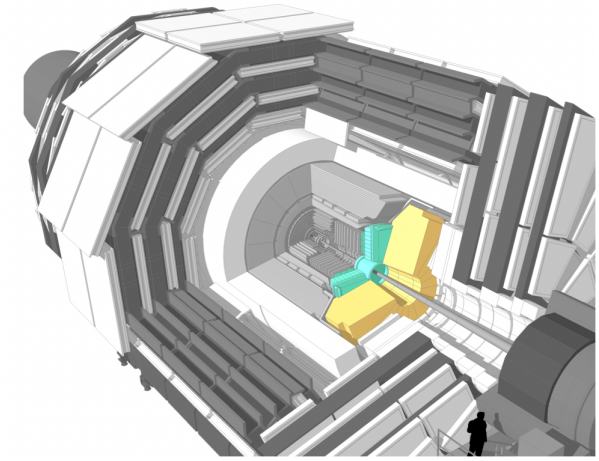


Figure 1: Present forward calorimeter (in yellow)

In the hadronic part of HGCAL, the scintillator sections have a grid-like structure, whereas the inner parts that contain the silicon is in the shape of a hexagon (Figure 2). Thus, the HGCAL possesses quite irregular geometry along the margins of these sections. In addition, it has a greater number of channels compared to the previous end-cap calorimeters. Because of the irregular geometry, one is forced to use particular flexible data structures, such as point clouds and graphs, when conducting further data analysis using machine learning.

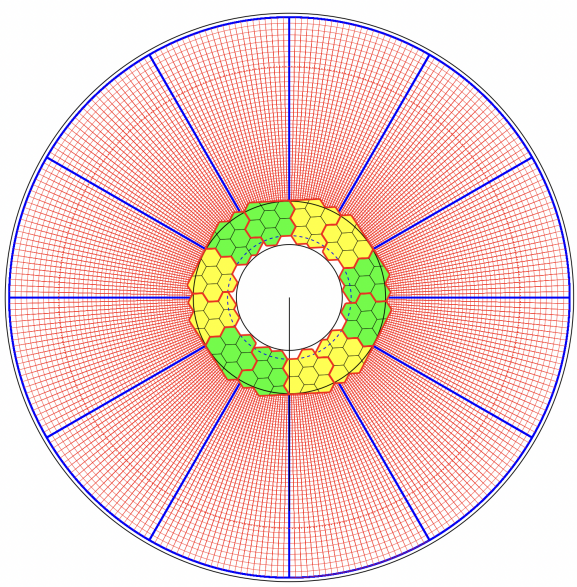


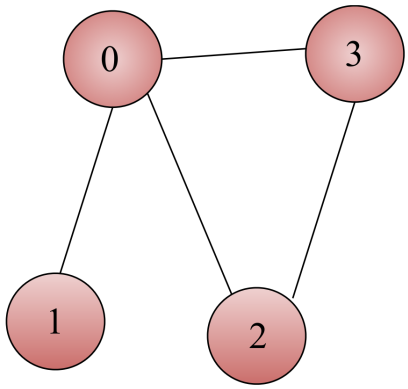
Figure 2: Layer in the CE-H with both silicon hexagons and scintillator tiles.

1.2 Graphs

One of the most widely used deep learning methods is the Convolutional Neural Networks (CNNs). (citation needed). In addition to the standard matrix multiplication that is carried out by traditional feedforward neural networks, CNNs implement at least one additional linear mathematical operation that is referred to as a convolution. They are highly efficient at processing data with a grid-like topology. The HGCAL's irregular geometry, on the other hand, cannot be transformed into a grid in any way. The existing software frameworks provide a neighborhood information based on the adjacency of the cells in which the hits are recorded.

A graph is a data structure consisting of two primary components: vertices and edges. It is possible to provide an adequate definition of a graph by denoting it as G = (V, E), where V is the set of vertices (or nodes), and E is set of the edges representing the connections between the nodes. Graphs are flexible tools that can effectively represent intricate network connections within a system.

The most common way to express a graph is via its adjacency matrix $\boldsymbol{A}$. An adjacency matrix is a 2-D array of $V \times V$ vertices. Each row and column represents a vertex. If the value of any element in the position a_{ij} is 1, it represents that there is an edge connecting vertex i and vertex j.



