Politecnico Di Milano

School of Industrial and Information Engineering

Advanced Computer Architecture

Project Report

GraphML D&D Dataset

Github Link

Student Name	Student ID
1. Hosein Mirhoseini	11046456
2. Ali Nabipour Dargah	11041035

Academic Year : 2024-2025

1 Introduction

In recent years, the use of graph-based data has become increasingly important in various fields, including biology, chemistry, social sciences, and beyond. These types of data often represent complex relationships and structures that are difficult to capture using traditional approaches. To address this, machine learning techniques that are specifically designed for graphs have emerged, offering new possibilities for understanding and making predictions from such data.

In this project, we focus on applying different graph-based machine learning methods to a real-world dataset. Our goal is to train models that can accurately classify data represented in graph form. However, our evaluation does not stop at accuracy alone. We are also interested in how practical these models are when it comes to resource usage, such as memory consumption, training time, and inference speed.

This perspective is especially important in situations where computational resources are limited or when the models are intended to run on edge devices or within time-sensitive environments. Therefore, we compare multiple approaches not only based on their performance in prediction but also on their efficiency and hardware impact. By doing so, we aim to find a balance between effectiveness and feasibility in real-world applications.

2 Dataset

The dataset used in this project is a well-known benchmark collection of graphs used for classification tasks. Each graph in the dataset represents a protein structure, where nodes correspond to secondary structure elements and edges reflect neighborhood relationships in the 3D space. This structure allows the data to naturally be represented in graph form, which makes it suitable for graph-based machine learning models.

Property	**Value**
Number of graphs	1113
Number of features per node	3
Number of classes	2

Table 1: Summary of the PROTEINS dataset

To provide a visual understanding of the data, Figure 1 shows an example graph from the dataset. Each node represents a component of a protein, and edges indicate structural relationships.

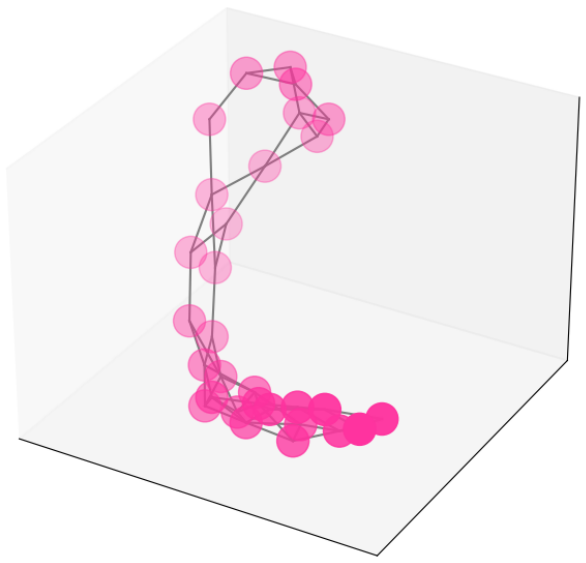

Figure 1: Example of a protein graph from the dataset

3 Proposed Models

In this project, we explore and compare three different graph-based models, each with distinct architectural principles. Our goal is to evaluate their performance on the dataset from multiple perspectives, not only in terms of classification accuracy but also with regard to hardware efficiency.

3.1 Hierarchical Graph Pooling with Structure Learning (HGP-SL)

Hierarchical Graph Pooling with Structure Learning (HGP-SL) [1] is a graph neural network architecture designed to learn rich, low-dimensional representations for entire graphs. It integrates hierarchical pooling with a structure learning mechanism, allowing the model to capture both node-level features and the overall graph topology in a coherent way.

The model operates in two primary stages within each pooling layer. First, a graph pooling operation selects a subset of informative nodes using learnable scoring functions, reducing the graph size while retaining critical structural and semantic information. This step effectively downsamples the graph based on both node features and the adjacency matrix.

Next, a structure learning layer reconstructs the graph connectivity among the retained nodes. It learns a new adjacency matrix that preserves key topological relationships, enabling the model to refine the graph structure dynamically. This component enhances the model's capacity to capture complex dependencies and adapt to varying graph shapes.

By stacking multiple HGP-SL layers, the model forms a hierarchical representation, progressively coarsening the graph and abstracting features at each level. The final representation is obtained by aggregating node embeddings from the top layer, which is then used for graph-level classification.

To provide a visual overview, Figure 2 illustrates the architecture of the HGP-

SL model.

