A Technical Report on Graph Neural Networks
Applying GNNs to the ZINC Dataset

Abdallah Abdelsameia
Mathematics of Machine and deep Learning Algorithms
Toulouse School of Economics
`abdallah.abdelsameia@ut-capitole.fr`

Abstract

Graph Neural Networks (GNNs) offer a powerful framework for analyzing graph-structured data, including molecular graphs where atoms are nodes and bonds are edges. In this project, we explore various GNN architectures—*GCN*, *GIN*, *GINE*, *GAT*, *GraphSAGE*, and a simplified *Graph Transformer*—to predict molecular properties within the ZINC dataset. By leveraging edge attributes (e.g., bond types), our edge-aware GINE models substantially outperform edge-agnostic counterparts, achieving a mean squared error (MSE) near 1.0 and an R^2 exceeding 0.79 on the full dataset. We demonstrate that integrating techniques such as attention-based pooling, residual connections, batch normalization, and dropout further enhances accuracy and stabilizes training. Experimental results highlight the critical role of bond-specific information for capturing chemical nuances and underscore the broader potential of GNNs for drug discovery. Our findings suggest that well-designed, edge-aware GNNs can significantly improve predictive performance in molecular property estimation tasks.

1 Introduction

Graphs are powerful mathematical abstractions used to describe relationships among entities, appearing in diverse contexts such as chemical structures, social networks, and transportation systems. In chemistry, *molecular graphs*—where atoms are nodes and bonds are edges—offer a natural representation of a molecule's structure and properties. This report focuses on exploiting **Graph Neural Networks (GNNs)** for property prediction on the ZINC dataset, a collection of molecular graphs widely used in drug discovery research.

By applying GNNs, we aim to move beyond traditional feature engineering and instead learn representations directly from the graph topology and node/edge attributes. Throughout this document, we will:

- Review the fundamental concepts of graph theory and introduce state-of-the-art GNN architectures.
- Detail our experimental pipeline, including dataset preparation, model configuration, and evaluation metrics.
- Discuss the results obtained on the ZINC dataset, highlighting how the integration of edge-aware GNNs improves predictive accuracy for molecular property prediction.
- Provide insights into challenges, future directions, and broader implications of using GNNs for chemistry applications.

Our ultimate goal is to demonstrate how GNNs can capture complex relationships within molecular graphs and, in doing so, pave the way for more accurate drug discovery workflows and other chemistry-related tasks.

2 Background

2.1 Graph Theory

Graphs are a ubiquitous mathematical structure used to model relationships between entities in numerous domains, ranging from social networks and transportation systems to molecular structures in chemistry. Formally, a *graph* G is defined as a pair (V, E), where:

- V is a set of *vertices* (or *nodes*),
- $E \subseteq V \times V$ is a set of *edges* (or *links*) connecting pairs of vertices.

2.1.1 Mathematical Definition

Let $V = \{v_1, v_2, \ldots, v_n\}$. An edge can be represented as a pair (v_i, v_j). Depending on the nature of the graph:

- **Undirected Graph:** An edge (v_i, v_j) implies there is a connection between v_i and v_j in both directions (i.e., the edge set is unordered).
- **Directed Graph (Digraph):** An edge $(v_i \rightarrow v_j)$ implies a link from v_i to v_j, not necessarily from v_j back to v_i.
- **Multigraph:** A graph that may contain multiple edges (parallel edges) between the same pair of vertices.
- **Pseudo-graph:** A graph that also allows self-loops (edges from a vertex to itself).

A *walk* in a graph is a sequence of vertices and edges (with possible repetitions), while a *path* is a walk with no repeated vertices. A *cycle* is a path whose start and end vertices coincide (and contains at least one edge). We say an undirected graph is *connected* if every vertex can be reached from any other vertex via some path. For directed graphs, the analogous concept is *strongly connected* (each vertex is reachable from every other vertex via directed edges).