	0	1	2	3
0	**0**	**1**	**1**	**1**
1	**1**	**0**	**0**	**1**
2	**1**	**0**	**0**	**1**
3	**1**	**0**	**1**	**0**

Figure 3: A simple illustration of an undirected graph and its adjacency matrix: Each entry in the matrix indicates whether a connecting edge exists between the corresponding nodes.

Another important feature of graphs is that nodes can contain feature vectors and edges can have edge weights associated with them, which makes it possible to store more information within the graph.

1.3 Graph Neural Networks (GNNs)

Recently, tremendous progress has been achieved in the field of Graph Neural Network (GNN) research. Notably, a growing number of unique GNN architectures have been developed, including GCN, GIN, GraphSAGE and a number of others.
GNNs use a message passing mechanism to propagate information across the graph by exchanging information between adjacent nodes. The following is a concise summary of the underlying process for the message passing in GNNs:

1. Every node in the graph computes a message for each of its neighbors.
2. Messages are sent, and every node aggregates the messages it receives.
3. After receiving the messages, each node updates its attributes.

Ultimately, the entire process can be formulated as:

$$x'_j = \text{update}(x_i, \text{aggregate}([x_j, j \in N(i)])),$$

where x_i and x_j stands for the node features and $N(i)$ denotes one-hop distance neighbors of node i. Different message passing layers are constructed through the combination of a wide variety of aggregate and update functions.

2 Regression with GNNs

Regression is the process of predicting a target variable given a set of features.In this work, the energy of the particle generating an EM-shower in the HGCAL was regressed on the simulated hits of this shower to predict the energy of the particle causing the shower. PyTorch Geometric, which is an extension of the PyTorch deep learning framework and enables a wide variety of applications relating to graphs, was utilized for all of the work that was carried out. The features include energy and the position (x,y,z) values for each of the hits in the shower. To simulate these showers, photons with an energy of [50, 100] GeV and η =2 with η being the spatial coordinate describing the angle of a particle relative to the beam axis. Adam optimizer was used with the parameters of weight decay being $1.0 \times 10^{-}4$ and the learning rate being $2 \times 10^{-}4$. The batch size was 50, number of training batches was 5000, the number of the validation batches was 100 and the number of testing batches was 1. The number of input features is 4.

2.1 Evaluation Method

The evaluation of a regression model cannot utilize accuracy metrics as in classification tasks. Instead, the error metrics that have been developed for regression problems are utilized. Mean Squared Error (MSE) was used for evaluation in the early stages of this project. It is calculated by squaring the difference of the true value of the target variable energy and the predicted value:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(E_{true} - E_{predicted})^2$$

Because the predicted values are the energies of the particles, taking the square of the difference amplified the error for high energies whilst diminishing it for low energies. As a result, a change in the error function was required. Mean Relative Error (MRE) was used afterwards for the evaluation. It is calculated by taking the absolute value of the difference between true value and the predicted value of energy and dividing by the true value:

$$MRE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{E_{true} - E_{predicted}}{E_{true}}\right|$$

2.2 Linear Regression

Linear Regression is a very straightforward type of regression model where a linear hypothesis is constructed between the features and the target variable. It served as a baseline to compare the GNN architectures.

2.3 Graph Convolutional Network (GCNConv)

A Graph Convolutional operator proposed by Kipf et.al. and implemented in PyTorch Geometric was used. It's node-wise formulation is given as:

$$\mathbf{x}'_i = \mathbf{\Theta}^\top \sum_{j \in \mathcal{N}(v) \cup \{i\}} \frac{e_{j,i}}{\sqrt{\hat{d}_j \hat{d}_i}} \mathbf{x}_j$$

with

$$\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$$

where $e_{j,i}$ denotes the edge weight from source node i to target node i (default: 1.0) [2]

The architecture of the model with the GCNConv layers is shown below:

```
class GCN_Conv(torch.nn.Module):
    def __init__(self):
        super(GCN_Conv, self).__init__()
        n_features = conf.loader.n_features
        self.upscale_conv = GCNConv(n_features, 20)
        self.conv1 = GCNConv(20, 10)
        self.conv2 = GCNConv(10, n_features)
        self.lin = Linear(n_features, 1)

    def forward(self, batch):
        x = self.upscale_conv(batch.x, batch.edge_index)
        x = F.relu(x)

        x = self.conv1(x, batch.edge_index)
        x = F.relu(x)

        x = self.conv2(x, batch.edge_index)
        x = F.relu(x)

        x = self.lin(x)
        x = global_add_pool(x, batch.batch, size=batch.num_graphs)

        return x
```

Figure 4: Implementation of a neural network in PyTorch Geometric using GCNConv layers

The evaluation results were promising with the best training MSE being 18.52 and the best validation MSE being 28.72.

