Assignment 2: Graph Attention Network (GAT) from scratch

ELTE University

by

Jacopo Manenti

Student Name	Student Number
Jacopo Manenti	G1R5RZ

Professor: Guettala Walid
Faculty: Computer Science, Budapest

0.1. Abstract

In this paper, we detail the implementation of a Graph Attention Network (GAT) layer from scratch, using only PyTorch and intentionally avoiding any pre-existing implementations from libraries such as PyTorch Geometric. To enhance the performance and flexibility of the GAT layer, we incorporated several advanced features, including batch normalization, residual connections, and multi-head attention, both within a single multi-head GAT layer and across multiple layers in stacked configurations (Chapter 0.4).

Our model was designed to perform a multi-task regression, predicting 14 distinct molecular properties for each graph in the dataset (QM7b), calculated through quantum mechanical methods at various levels of theory. These properties include energies, polarizabilities, and orbital information.

Since the QM7b dataset lacks inherent node features (Chapter 0.2), we initially used randomly generated node features as a baseline. To improve model performance, we then crafted custom node features that capture structural properties of the graph, including: *Node degree*, *Degree Centrality*, *Local Clustering Coefficient*, *Eccentricity*. Additionally, we incorporated the edge features provided by the QM7b dataset to further enrich the input to the model.

We tested various configurations of the GAT model, ultimately selecting the best-performing model for further analysis (as described in Chapter 0.5). To examine the effects of layer depth on model performance, we implemented multi-layer GATs with a number of layers exceeding the graph's diameter to observe if the model experienced over-smoothing. The findings, detailed in Chapter 0.7, revealed that the optimal configuration was achieved when the number of layers was set equal to the graph's diameter, d, as this balanced complexity with performance.

0.2. Dataset

The QM7b dataset [2] extends the QM7 dataset to provide a richer collection of quantum mechanical properties for small organic molecules. It includes 7,211 molecules (graphs), each with up to seven heavy atoms—primarily carbon, nitrogen, oxygen, and sulfur. Unlike QM7, QM7b offers a comprehensive set of 14 molecular properties, making it a versatile resource for multi-task learning in quantum chemistry.

In QM7b, molecules have an average graph diameter of approximately 1, represented as an undirected graph structure with no initial node features, making feature engineering essential for meaningful input data. The graph is characterized by an `edge index` of shape $[2, 1.766.366]$, `edge attr` of shape $[1.766.366]$, and target labels `y` of shape $[7.211, 14]$, with 111.180 `total nodes` across the dataset.

The dataset supports 14 distinct regression tasks, each corresponding to a separate molecular property. These properties, computed using quantum mechanical methods, span various levels of theory such as *HOMO energy* (Highest Occupied Molecular Orbital), *LUMO energy* (Lowest Unoccupied Molecular Orbital), ecc...

Predicting these 14 quantum properties has substantial practical value in materials science, drug discovery, and molecular design. By accurately modeling these properties, multi-task models trained on QM7b can provide rapid, reliable predictions for new molecules, guiding experimental work and minimizing reliance on costly laboratory measurements and computationally intensive simulations.

Note: The data in the dataset has been normalized using `NormalizeFeatures` *when imported from the PyTorch dataset, meaning that each feature has been scaled to improve model performance and stability.*

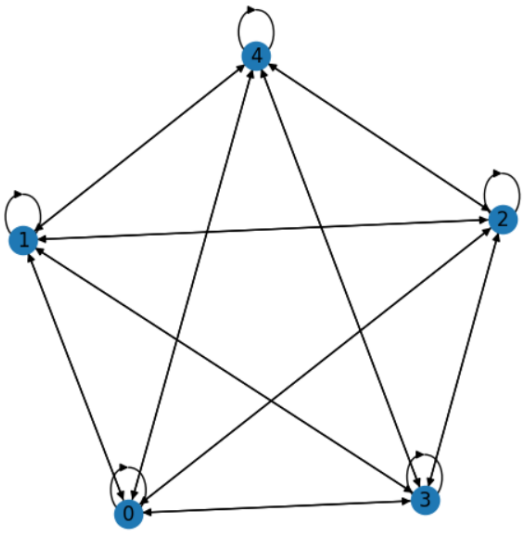


Figure 1: A visual representation of one graph (atom) in QM7b dataset

0.3. GAT Implementation

This **GATLayer** implementation is designed for a Graph Attention Network (GAT) layer in PyTorch, handling two different types of input representations: an adjacency matrix ($\mathbb{R}^{nxn}$) and an edge list matrix ($\mathbb{R}^{2x\text{number of edges}}$). This flexibility allows the model to adapt based on how the input graph is structured during training.

The GAT (Graph Attention Network) layer begins by initializing key elements and defining the forward pass. The components are initialized as follows:

- **Input and Output Features:** Let F_{in} be the input feature dimension and F_{out} be the output feature dimension after transformation.
- **Dropout and LeakyReLU Activation:** Dropout, denoted as p, is used to prevent overfitting. A LeakyReLU activation function with negative slope α is used in the attention mechanism to introduce non-linearity.
- **Weight Matrices:**
  - **W** $\in \mathbb{R}^{F_{\text{in}} \times F_{\text{out}}}$: This learnable weight matrix projects node features from F_{in} to F_{out}.
  - **a** $\in \mathbb{R}^{2F_{\text{hidden}}}$: This attention vector is initialized using Xavier initialization to ensure stable training (as stated in the original paper[1]), and it computes attention scores between node pairs.
- **Residual Projection:** A residual projection layer is used to ensure that the input features match the output dimensions if $F_{\text{in}} \neq F_{\text{out}}$.