Subgraphs. A *subgraph* $H = (V_H, E_H)$ of a graph $G = (V_G, E_G)$ is any graph whose vertex set V_H is a subset of V_G, and whose edge set E_H is a subset of E_G restricted to the vertices in V_H. An *induced subgraph* is one where E_H contains all edges in E_G that connect the chosen vertices V_H.

Isomorphism. Two graphs G_1 and G_2 are *isomorphic* if there exists a bijection between their vertices that preserves the adjacency structure. Determining whether two graphs are isomorphic is a central open problem in computer science.

Weighted Graphs. In a *weighted* graph, each edge (v_i, v_j) may store a numerical weight $w_{i,j}$ instead of a simple 1 or 0. Weights can represent distances, costs, similarities, or capacities depending on the application. In our particular situation edges weights are bond types.

2.1.2 Graph Representations

Several data structures are used to represent a graph:

Adjacency List. An adjacency list keeps, for each vertex v_i, a list of vertices it is connected to. This representation is memory-efficient for sparse graphs (where the number of edges is on the same order as $|V|$) but can be less convenient for certain matrix-based operations or algorithms requiring frequent random access to edges.

Adjacency Matrix. An adjacency matrix A is an $n \times n$ matrix where:

$$A[i,j] = \begin{cases} 1, & \text{if there is an edge between } v_i \text{ and } v_j, \\ 0, & \text{otherwise.} \end{cases}$$

In a *weighted* graph, $A[i,j]$ may store a numerical weight instead of a simple 1 or 0. For undirected graphs, A is symmetric ($A = A^T$), whereas directed graphs generally yield a non-symmetric A. Although an adjacency matrix allows fast lookup for edges, it can be expensive in memory for large sparse graphs (since it stores n^2 entries). Further, if we represent different adjacency matricies for different edge types, the 3D strcture of the matrix becomes harder to handle, multiple

Incidence Matrix. Another less common representation uses an *incidence matrix*, which is an $n \times |E|$ matrix that marks the relationship between vertices and edges. This format can be convenient for certain linear-algebra-based graph algorithms but is less frequently used in GNN contexts.

2.1.3 Graph Types & Properties

Complete Graph (K_5). A complete graph on 5 vertices has an edge between *every* pair of nodes. Hence, $|E| = \frac{5\times4}{2} = 10$ edges in the undirected case.

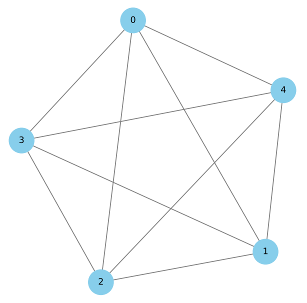


Figure 1: Visualization of a complete graph on 5 nodes (K_5). Every node is connected to every other node.

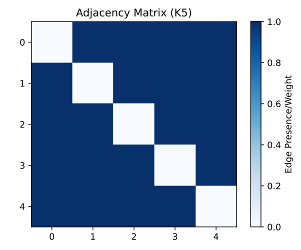


Figure 2: Adjacency matrix for K_5. Dark cells indicate edges between nodes.

Cycle Graph (C_4)**.** A cycle graph with 4 vertices, denoted C_4, arranges nodes in a closed loop. It's often used as a simple example of a cyclic structure in graph theory.

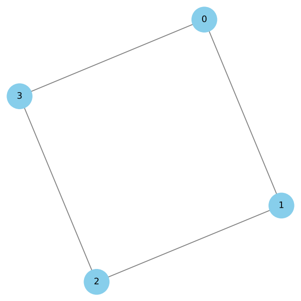


Figure 3: A cycle graph with 4 nodes (C_4). Each node connects to two neighbors, forming a single cycle.

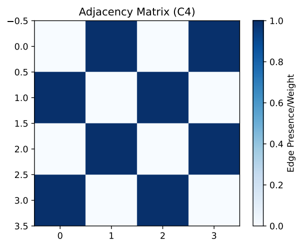


Figure 4: Adjacency matrix of the C_4 cycle graph. Each node is adjacent to exactly two others.

Bipartite Graph. A bipartite graph splits the vertex set into two disjoint subsets such that no edges exist within the same subset. For example, $\{0, 1\}$ and $\{2, 3\}$ in the figure below.