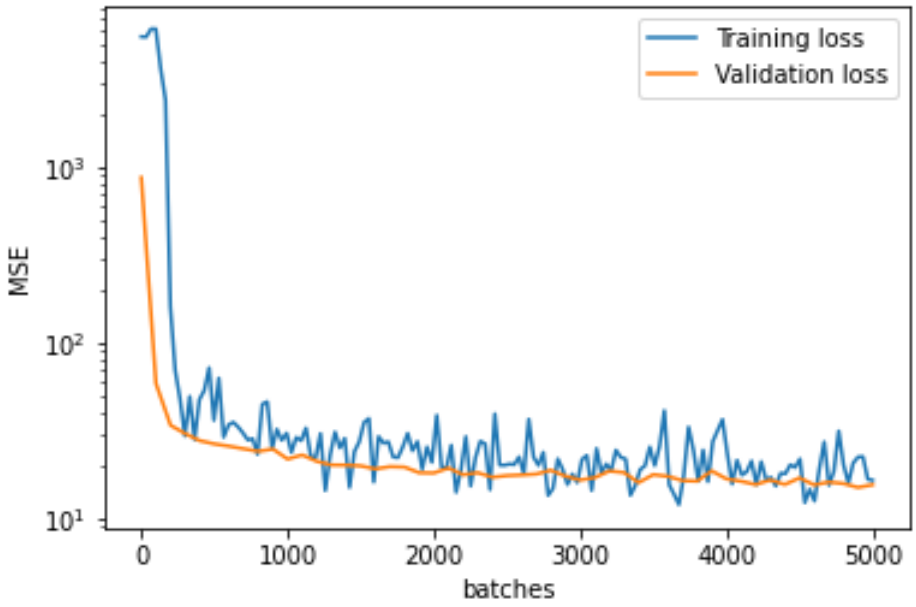


Figure 5: MSE vs. batches for GCNConv

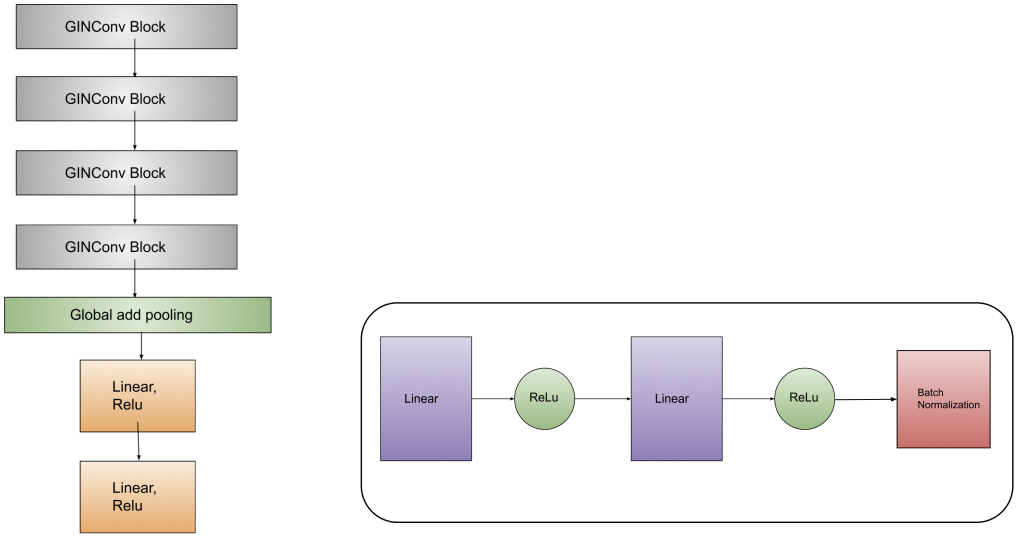
2.4 Graph Isomorphism Network (GINConv)