The forward pass of the GAT layer computes the transformed and attention-weighted node features. It is designed to take as input a set of node features $\mathbf{H} \in \mathbb{R}^{N \times F_{\text{in}}}$ and an adjacency matrix **A** representing the graph structure, where N is the number of nodes in a batch. The key steps are as follows:

- **Device Management:** All parameters are transferred to the appropriate device (e.g., GPU) to ensure compatibility with accelerated training.
- **Linear Transformation:** Each node feature $\mathbf{h}_i \in \mathbb{R}^{F_{\text{in}}}$ is linearly transformed:

$$\mathbf{h}'_i = \mathbf{W}\mathbf{h}_i,$$

where $\mathbf{h}'_i \in \mathbb{R}^{F_{\text{out}}}$.

- **Attention Mechanism:** For each pair of nodes (i, j), the attention score e_{ij} is computed as:

$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^\top [\mathbf{h}'_i \,\|\, \mathbf{h}'_j]\right),$$

  where $\|$ denotes concatenation.
- **Masked Attention:** Attention scores are masked using the adjacency matrix **A**, so only connected nodes are considered:

$$e_{ij} = \begin{cases} e_{ij}, & \text{if } (i,j) \in \mathcal{E}, \\ -\infty, & \text{otherwise}, \end{cases}$$

  where $\mathcal{E}$ is the edge set.
- **Softmax and Dropout:** The attention coefficients α_{ij} are computed with softmax normalization:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}(i)} \exp(e_{ik})},$$

  where $\mathcal{N}(i)$ represents the neighbors of i. Dropout is applied during training to regularize the attention scores.
- **Aggregation:** The final output features for each node i are computed as a weighted sum of its neighbors' transformed features:

$$\mathbf{h}_i^{\text{out}} = \sigma\left(\sum_{j \in \mathcal{N}(i)} \alpha_{ij} \mathbf{h}'_j\right),$$

  where σ is a non-linear activation function such as ELU.
- **Residual Connection (Optional):** A residual connection is added if desired, by projecting the input features to the output dimension and summing:

$$\mathbf{h}_i^{\text{out}} = \mathbf{h}_i^{\text{out}} + \mathbf{h}_i^{\text{res}},$$

  where $\mathbf{h}_i^{\text{res}} = \mathbf{W}_{\text{res}} \mathbf{h}_i$ if $F_{\text{in}} \neq F_{\text{out}}$.

This implementation supports both adjacency matrices and index matrices, making it adaptable depending on the input format:

- **Adjacency Matrix**: Directly uses 'adj' for masking attention scores, as shown in the 'Masked Attention' step.
- **Edge list Index Matrix**: For each batch, the edge list is used to convert each batch into into a NetworkX graph, G, and the adjacency matrix is created via `nx.to_numpy_array(G)`. This adjacency matrix (adj) is then converted to a PyTorch tensor.

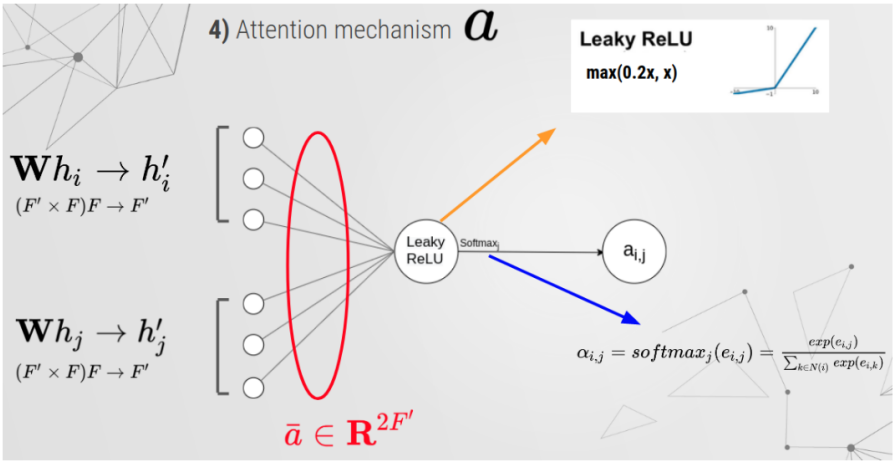


Figure 2: The image outlines an attention mechanism for GAT layer, which uses the Leaky ReLU activation function within the process.

0.4. Experiments

0.4.1. MultiHeadGAT

The `MultiHeadGAT` is a variant of the Graph Attention Network (GAT) that utilizes multiple attention heads. Each attention head performs a separate attention mechanism on the same input (with different weights initialization), and their outputs are either concatenated or averaged to form a richer node representation. This approach allows the model to capture multiple aspects of the graph structure and can be particularly useful for graph classification. To perfom graph classification, the multihead attentions' feature are then pooled, producing an output of dimension $\mathbf{H} \in \mathbb{R}^{F_{\text{number of graphs in a batch}} \times F_{\text{regression task}}}$ for each batch.