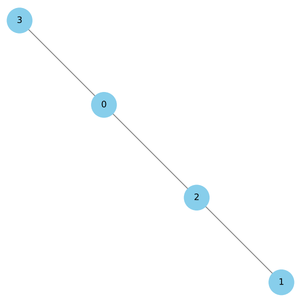


Figure 5: Bipartite graph example where nodes $\{0, 1\}$ only connect to $\{2, 3\}$.

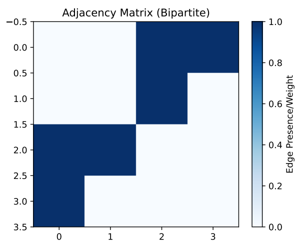


Figure 6: Adjacency matrix for the bipartite graph, showing no edges among nodes in the same subset.

Trees. A tree is a connected, acyclic graph. With n vertices, a tree always has $n-1$ edges. Common algorithms on trees include Depth-First Search (DFS) and Breadth-First Search (BFS) to explore or label nodes in a rooted or unrooted fashion.

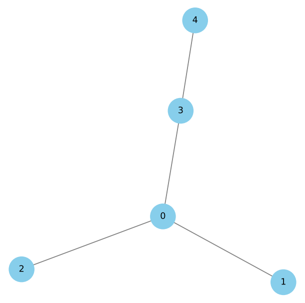


Figure 7: A simple tree structure with 5 nodes, forming a star plus one branch.

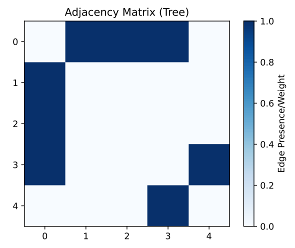


Figure 8: Adjacency matrix for a 5-node tree, highlighting the acyclic, minimally connected structure.

Connectedness and Cliques. An undirected graph is *connected* if there is a path between every pair of vertices. A *clique* is a subset of vertices in which every two distinct vertices are adjacent. Studying cliques and connectivity is essential in social network analysis, combinatorial optimization, and many other domains.

These foundational concepts in graph theory serve as building blocks for more sophisticated techniques, including **Graph Neural Networks**, which leverage these principles to learn data-driven representations of graph-structured information. **Our aim is to use the basic concepts of graph theory to better understand the representation of the chemical molecules in our data set and be able to better utitlize GNN archetictures.** In the next section, we explore the essential ideas behind GNNs and how they combine graph-theoretic foundations with modern deep learning.

2.2 Graph Neural Networks (GNNs)

In recent years, **Graph Neural Networks (GNNs)** have attracted considerable attention as a powerful framework for analyzing graph-structured data, which appear in numerous domains such as social networks, biological systems, communication infrastructures, and molecular chemistry Zhou et al. [2021], Yun et al. [2019]. GNNs extend the ideas behind standard neural networks to handle arbitrary topologies, enabling them to process graph inputs where the ordering and number of neighbors can vary across nodes.

2.2.1 Core Concepts and Motivations

The foundational idea of GNNs is to represent nodes (or sometimes edges and entire subgraphs) with latent embeddings, which are iteratively updated based on the connectivity of the graph. This *iterative update* paradigm is often framed as *message passing* or *graph convolution* Zhou et al. [2021]:

1. **Local Aggregation:** Each node collects ("aggregates") representations from its neighbors.

2. **Combine & Update:** The node then combines these neighbor features—often via a learnable function (e.g., a linear transform or MLP) and a nonlinear activation—to produce an updated node embedding.

3. **Multiple Layers:** By stacking several such layers, a node can receive information from multi-hop neighborhoods, capturing higher-order dependencies.

This procedure parallels the convolution operation in image processing, but rather than a fixed grid neighborhood, each node's neighborhood is defined by the graph's adjacency structure. Historically, GNNs were introduced via two main perspectives:

- *Spectral Methods* Zhou et al. [2021]: Convolutions are defined in the spectral domain, leveraging the eigenbasis of the graph Laplacian. Classical examples include *ChebNet* and early *Graph Convolutional Networks (GCNs)*.