Graph Isomorphism operator, proposed by Xu et. al. is inspired by The Weisfeiler-Lehman Isomorphism Test, an algorithm for determining whether two graphs are topologically identical. Two graphs are considered topologically identical if there is a mapping between them that preserves the node adjacencies. The developers of this operator assert that it possesses superior power than that of any other graph operator. [3]

Its node-wise formulation is:

$$\mathbf{x}'_i = h_{\mathbf{\Theta}}\left((1+\epsilon)\cdot \mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \mathbf{x}_j\right)$$

where ϵ is a slight variation that is added to the calculation and can be a trainable parameter or a fixed scalar value and $h_{\mathbf{\Theta}}$ denotes a neural network, .i.e. a Multi Layer Perceptron (MLP). Implementation in this work is illustrated in Figure 6.



(a) (b)

Figure 6: a) Architecture of the entire model. b) There are linear layers, activation functions and one batch normalization layer inside one GINConv block.

The following is the MSE across batches plot obtained with this architecture.

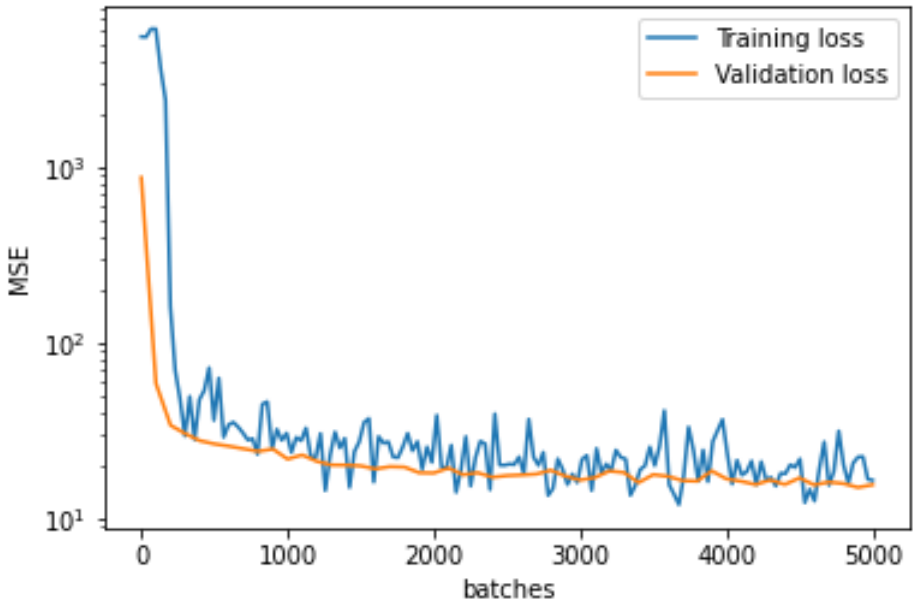


Figure 7: MSE vs. batches for GINConv

2.5 Changing the Dataset

The first dataset used was limited to the most energetic 128 hits in the cells. Then a dataset that did not have such limitation was reproduced, meaning a great increase in the size of the graph.