The `MultiHeadGat` class is implemented as a neural network module in PyTorch and is designed to work with graph data represented as node features and adjacency matrices. The architecture can be broken down into several key components:

Initialization:
The model takes the following hyperparameters:

- `num_heads`: The number of attention heads used in the multi-head attention mechanism.
- `in_features`: The input feature dimension for each node in the graph.
- `hidden_features`: The number of features each node will have after the attention mechanism.
- `out_features`: The number of output features after applying the final linear transformation, i.e. equal to the number of regression tasks.
- `dropout`: The dropout rate applied during training to prevent overfitting.
- `alpha`: The leaky ReLU parameter for the attention mechanism.
- `concat`: Whether to concatenate the outputs of the attention heads (`True`) or average them (`False`).
- `pooling_type`: The pooling type used for graph-level tasks: `'mean'` or `'max'`.
- `is_layerNorm`: Whether to apply Layer Normalization after each attention head.
- `residual`: Whether to apply residual connections in the attention heads.

The model consists of the following major components:

1. **Attention Heads**: A list of attention heads, each corresponding to an individual `GATLayer`. The attention heads are implemented as a `ModuleList`, and each head independently computes the attention mechanism.
2. **Head Transformation Layer**: If `concat` is `True`, the outputs of the attention heads are concatenated, and a linear transformation is applied to map the combined features to the desired dimensionality.
3. **Layer Normalization**: If `is_layerNorm` is `True`, Layer Normalization is applied after each attention head to stabilize training and ensure consistent output distributions.
4. **Final Classifier**: The `self.classifier` layer is a final linear layer that maps the combined node representations to the output space. For node classification tasks, this maps to the number of regression tasks.

Forward Pass
The forward pass of the `MultiHeadGat` class can be broken down into the following steps:

1. **Attention Computation**: Each attention head computes the attention mechanism independently. The attention mechanism uses the node features and the adjacency matrix to learn how nodes should aggregate information from their neighbors.
2. **Layer Normalization (optional)**: If `is_layerNorm` is `True`, each output from the attention heads is normalized using Layer Normalization.
3. **Concatenation or Averaging**: Depending on the value of `concat`, the outputs of the attention heads are either:

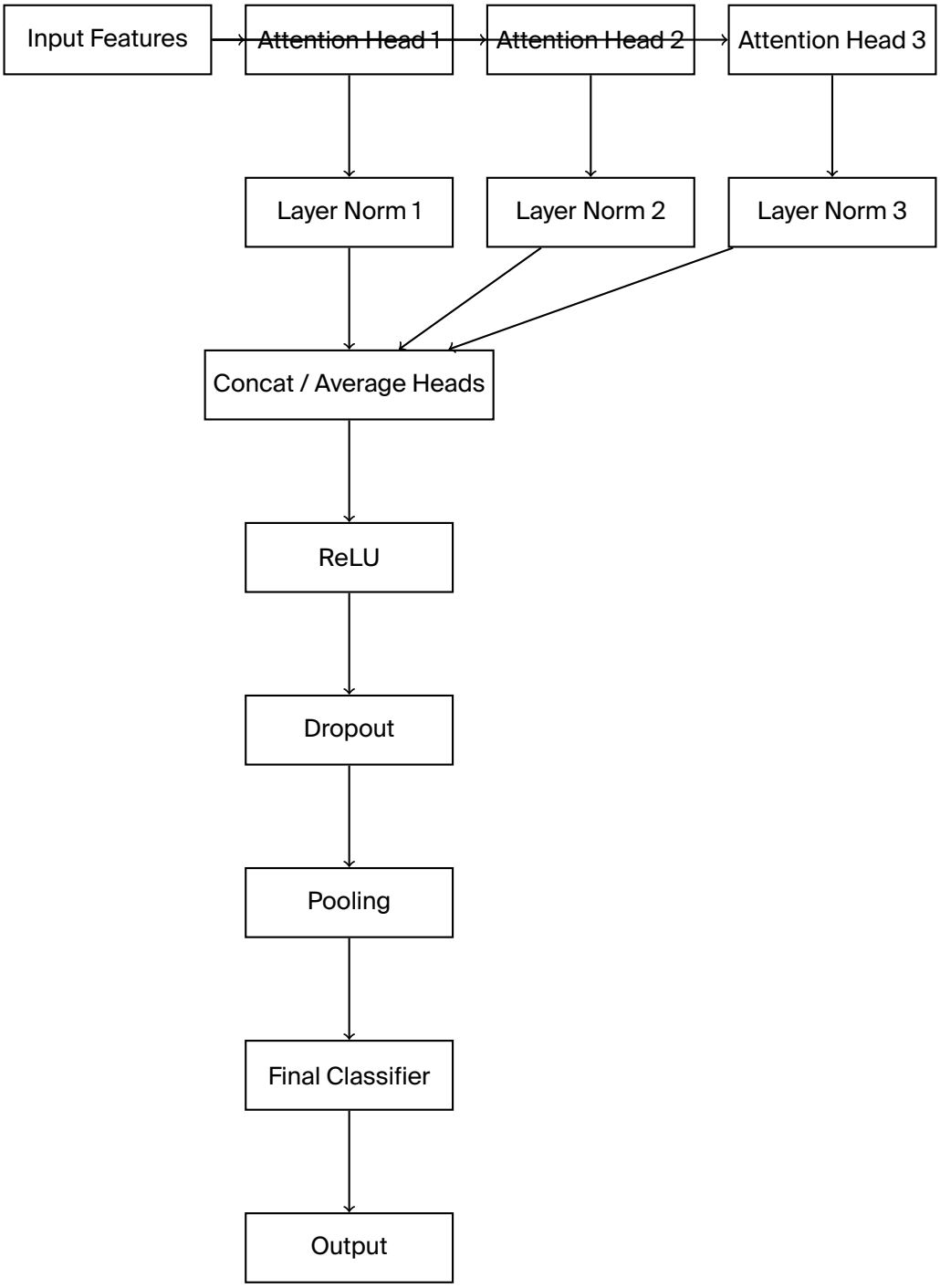
- **Concatenated**: If `concat=True`, the outputs from all attention heads are concatenated along the feature dimension, producing a tensor of shape $[N, \texttt{num\_heads} \times \texttt{hidden\_features}]$, where N is the number of nodes. The concatenated output is then passed through a linear transformation, producing an putput of $[N, \texttt{hidden\_features}]$
- **Averaged**: If `concat=False`, the outputs from all attention heads are averaged, producing a tensor of shape $[N, \texttt{hidden\_features}]$.