- *Spatial Methods*: Emphasize direct neighbor aggregation in the node domain (a "message passing" interpretation). Examples include *GraphSAGE*, *Graph Attention Networks (GAT)*, and many variants that use localized "filters" operating on each node's neighborhood.

While spectral methods have strong theoretical underpinnings, spatial methods are often more scalable and intuitive for large or diverse graphs. We use some of the Spatial methods in our applications in this paper.

GNNs combine the computational graph of neural networks with the combinatorial graph from classic graph theory. Key graph-theoretic concepts—such as adjacency matrices, connectivity, and node degrees—directly shape how information flows during GNN training. For instance, $\tilde{A} = A + I$ (adding self-loops) is a common trick ensuring each node also sees its own features at each layer.

2.2.2 Graph Convolution Mechanisms

In many widely used GNN architectures, the "graph convolution" step can be summarized as follows Zhou et al. [2021]:

Graph Convuloution

$$\mathbf{h}_v^{(l+1)} = \sigma\Big(\text{AGGREGATE}\big(\{\mathbf{h}_u^{(l)} : u \in \mathcal{N}(v)\}\big)\, \mathbf{W}^{(l)}\Big),$$

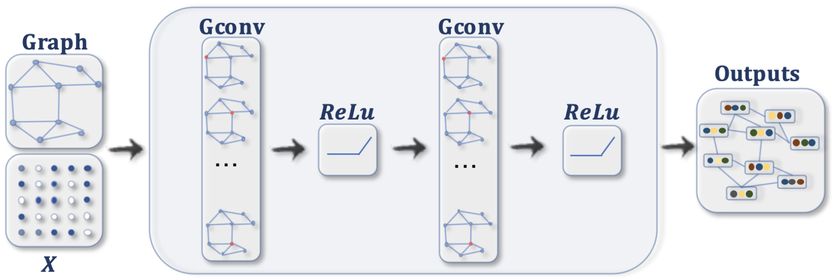


Figure 9: An illustrative overview of a two-layer GNN pipeline, showing how node features and adjacency information flow through stacked graph convolutions (Message Passing).

where $\mathcal{N}(v)$ denotes neighbors of node v, ACT is an activation function, and different GNN variants define different AGGREGATE and transformation steps. To illustrate, each model (GCN, GIN, GINE, GAT, GraphSAGE) adopts a **two-layer** graph convolution design for illustrative purposes. During each graph convolution (or *message-passing*) step, every node v gathers the current embeddings of its neighbors $\mathcal{N}(v)$. These neighbor embeddings are then combined via an *aggregation* function (e.g., sum, mean, or attention) and passed through an *activation* function (ACT), such as ReLU. Formally, this process updates each node's embedding $\mathbf{h}_v^{(l)}$ (at layer l) to a new embedding $\mathbf{h}_v^{(l+1)}$. Although all these GNN variants share this overall neighbor-aggregation concept, they differ in how they implement the aggregation and transformation steps—for example, GCN uses normalized adjacency matrices, GIN applies MLP-based aggregators, GINE incorporates edge attributes, GAT employs attention coefficients, and GraphSAGE employs different neighborhood sampling strategies.

- **Activation Function** (`activation`): e.g., `relu` or `leakyrelu`.
- **Dropout** (`dropout_layer`): randomly zeroes out elements in the embedding, providing regularization.
- **BatchNorm** (`batch_norm`): normalizes each node's embedding to have zero mean and unit variance, further stabilizing training.
- **Residual Skip Connections** (`residual`): we optionally add the previous layer's embedding $\mathbf{x}_{\text{in}}$ to the post-convolution embedding $\mathbf{x}_{\text{out}}$ if their shapes match. This helps mitigate *over-smoothing* and eases optimization in deeper GNNs.