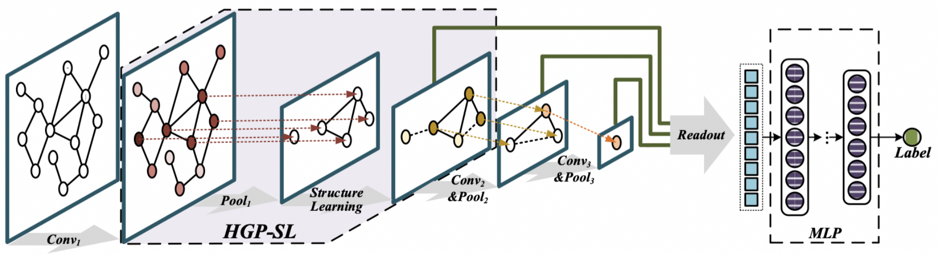

Figure 2: HGP-SL architecture

In our implementation, we closely adhere to the original design, leveraging both node features and graph structure in the pooling and structure learning modules. This enables the model to effectively capture both local interactions and global graph patterns, resulting in improved performance for graph classification tasks.

3.2 Enhanced Protein Graph Neural Network (EP-GNN)

The Enhanced Protein Graph Neural Network (EP-GNN) is a hybrid graph neural network architecture tailored for protein classification tasks. It is designed to combine both spectral and spatial learning mechanisms, thereby capturing a more comprehensive representation of the graph data.

The model begins with an initial node embedding layer, which maps raw node features into a higher-dimensional space using a fully connected transformation followed by batch normalization and a ReLU activation. This sets the stage for deeper graph processing.

EP-GNN employs a sequence of convolutional layers that alternate between different types of GNN operators. The first is a Chebyshev convolution (ChebConv), which captures spectral information from the graph Laplacian, allowing the model to learn smooth and expressive features over the graph structure. This is followed by an EdgeConv layer, which is designed to learn local geometric structures by considering relative positions of neighboring nodes. The combination of ChebConv and EdgeConv allows EP-GNN to learn both global and local patterns in protein graphs.

To further enhance stability and mitigate the problem of over-smoothing in deeper GNNs, the model incorporates PairNorm, a normalization technique that helps preserve variance in node representations across layers. Residual connections and dropout are used throughout to support efficient gradient flow and regularization.

A second ChebConv layer further refines the learned representations by deepening the spectral understanding of the graph. The model then performs a graph-level aggregation using a configurable readout function, such as mean, sum, or mean-max pooling, to condense node-level features into a single graph-level vector.

Finally, a multi-layer feedforward classifier processes the aggregated graph embeddings to produce class predictions. The classifier uses batch normalization, dropout, and non-linear activations to stabilize training and improve generalization.

The architectural overview of EP-GNN is illustrated in Figure 3.

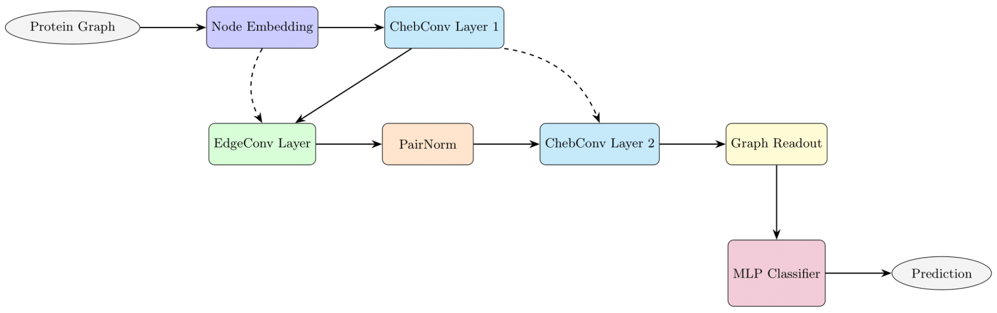

Figure 3: EP-GNN architecture

EP-GNN's design aims to capture complex protein structures by leveraging both spectral and spatial cues while maintaining a balance between expressiveness and computational efficiency. This makes it a strong candidate for graph-level classification tasks in protein analysis.

3.3 ProteinGraphTransformer: Transformer-Based Graph Neural Network

Our third proposed model, **ProteinGraphTransformer**, leverages attention-based message passing using the `TransformerConv` layer from PyTorch Geometric. This architecture is inspired by the success of the Transformer framework in sequence modeling, adapted here to operate on graph-structured protein data. The core idea is to allow each node to dynamically attend to its neighbors based on learned attention scores, enabling the model to capture both local and long-range dependencies that are often crucial in protein structures.