4. **Pooling (optional)**: If the model is applied to graph-level tasks and the `batch` variable[1] is provided, global pooling is applied to aggregate node features into graph-level representations, producing an output of $[\texttt{\#graphs in a batch}, \texttt{hidden\_features}]$. The two pooling types supported are:

   - `global_mean_pool`: Averages the node features within each graph.
   - `global_max_pool`: Takes the maximum node feature value within each graph.

5. **Final Classifier**: The aggregated node features (or graph features) are passed through a final linear layer `self.classifier` to produce the output of the model. The output shape will be $[\texttt{num\_graphs}, \texttt{out\_features}]$ for graph classification.

[1] A mask indicating which node belongs to which graph in a batch



0.4.2. GAT with edge attributes

The `GATLayer2` is a variant of the standard Graph Attention Network (GAT) that incorporates edge features into the attention mechanism. The attention input is expanded to include both node features h_i, h_j and the edge features e_{ij}, which allows the model to learn attention scores that account for both the nodes' characteristics and their relationships (as represented by the edge features). This change improves the model's ability to capture more complex graph structures and enhances its expressiveness in tasks where edge attributes are crucial.

Specifically, the following changes are made:

- **Edge Features in Attention Calculation:** In the standard GAT, the attention mechanism uses only the node features h_i and h_j (features of node i and node j) to compute the atten-

tion scores. In `GATLayer2`, edge features e_{ij} are incorporated into the attention calculation by concatenating the node features with the edge features:

$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^T \cdot [h_i || h_j || e_{ij}]\right)$$

where $||$ denotes concatenation of the node features h_i, h_j, and the edge feature e_{ij}.

- **Edge Features in Attention Calculation:** The edge features are mapped into the attention mechanism using the adjacency matrix. Specifically, the non-zero elements of the adjacency matrix indicate the presence of edges, and these positions are filled with the corresponding edge features.
- **Edge Feature Masking:** The adjacency matrix is used to mask the attention scores, ensuring that only the valid edges contribute to the attention mechanism. This way, only the edges with non-zero adjacency values will have their corresponding attention scores computed.

Advantages of Including Edge Features
For the QM7b dataset, which involves molecular graphs with nodes representing atoms and edges representing bonds between atoms, incorporating edge features offers the following specific advantages:

- **Incorporation of Bond Information:** In molecular graphs, edge features represent the type and strength of bonds between atoms. By including edge features, the model can better capture the chemical properties and interactions between atoms, leading to more accurate predictions of molecular properties.
- **Better Capture of Molecular Structure:** Including edge features allows the model to better understand the relationships between atoms based on the connectivity and types of bonds, leading to a more comprehensive understanding of the molecular graph's structure.

0.4.3. GAT with Handcrafted features

In addition to including edge features, we also explored the manual crafting of node features based on graph properties. This approach enhances the model's ability to understand the structure of the graph and provides additional information for learning. The following graph-level features are computed and added to each node in the graph:

- **Degree of Nodes:** The degree of a node represents the number of edges connected to it.
- **Local Clustering Coefficient:** The local clustering coefficient reflects the tendency of a node's neighbors to be connected to each other. It helps capture the local cohesiveness of the graph's structure.
- **Degree Centrality:** This metric measures the relative importance of a node based on its degree. Nodes with higher degree centrality are more connected and can have more influence within the graph.
- **Eccentricity:** Eccentricity measures the maximum distance from a node to all other nodes in the graph, providing insight into the centrality of the node in the overall graph structure. For disconnected graphs, a default value (0) is used.

Implementation Workflow
The workflow for computing these handcrafted features is as follows:

1. **Graph Masking and Edge Indexing:** For each graph in the batch, we create a mask to identify the nodes that belong to the current graph. The edge indices are filtered based on this mask, ensuring that we only consider edges within the current graph.
2. **NetworkX Conversion:** After filtering the edges, the graph is converted to a `NetworkX` graph, enabling the use of its graph algorithms to compute the desired features.
3. **Feature Extraction:** The following features are computed for each node:
    - **Degree:** Calculated using `G.degree()`.
    - **Clustering Coefficient:** Calculated using `nx.clustering()`.

- **Degree Centrality:** Calculated using `nx.degree_centrality()`.
- **Eccentricity:** Calculated using `nx.eccentricity()`, with error handling for disconnected graphs.

4. **Feature Aggregation:** The computed features are aggregated across all graphs in the batch and stored in separate lists. These lists are then converted to tensors.
5. **Concatenation of Features:** The new handcrafted features are concatenated with the existing node features in the `graph_batch`.