Pooling (Readout). When the goal is to predict a property of an entire graph (rather than individual nodes), each node's final embedding must be *aggregated* into a single vector representing the entire graph. We refer to this aggregation step as *pooling*. In our framework, we provide a **pooling function** pool $\in \{\texttt{mean}, \texttt{max}, \texttt{attention}\}$, which reduces the set of node embeddings to one global embedding:

- `mean`: Computes the average (mean) of all node embeddings, resulting in a global mean pool. This approach treats each node equally and is often effective for moderately sized graphs, though it may overlook varying node importance.
- `max`: Takes the elementwise maximum across all node embeddings. Here, the most prominent features (largest activations) in the node set dominate the global representation. This can capture strong local signals but sometimes ignores subtler global structures.
- `attention`: Employs a learned gating mechanism (`GlobalAttention`) to weigh each node's embedding prior to aggregation. By assigning higher weights to certain nodes, this method can focus on the most relevant parts of the graph, yielding a more expressive readout.

Once a single graph-level vector is obtained, a final linear layer (`self.lin`) produces the ultimate predictions (regression or classification targets). This architectural pattern of *(1) node-level message passing followed by (2) global pooling and (3) a prediction layer* is a standard approach for graph-level tasks such as molecular property prediction.

Further, we delve deeper into the mathematical underpinnings of the popular GNN layers used in our models. We focus on their layer-wise update equations, their handling (or ignoring) of edge attributes, and the role of hyper parameters such as normalization, trainable skip terms, and attention heads. All the structure of the math follows the official documentation on torch-geometric, the reader is advised to refer to the original documentation for further details and references.

2.2.3 GCNModel:

A single GCN layer, as introduced in Kipf and Welling [2016], can be written in matrix form as:

GCN Layer

$$\mathbf{X}' = \hat{\mathbf{D}}^{-\frac{1}{2}} \, \hat{\mathbf{A}} \, \hat{\mathbf{D}}^{-\frac{1}{2}} \, \mathbf{X} \, \Theta,$$

where:

- $\mathbf{X} \in \mathbb{R}^{N \times d}$ is the node feature matrix, for N nodes each with d-dimensional features.
- $\Theta \in \mathbb{R}^{d \times d'}$ is a learnable weight matrix that maps from d input features to d' output features (per node).
- $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}$ is the adjacency matrix $\mathbf{A}$ with self-loops added along the diagonal.
- $\hat{\mathbf{D}}$ is the diagonal degree matrix of $\hat{\mathbf{A}}$, with $\hat{D}_{ii} = \sum_j \hat{A}_{ij}$.

The multiplication by $\hat{\mathbf{D}}^{-\frac{1}{2}}$ on both sides effectively normalizes node features by their degrees, mitigating issues with scale differences across different nodes.

The node-wise expression for the updated feature $\mathbf{x}_i'$ of node i is:

Node-Wise Expression

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

where $\hat{d}_i = 1 + \sum_{j \in \mathcal{N}(i)} e_{j,i}$ (due to adding self-loops, hence the "+1" in the degree), and $e_{j,i}$ represents the edge weight from node j to node i. In most simple settings, $e_{j,i} = 1$.

GCNs are motivated by the idea of spectral filtering on graphs, where $\hat{\mathbf{A}}$ approximates a low-frequency filter in the graph Fourier domain Kipf and Welling [2016].

2.2.4 GINModel:

The Graph Isomorphism Network (GIN), modifies the convolution by using an MLP-based aggregator. In its simplest form:

GINE Aggregator

$$\mathbf{x}_i' = h_\Theta\Big((1 + \epsilon) \, \mathbf{x}_i \; + \sum_{j \in \mathcal{N}(i)} \mathbf{x}_j\Big),$$

where:

- h_Θ is a learnable MLP (e.g., multiple linear layers with non-linear activations).
- ϵ is a parameter (scalar) that can be fixed or trainable, giving the layer a "skip" or "self-connection" weighting on $\mathbf{x}_i$.

This formulation is designed to be as powerful as the Weisfeiler-Lehman (WL) test for graph isomorphism, under certain conditions. The sum-based aggregator is crucial for capturing topological distinctions.

- We typically stack two (or more) `GINConv` layers, each with its own MLP.
- `edge_attr` is *ignored* in the vanilla GIN version, i.e., edges are assumed to have no additional features beyond connectivity.
- Optional `batch_norm` and `residual` connections can be applied.