In this model, the input protein graph is represented by node features (such as residue-level descriptors) and an edge index indicating topological connections. These features are passed through a stack of multiple `TransformerConv` layers. The first layer projects the input into a higher-dimensional hidden space and applies multi-head self-attention to compute context-aware node representations. Subsequent layers further refine these embeddings using additional attention heads, capturing richer interactions across the graph. The final `TransformerConv` layer reduces the number of heads to one, consolidating the learned features into a unified embedding space.

After each `TransformerConv` layer, the output is passed through a ReLU activation function followed by dropout regularization, which helps mitigate overfitting and improve generalization. Once all layers have been processed, the node-level features are aggregated to form a graph-level embedding using global mean pooling. This fixed-size representation captures the overall characteristics of the protein structure and is passed through a fully connected linear layer to perform classification.

4 Implementation Overview

The implementation leverages popular deep learning and graph processing libraries, including PyTorch, PyTorch Geometric, and related utilities for efficient GPU computation and graph data handling. The codebase is organized into modular components for model definition, data preprocessing, training, and evaluation, enabling flexible experimentation with various graph neural network architectures and pooling strategies for protein graph classification.

4.1 Evaluation Metrics

For all models, a consistent set of evaluation metrics is computed, including balanced accuracy, precision, recall, F1-score, number of epochs, total training time, training time per epoch, and inference latency. Of particular note is the measurement of **Peak Memory Load**, which is obtained by monitoring the maximum GPU memory allocated during training using PyTorch's built-in memory tracking utilities. All experiments are conducted on an NVIDIA RTX-3050 GPU with 4GB of memory, ensuring fair and consistent hardware conditions across different models.

This modular structure allows for systematic comparison of different GNN architectures and pooling strategies on the protein classification task, with standardized metric reporting for reproducibility and benchmarking.

5 Results

5.1 Raw Evaluation Metrics

Tables 2 and 3 report the raw evaluation metrics for all models before any normalization or score aggregation. These include classification performance, training details, memory usage, model size, and inference latency.

Model	Bal. Acc.	Acc.	Prec.	Recall	F1
EP_GNN	0.803	0.812	0.683	0.778	0.727
GT	0.755	0.777	0.641	0.694	0.667
HGP_SL	0.845	0.830	0.667	0.882	0.759

Table 2: Classification performance metrics of each model.

Model	T/Ep (s)	Mem (MB)	Params	Size (MB)	Latency (ms)
EP_GNN	1.145	235.317	306,370	1.176	0.972
GT	1.109	752.852	8,938,242	34.097	0.431
HGP_SL	0.255	925.521	75,458	0.288	0.241

Table 3: Model resource usage and complexity metrics.

5.2 Model Scoring Function

To compare multiple models across a variety of performance and efficiency metrics, we define a unified scoring function. This function aggregates five normalized metrics into a single scalar score that reflects both prediction quality and resource efficiency.

The selected metrics are:

- **Balanced Accuracy** (`BA`): higher is better
- **Training Time per Epoch** (`TTE`): lower is better
- **Peak Memory Usage (MB)** (`MEM`): lower is better
- **Model Size (MB)** (`SIZE`): lower is better
- **Inference Latency per Sample (ms)** (`LAT`): lower is better

Each metric is min-max normalized to the range $[0, 1]$ across all models. The overall score is then computed using a weighted sum of transformed values:

$$\text{Score} = w_{\text{BA}} \cdot \text{BA} + w_{\text{TTE}} \cdot \frac{1}{1 + \text{TTE}} + w_{\text{MEM}} \cdot \frac{1}{1 + \text{MEM}} + w_{\text{SIZE}} \cdot \frac{1}{1 + \text{SIZE}} + w_{\text{LAT}} \cdot \frac{1}{1 + \text{LAT}}$$

The use of the transformation $\frac{1}{1+x}$ for the efficiency-related metrics ensures that lower values are rewarded, while maintaining numerical stability and bounded outputs. This avoids negative scores and keeps the interpretation intuitive.