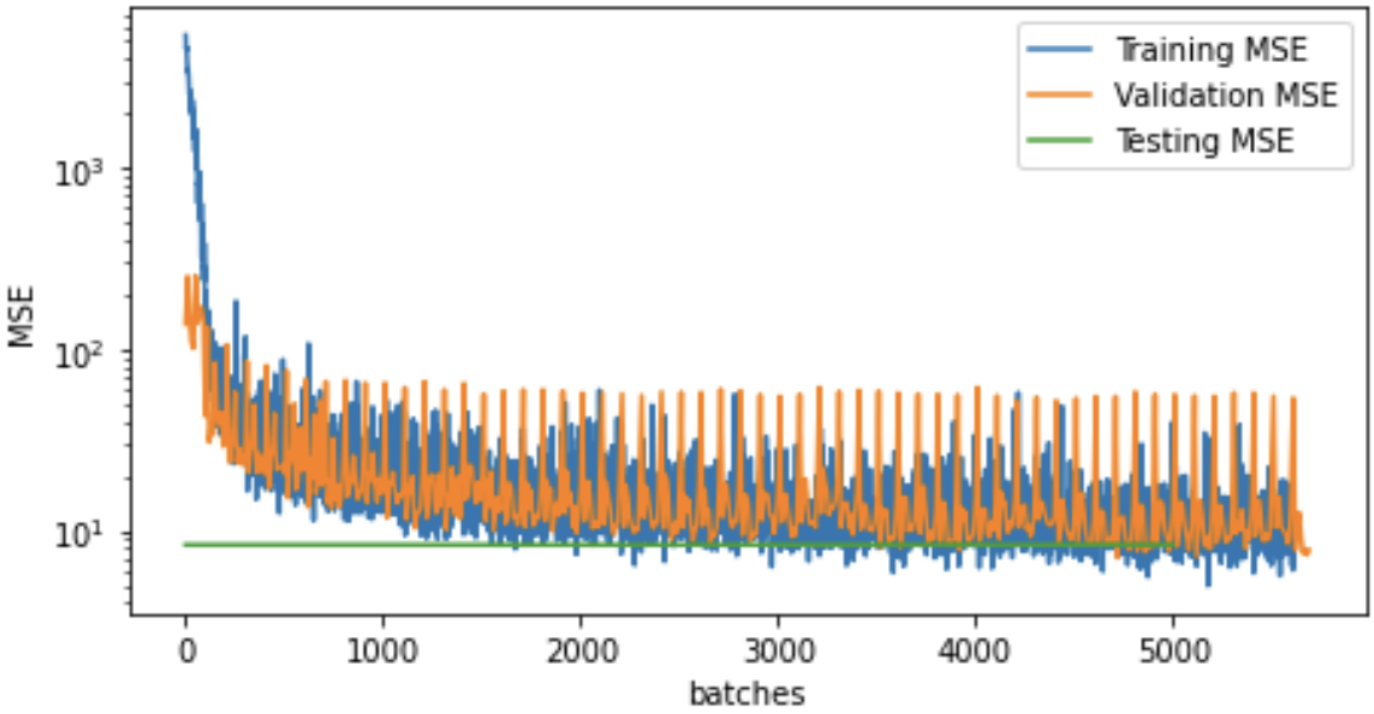


Figure 8: MSE vs. batches for GINConv with the full dataset

2.6 Memory Efficiency

In order to collect messages from the neighboring nodes, the MessagePassing interface of PyG uses a gather-scatter procedure. Under the hood, the MessagePassing implementation produces a code that looks as follows:

```python
from torch_scatter import scatter

x = ...           # Node features of shape [num_nodes, num_features]
edge_index = ...  # Edge indices of shape [2, num_edges]

x_j = x[edge_index[0]]  # Source node features [num_edges, num_features]
x_i = x[edge_index[1]]  # Target node features [num_edges, num_features]

msg = MLP(x_j - x_i)  # Compute message for each edge

# Aggregate messages based on target node indices
out = scatter(msg, edge_index[1], dim=0, dim_size=x.size(0), reduce="add")
```

Figure 9: Gather-scatter scheme in PyTorch

It has the disadvantage of explicitly materializing the source and target node features, x_j and x_i, resulting in a high memory footprint on large and dense graphs [4]. Because of this, the increase in the size of the graph mentioned in the previous section was costly in terms of the GPU memory in PyTorch and the models failed to converge. However, GINConv is one of the GNN implementations that does not need an explicit materialization of x_j and x_i. Namely, the GINConv layer

$$\mathbf{x}_i' = \text{MLP}\left((1+\epsilon)\cdot\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)}\mathbf{x}_j\right),$$

can also be implemented as:

$$\mathbf{X}' = \text{MLP}\left((1+\epsilon)\cdot\mathbf{X} + \mathbf{A}\mathbf{X}\right),$$

where $\mathbf{A}$ denotes a sparse adjacency matrix of shape `[num_nodes, num_nodes]`. This formulation allows to leverage dedicated and fast sparse-matrix multiplication implementations. So, the class `SparseTensor` was introduced in PyG $\geq$ 1.6.0 which implements fast forward and backward passes for sparse-matrix multiplication. So instead of inputting the edge index matrix, `edge_index` is transformed into the `SparseTensor` format using the `ToSparseTensor` method.

```
DataBatch(x=[27428, 4], edge_index=[2, 87958], y=[50], batch=[27428], ptr=[51])
```

(a)

```
DataBatch(x=[27656, 4], y=[50], batch=[27656], ptr=[51], adj_t=[27656, 27656, nnz=89496])
```