2.2.5 GINEModel:

The GINE operator extends GIN to incorporate edge attributes $\mathbf{e}_{j,i}$. Its aggregator is:

GINE Model Structure

$$\mathbf{x}'_i = h_\Theta\Big((1+\epsilon)\,\mathbf{x}_i \;+\; \sum_{j\in\mathcal{N}(i)} \mathrm{ReLU}\big(\mathbf{x}_j + \mathbf{e}_{j,i}\big)\Big).$$

Here, the edge attributes are first added into the neighbor's embedding, then a ReLU nonlinearity is applied before summation. By including $\mathbf{e}_{j,i}$, the message depends explicitly on edge-specific information (e.g., bond type or distance).

Dimension Handling in `GINEModel`:

- If `edge_attr` is 1D, we reshape it to $(E, 1)$ to align with the GINEConv's requirement.
- The parameter `edge_dim` indicates whether a linear transformation is applied to the edge features to match the node feature dimension.
- As in GIN, `batch_norm`, `residual`, and multiple layers can be used.

This modification is particularly beneficial when edges encode meaningful information that cannot be overlooked, significantly improving representational power on tasks such as molecular property prediction.

2.2.6 GATModel:

Graph Attention Networks (GAT) Veličković et al. [2018] learn to compute an attention coefficient ($\alpha_{i,j}$ for each edge (i, j). The layer's update for node i is:

GAT Structure

$$\mathbf{x}'_i = \sum_{j\in\mathcal{N}(i)\cup\{i\}} \alpha_{i,j}\,\Theta_t\,\mathbf{x}_j,$$

where Θ_t is a trainable transform. The coefficients $\alpha_{i,j}$ are normalized exponentials of a shared attention mechanism:

GAT Attention Coefficicnet

$$\alpha_{i,j} = \frac{\exp\Big(\mathrm{LeakyReLU}\big(\mathbf{a}_s^\top\Theta_s\mathbf{x}_i + \mathbf{a}_t^\top\Theta_t\mathbf{x}_j + \mathbf{a}_e^\top\Theta_e\mathbf{e}_{i,j}\big)\Big)}{\sum_{k\in\mathcal{N}(i)\cup\{i\}}\exp\Big(\mathrm{LeakyReLU}\big(\mathbf{a}_s^\top\Theta_s\mathbf{x}_i + \mathbf{a}_t^\top\Theta_t\mathbf{x}_k + \mathbf{a}_e^\top\Theta_e\mathbf{e}_{i,k}\big)\Big)},$$

in the case where edge features $\mathbf{e}_{i,j}$ are employed. In many implementations (including `GATModel`), edge attributes may be turned off (`edge_attr` ignored), defaulting to the simpler form without $\mathbf{a}_e$ or Θ_e.

Multi-Head Attention. GAT often uses multiple attention heads (say M heads) to stabilize learning and enrich representation. Each head computes a separate set of $\alpha_{i,j}^{(m)}$ and $\Theta^{(m)}$, either concatenating or averaging the outputs across heads:

Multi-head Attention

$$\mathbf{x}_i'^{(\text{concat})} = \Vert_{m=1}^{M} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{i,j}^{(m)} \Theta_t^{(m)} \mathbf{x}_j, \quad \text{or} \quad \mathbf{x}_i'^{(\text{avg})} = \frac{1}{M} \sum_{m=1}^{M} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{i,j}^{(m)} \Theta_t^{(m)} \mathbf{x}_j.$$

In `GATModel`, we expose hyperparameters like `heads`, `dropout`, and optional `residual` connections to mitigate training difficulties.

2.2.7 SAGEModel:

GraphSAGE Hamilton et al. [2017] introduced an inductive framework where node embeddings are generated by sampling and aggregating neighbor features. A typical SAGE layer uses:

SAGE Structure

$$\mathbf{x}_i' = \mathbf{W}_1 \mathbf{x}_i + \mathbf{W}_2 \cdot \text{mean}_{j \in \mathcal{N}(i)}(\mathbf{x}_j),$$

where $\mathbf{W}_1$ and $\mathbf{W}_2$ are learnable parameters. Various aggregators (e.g., mean, max-pool, LSTM) can be used.