0.4.4. DeepMultiHeadGAT

We also tested a `DeepMultiHeadGAT`, which consists of two main components: the `MultiHeadGat` and `GATLayer` classes. This architecture is designed to utilize multi-head attention mechanisms with edge features, nodes features and additional layers ($l > d$) of attention-based learning.

Constructor Parameters

The constructor for `MultiLayerGat` accepts the following parameters:

- **num_layers:** The number of layers in the model, each containing a multi-head GAT attention mechanism.
- **num_heads:** The number of attention heads used in each multi-head GAT layer.
- **in_features:** The number of input features per node.
- **hidden_features:** The number of hidden features (i.e., the number of output features from each GAT layer).
- **out_features:** The number of regression task for the final layer.
- **dropout:** Dropout probability for regularization.
- **alpha:** The negative slope of the LeakyReLU activation used in the attention mechanism.
- **concat:** A boolean list specifying whether to concatenate the outputs of the attention heads in each layer.
- **pooling_type:** The type of pooling operation used for graph-level aggregation, either `'mean'` or `'max'`.
- **is_layerNorm:** A boolean indicating whether to apply layer normalization to the outputs of each attention head.
- **residual:** A boolean indicating whether to include residual connections in the multi-head attention layers.
- **concat_res:** A boolean that determines whether to concatenate the graph-level pooled representations from each layer, or average them.

Model Architecture

The architecture of the `MultiLayerGat` model consists of the following components:

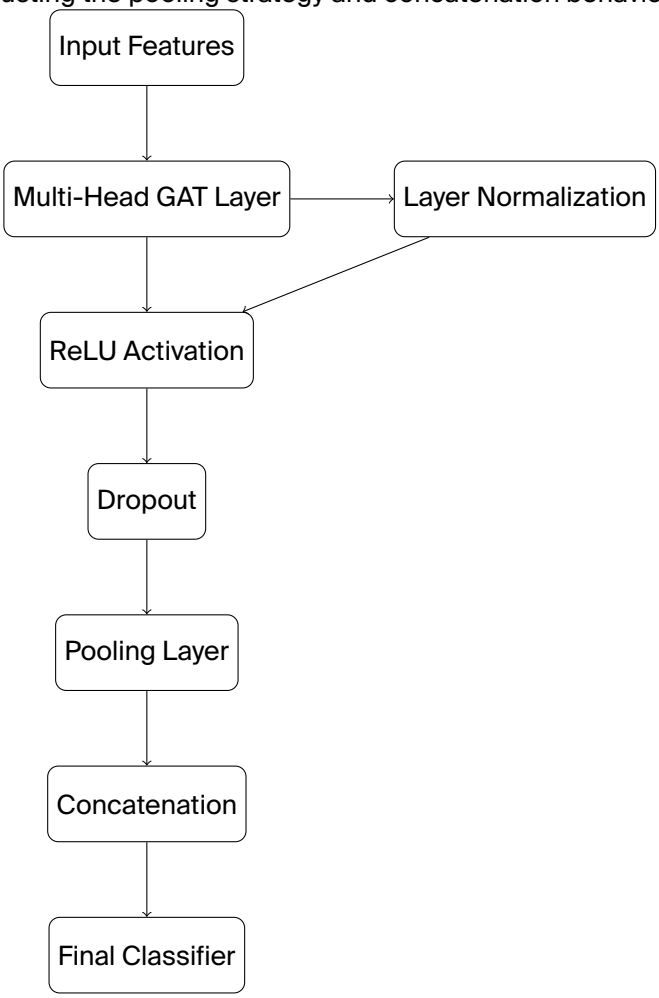
- **Multi-head Attention Layers:** The model uses multiple layers of multi-head GAT, each consisting of several attention heads. Each layer computes node features based on the graph structure (node features and edge attributes). The outputs of the attention heads can be either concatenated or averaged, depending on the configuration.
- **Pooling Layer:** After the attention computation in each layer, the node-level representations are aggregated to form a graph-level representation. This aggregation is performed using either `global_mean_pool` or `global_max_pool`, depending on the specified pooling type. The choice of pooling method allows the model to capture different aspects of the graph structure.
- **Layer Normalization:** If `is_layerNorm` is set to `True`, the outputs of each attention head are normalized using layer normalization. This helps stabilize the training process and improves convergence.
- **Residual Connections:** If `residual` is `True`, residual connections are introduced in the multi-head GAT layers, where the input to the layer is added to the output of the attention mechanism. This enables better gradient flow and allows the model to learn more complex representations.

- **Final Classifier:** After the graph-level representations are computed and aggregated across layers, they are passed through a final linear classifier that outputs the predicted class for the graph. The input to the classifier can either be the concatenated or averaged representations from all layers, based on the `concat_res` flag.

Forward Pass
In the forward pass, the following steps occur:

1. The node features, adjacency matrix, and edge attributes are passed through each layer of multi-head GAT attention.
2. After computing the attention outputs, the node-level features are pooled into a graph-level representation using the specified pooling method.
3. If the model contains multiple layers, the graph-level representations from each layer are either concatenated or averaged, based on the `concat_res` parameter.
4. The resulting aggregated graph-level representation is passed through the final classifier to obtain the output predictions.

The model is flexible and can handle various graph-based tasks, such as graph classification, by adjusting the pooling strategy and concatenation behavior across layers.