(b)

Figure 10: a) DataBatch instance with edge index. b) DataBatch instance with sparse tensor transform

And inside the `forward` function, `SparseTensor` format is given as an input.

```
def forward(self, batch):
    x = self.convs[0](batch.x, batch.edge_index)
    x = self.convs[1](x, batch.edge_index)
```

(a)

```
def forward(self, batch):
    x = self.convs[0](batch.x, batch.adj_t)
    x = self.convs[1](x, batch.adj_t)
```

(b)

Figure 11: a) Edge index tensor is given as input to the layers. b) Edge index transformed into the sparse tensor format is given as input to the layers.

The predicted outcomes are not seen when the maximum memory allocated in PyTorch GPU is observed during the increasing of the inner dimensions of the GINConv model (Figure 11). This stays as an unresolved issue due to time constraints of the project.

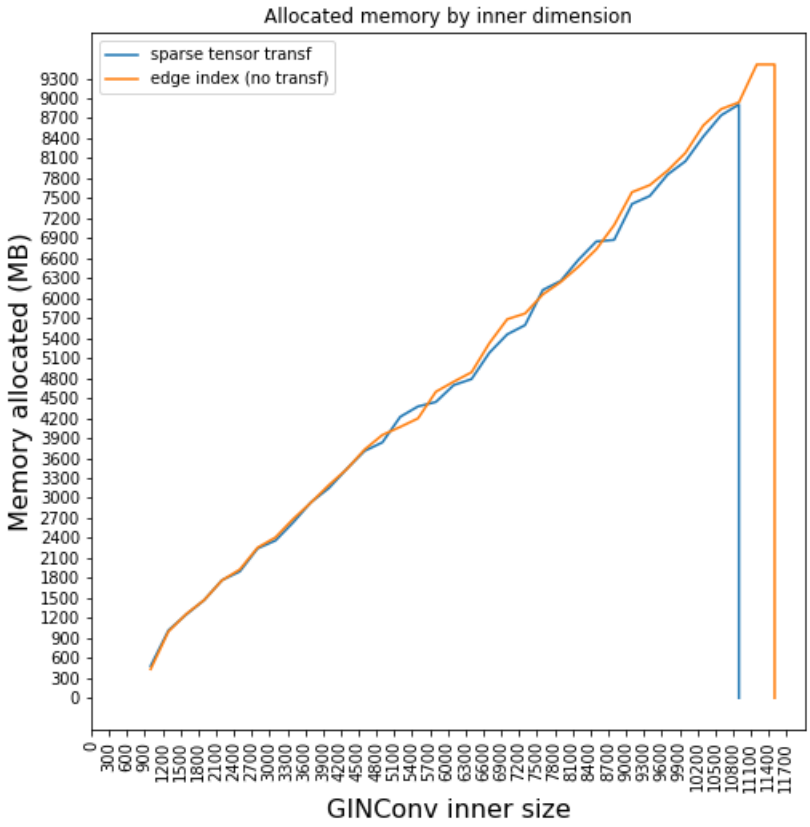


Figure 12: Memory Allocation in the PyTorch GPU by the inner dimensions in GINConv layers

3 Conclusion

After all the modifications in the code, the final results for model comparison are depicted in Figure 13.

	LinReg with limited dataset	GCNConv with limited dataset	GINConv with limited dataset	GINConv with the full dataset
Testing loss (MSE)	4756.43	30.54	16.88	8.3
Best training loss (MSE)	2540.31	18.52	8.23	12.05
Best validation loss (MSE)	2842.54	28.72	14.8	7.8

Figure 13: Comparison of the results

A few of the take-aways can be summarized as follow:

1. GNN's are promising for regression tasks.
2. GINConv performs better than GCNConv in any case, especially for larger graphs.
3. Using larger graphs increases the regression performance but is costly in terms of memory allocation for large inner dimensions.
4. Memory efficiency is a problem to be resolved.

References

[1] CMS Collaboration, CMS TDR-019

[2] PyTorch Geometric (PyG) Documentation

[3] Keyulu Xu, Weihua Hu, Jure Leskovec, Stefanie Jegelka. How Powerful Are Graph Neural Networks? 2019

[4] PyTorch Geometric (PyG) Documentation