The weights used in our experiments are:

$$w_{\text{BA}} = 0.4, \quad w_{\text{TTE}} = 0.15, \quad w_{\text{MEM}} = 0.25, \quad w_{\text{SIZE}} = 0.10, \quad w_{\text{LAT}} = 0.10$$

These weights reflect the relative importance of each metric, with a higher penalty applied to memory usage. The final score is bounded and can be used to rank models for a balanced trade-off between performance and efficiency.

5.3 Final Model Scores

Table 4 presents the final computed scores for each model based on the scoring function described above.

Model	**Score**
HGP_SL	0.8750
EP_GNN	0.6855
GT	0.3488

Table 4: Final normalized scores of each model

5.4 DeepDive

5.4.1 Peak Memory Usage

Why HGP_SL has the highest memory usage:

1. **Dense Adjacency Matrix Operations**: The HGPSLPool layer creates full adjacency matrices:

```
adj = torch.zeros((x.size(0), x.size(0)),
    dtype=torch.float, device=x.device)
```

This creates O(N^2) memory usage where N is the number of nodes.

2. **Multi-level Pooling**: Maintains multiple graph representations simultaneously at different hierarchical levels.

3. **Batch Processing**: With batch_size=512 (much larger than others), memory consumption is amplified.

5.4.2 Training Time Per Epoch

Why HGP_SL trains fastest despite high memory usage:

1. **Simple Base Operations**: Uses standard GCN convolutions which are computationally efficient
2. **Parallelizable Operations**: Dense matrix operations are highly optimized on GPUs
3. **Large Batch Size**: Better GPU utilization with batch_size=512
4. **Fewer Complex Operations**: No spectral decompositions or multi-head attention

5.4.3 Latency Analysis:

1. **HGP_SL**: Fast inference because structure learning happens during training; inference uses pre-computed hierarchical representations
2. **Graph Transformer**: Attention mechanisms require matrix multiplications but are still relatively efficient
3. **EP_GNN**: Multiple convolution types (Chebyshev + Edge) and complex feature transformations slow down inference

5.4.4 HGP_SL's Efficiency Paradox:

Despite highest memory usage, HGP_SL achieves:

- **Best performance** (0.845 balanced accuracy)
- **Fastest training** (0.255s/epoch)
- **Fastest inference** (0.241ms)

This is because:

1. **Memory vs. Computation Trade-off**: Uses memory to precompute and store structures, reducing computation
2. **Hierarchical Efficiency**: Multi-level pooling captures important features quickly
3. **Structure Learning**: Adapts graph topology to the task, improving both speed and accuracy

5.4.5 Summary

The performance metrics reflect fundamental architectural differences:

- **EP_GNN**: Balanced approach with moderate resource usage and good performance
- **Graph Transformer**: Parameter-heavy with high memory needs but underwhelming results
- **HGP_SL**: Memory-intensive but computationally efficient, achieving the best speed-accuracy trade-off through intelligent structure learning and hierarchical processing

The key insight is that **memory usage doesn't always correlate with computational complexity**. HGP_SL's high memory usage enables efficient computation patterns, while the Graph Transformer's high parameter count creates computational bottlenecks without corresponding performance benefits.

6 Conclusion

In this project, we evaluated three different models `EP_GNN`, `GT`, and `HGP_SL` on the PROTEINS graph classification dataset. Each model was assessed not only in terms of classification performance (such as accuracy, precision, recall, and F1 score), but also through system-level metrics including training time, memory usage, model size, and inference latency.

Given the diversity of these metrics, we proposed a weighted scoring function to enable holistic comparison of the models. This function balances predictive accuracy with efficiency indicators, ensuring that both performance and resource demands are fairly considered.

Among the tested models, `HGP_SL` achieved the highest balanced accuracy (0.845) and overall score, confirming its strong classification capability. However, this came at the cost of significantly higher peak memory usage (925.5 MB), which is nearly four times greater than that of `EP_GNN` (235.3 MB). On the other hand, `EP_GNN` offered competitive accuracy (0.803) with much lower memory consumption and model size, making it a more resource-efficient alternative.

These results highlight the importance of evaluating models beyond accuracy alone, especially in practical scenarios where memory and latency constraints are critical. Our scoring methodology facilitates such a comprehensive evaluation, enabling better-informed model selection for graph-based learning tasks.

References

[1] Zhen Zhang, Jiajun Bu, Martin Ester, Jianfeng Zhang, Chengwei Yao, Zhi Yu, and Can Wang. Hierarchical graph pooling with structure learning. *arXiv preprint arXiv:1911.05954*, 2019.