0.5. Evaluate the models

In this part of the project, we evaluate five different models with varying configurations of features and model architectures. Each model is evaluated over 10 epochs using the following settings:

Model Configurations:

1. **Model 1:** MultiHeadGat with random node features, concatenation of attention heads, and mean pooling, with no layer normalization and no residual connections.

```
model1 = MultiHeadGat(num_heads=3, in_features=3, hidden_features=64, out_features=14,
    dropout=0.6, alpha=0.2,
                      concat=True, pooling_type='mean', is_layerNorm=False, residual=False)
```

2. **Model 2:** MultiHeadGat with random node features, concatenation of attention heads, mean pooling, no layer normalization, and residual connections.

```
model2 = MultiHeadGat(num_heads=3, in_features=3, hidden_features=64, out_features=14,
    dropout=0.6, alpha=0.2,
                      concat=True, pooling_type='mean', is_layerNorm=False, residual=True)
```

3. **Model 3:** MultiHeadGat with random node features, concatenation of attention heads, mean pooling, layer normalization, and residual connections.

```
model3 = MultiHeadGat(num_heads=3, in_features=3, hidden_features=64, out_features=14,
    dropout=0.6, alpha=0.2,
                      concat=True, pooling_type='mean', is_layerNorm=True, residual=True)
```

4. **Model 4:** MultiHeadGat with random node features and edge features, concatenation of attention heads, mean pooling, layer normalization, and residual connections.

```
model4 = MultiHeadGat2(num_heads=3, in_features=3, hidden_features=64, out_features=14,
    dropout=0.6, alpha=0.2,
                       concat=True, pooling_type='mean', is_layerNorm=True, residual=True)
```

5. **Model 5:** MultiHeadGat with handcrafted node features and edge features, concatenation of attention heads, mean pooling, layer normalization, and residual connections.

```
model5 = MultiHeadGat2(num_heads=3, in_features=4, hidden_features=64, out_features=14,
    dropout=0.6, alpha=0.2,
                       concat=True, pooling_type='mean', is_layerNorm=True, residual=True)
```

Loss Functions and Evaluation Metrics:

- **Training Loss:** The criterion used for training the models is *Mean Squared Error (MSE)* loss, which is a standard loss function for regression tasks. The formula for MSE loss is:

$$\text{MSELoss}(y, \hat{y}) = \frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K}(y_{ij} - \hat{y}_{ij})^2$$

  where:

  - y_ij is the true value for the i-th sample of the j-th target value.
  - $\hat{y}_i$ is the predicted value for the i-th sample of the j-th target value.
  - N is the number of samples.
  - K is the number of regression tasks.

- **Testing Metric:** For testing, we use *Root Mean Squared Error (RMSE)* as the evaluation metric. RMSE is computed as:

$$\text{RMSE}(y, \hat{y}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{K}(y_{ij} - \hat{y}_{ij})^2}$$

  The reason RMSE is used for testing rather than MSE is that it gives a more direct interpretation of the model's prediction performance by reflecting the scale of the errors in the same units as the target variable. This is particularly useful when comparing models or assessing how well the predictions approximate real-world values.

- **Optimizer:** The optimizer used for training the models is *Adam*, a popular choice for deep learning models due to its adaptive learning rates. The learning rate and weight decay are set as follows:

$$\text{opt} = \text{torch.optim.Adam}(model.parameters(), lr = 0.01, \text{weight\_decay} = 5e-4)$$

- **Learning Rate (lr = 0.01):** This controls the step size at each iteration while moving toward a minimum of the loss function.
- **Weight Decay (wd = 5e-4):** This is used for regularization to prevent overfitting by penalizing large weights.

0.6. Results

Fig.3 shows the training loss for models defined in section 0.5 across 10 epochs. The loss values decrease over the epochs, indicating that the models are learning and improving their performance on the training data. The different models exhibit varying rates of loss reduction, with some models (e.g., model 0) showing a more rapid decrease in loss compared to others.

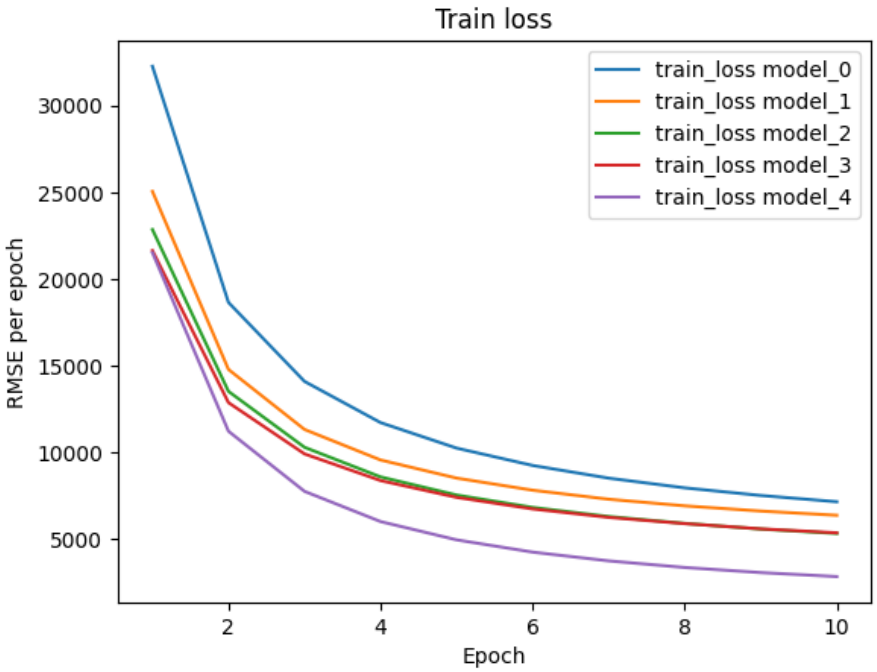


Figure 3: Training loss for the models. Model 5 seems to perfom better.

Fig.4 displays the RMSE (Root Mean Squared Error) values for the different models (1, 2, 3, 4, and 5) on a test set. The RMSE values provide a measure of the model's performance, with lower values indicating better performance. The results show that model 5 has the lowest RMSE, suggesting it is the best-performing model among the ones tested.

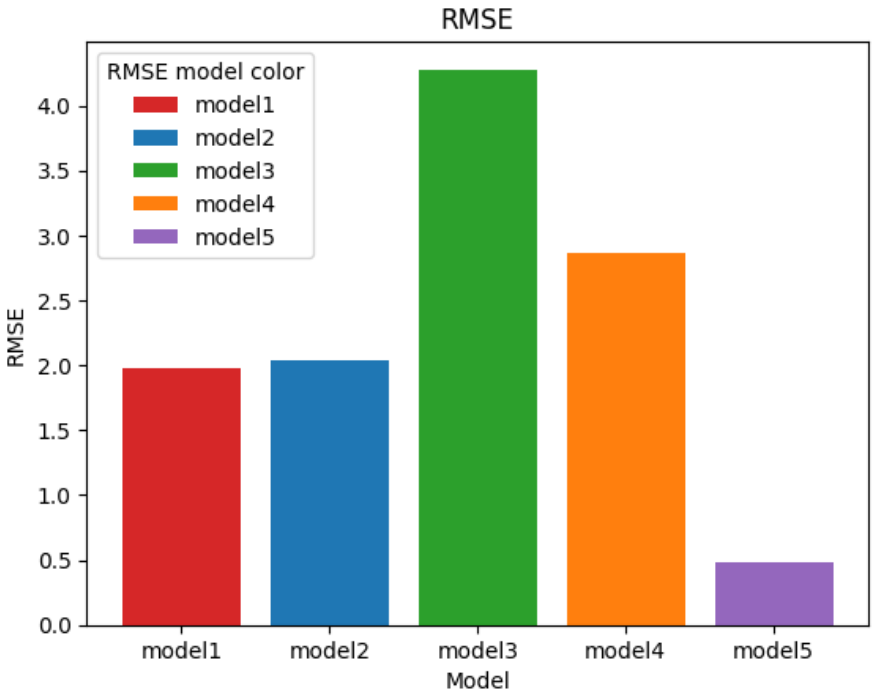


Figure 4: RMSE for each evaluated model. Model 5 has better performance.

Finally, Fig.5 shows the scatter plot between the predicted values by model 5 (x-axis) vs. the true values (y-axis) for various regression tasks (from 1 to 14). The scatter plots provide a visual representation of the model's performance, with the points closer to the diagonal line (y = x) indicating better predictions. For each of the 14 tasks, the scatter plot shows a tight clustering of points along the diagonal line, suggesting that the model is making accurate predictions for this task.

In conclusion, we can say that the model is making accurate predictions, with the points clustering closely around the diagonal line.

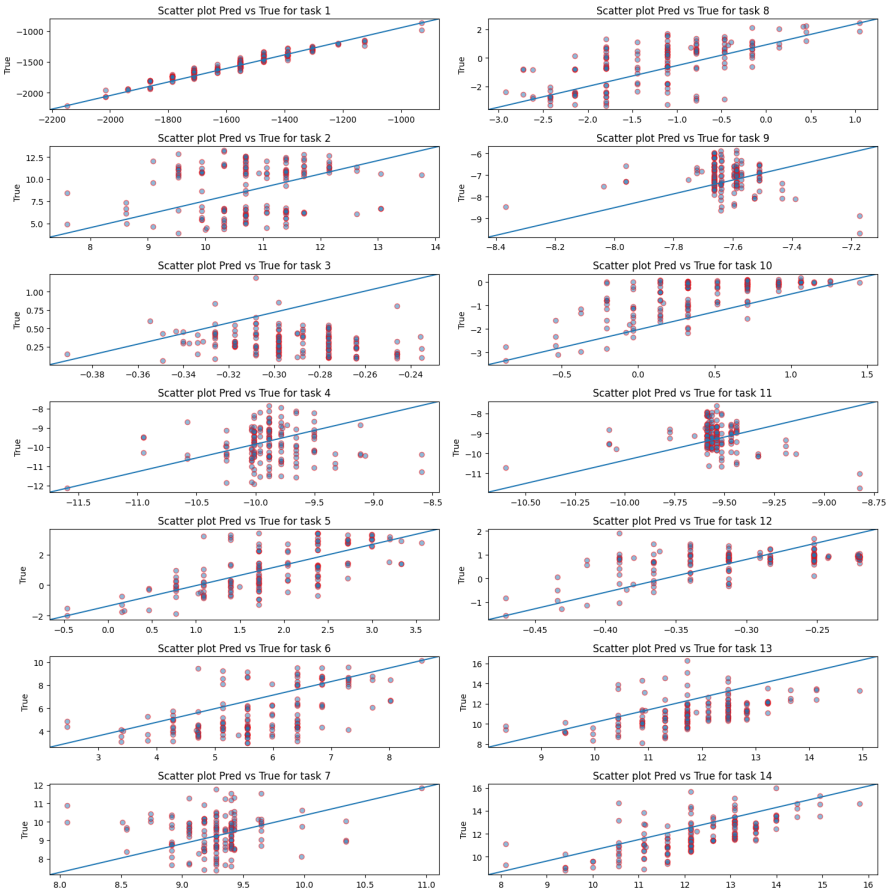


Figure 5: Model 5 scatterplot, each for each regression task

The choice of model 5 as the best-performing model was expected, as it incorporates several key components that are important for effective graph classification tasks.

The model 5 utilizes a Multi-Head Graph Attention (MultiHeadGAT) architecture, which is a powerful technique for capturing the complex relationships and dependencies within graph-structured data. The MultiHeadGAT mechanism allows the model to learn multiple attention patterns, capturing diverse sources of information from the graph.

Additionally, the model incorporates hand-crafted node and edge features, which can provide valuable domain-specific insights and cues to the model. The concatenation of attention heads helps the model integrate the different perspectives learned by the attention mechanisms, leading to a more comprehensive understanding of the graph structure.

The use of mean pooling and layer normalization techniques helps the model to effectively aggregate information and normalize the feature representations, respectively. These components contribute to the model's ability to generalize well and handle the inherent complexity of graph data.

Lastly, the inclusion of residual connections in the model architecture is an important design choice. Residual connections allow the model to bypass certain layers, facilitating the flow of information and gradients during training. This can help the model learn more effective representations, mitigate the vanishing gradient problem, and improve the overall performance on the graph classification task.

The combination of these components in model 5 likely enables it to capture the intricate relationships and patterns within the graph data more effectively compared to the other models, as evidenced by the RMSE results and good scatter plot for each of the 14 tasks.

0.7. #Layers > diameter

In the final step, we tested the best performing model (model 5) with the `DeepMultiHeadGAT`. Fig.6 shows the RMSE (Root Mean Squared Error) values for two models - model5 and model6. Model6 has not a significantly higher RMSE value compared to model5, indicating that model5 cannot be ruled out for model6.

Note that model6 has more than the graph diameter number of layers. From a theoretical perspective, this is expected to worsen the performance, as having more layers than the graph diameter can lead to oversmoothing, where the node features become indistinguishable. However, we note that the performance of the deep multi-head attention model seems to be about the same as the best performing model (model5). However, this result could be because the number of layers (just 2 [2]) is not enough to introduce the oversmoothing effect and worsen the performance. Additionally, the training of the deep multi-head attention model took significantly longer compared to model5, which means that it is more computationally complex and may not provide a substantial performance improvement over the simpler, yet effective, model5.

In summary, the performance similarity between the best performing model (model5) and the deep multi-head attention model with more layers than the graph diameter can be attributed to the relatively small number of layers (2) in the deep model, which did not lead to the anticipated oversmoothing and performance degradation. This result highlights the importance of considering the trade-off between model complexity and performance, as well as the potential limitations of theoretical expectations when dealing with real-world graph data and architectures.

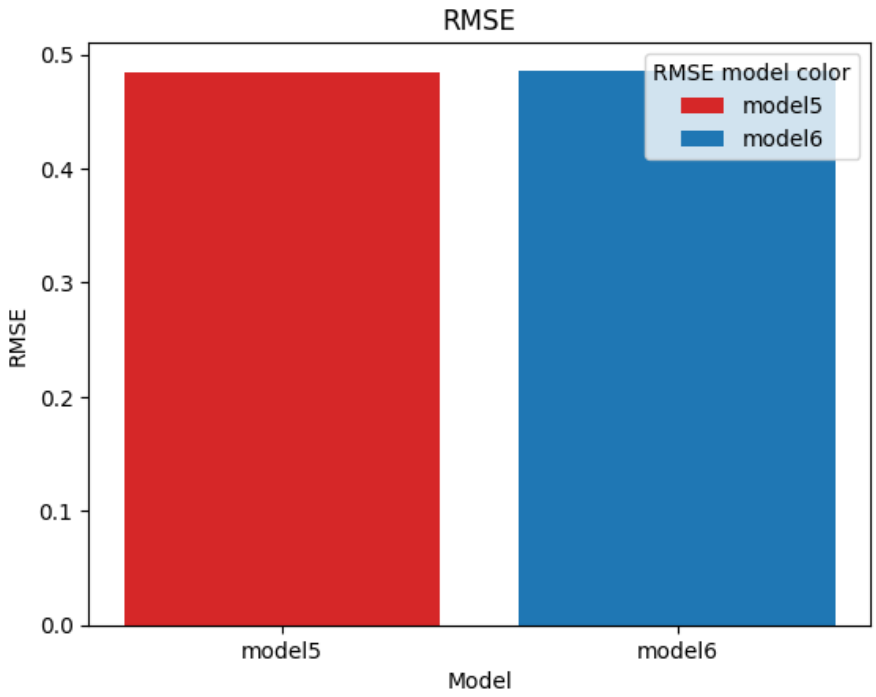


Figure 6: Enter Caption

[2]We were limited to using only 2 layers due to the RAM constraints in Google Colab.

References

[1] Petar Veličković et al. *Graph Attention Networks*. Available at: https://arxiv.org/abs/1710.10903. 2017. URL: https://arxiv.org/abs/1710.10903.

[2] Zhenqin Wu et al. *MoleculeNet: A Benchmark for Molecular Machine Learning*. Available at: https://arxiv.org/abs/1703.00564. 2017. URL: https://arxiv.org/abs/1703.00564.