Normalization and Inductive Capability. Often, a non-linear activation and *layer normalization* steps are applied, and one can scale or normalize $\mathbf{x}_i'$ (depending on the `normalize` flag). The key advantage is that GraphSAGE can naturally handle unseen nodes at test time by aggregating from their neighbors, as it does not require a global Laplacian or adjacency matrix factorization.

Implementation in `SAGEModel`.

- Similar to the other models, `SAGEModel` can incorporate `dropout`, `batch_norm`, and `residual` connections.
- While `edge_attr` is not explicitly used in standard SAGEConv, one can extend the aggregator to weigh neighbors by edge features or combine them in the transform.
- Large-scale implementations may use *neighbor sampling* to handle high-degree nodes efficiently and support inductive training on massive graphs.

Overall, these layers capture a broad range of design philosophies—spectral filtering (GCN), powerful injective mapping (GIN/GINE), attention-based (GAT), and inductive neighbor-aggregation (SAGE)—each with unique mathematical formulations and practical trade-offs. Given the very limited time frame of this project, the choice of the underlying architectures might not have been the optimal- but the main aim was to expermint with different architectures and see how they perform.

2.3 Graph Transformers and Their Differences from Conventional GNNs

Recent work on *Graph Transformer Networks (GTNs)* Yun et al. [2019] expands traditional Graph Neural Network (GNN) paradigms by learning *new* graph structures—often referred to as *meta-path graphs*—rather than relying on a fixed adjacency matrix. This approach is especially beneficial in heterogeneous or noisy settings, where node or edge types vary substantially or where the existing graph may be misspecified.

Key Ideas.

- **End-to-End Graph Generation.** Unlike most GNNs that assume a *fixed* graph structure, GTNs learn to *softly select and combine* edges from multiple input adjacency matrices (e.g., for different edge types). Through matrix multiplications and learned weights (as in a "convolution" over edge-type tensors), GTNs create new "meta-path" graphs. For instance, an author in a citation network might connect to relevant conferences via intermediate papers, forming an `Author` $\rightarrow$ `Paper` $\rightarrow$ `Conference` path. GTNs determine which paths—and how many hops—are most useful, rather than requiring them to be hand-specified.

- **Handling Heterogeneous Graphs.** Many real-world graphs (e.g., social networks, citation networks, molecule-ligand complexes) contain multiple node/edge types. By generating separate adjacency matrices for each relation type (e.g., `Paper` $\leftrightarrow$ `Author`, `Paper` $\leftrightarrow$ `Conference`), GTNs assemble these base matrices into multi-hop "composite" edges. This procedure allows the model to learn higher-order or cross-type relationships without manual meta-path engineering.
- **Meta-Path Learning.** A *meta-path* is a sequence of edge types, such as `Author` $\rightarrow$ `Paper` $\rightarrow$ `Author` (APA). Traditional methods either fix such paths or require domain experts to manually specify them. GTNs, however, discover these paths automatically by assigning different attention scores to each edge type combination. This yields interpretability—one can inspect the highest-weighted paths to see which composite relations the model finds salient for a particular task.
- **Graph Convolution on Learned Structures.** Once a meta-path adjacency matrix is formed, GTNs apply a typical graph convolution layer (akin to GCN) on that newly formed matrix, thereby enabling end-to-end optimization of *both* the graph structure *and* the node embeddings. Multiple channels can be concatenated (or otherwise combined) for richer expressiveness.

Differences from Conventional GNNs.

- **Fixed vs. Learned Graph Structure:** Standard GNNs (e.g., GCN, GIN, GAT) operate on one given adjacency matrix (possibly with minor modifications like adding self-loops). GTNs, in contrast, *learn* new adjacency matrices to emphasize crucial connections and de-emphasize less relevant ones.
- **Heterogeneous vs. Homogeneous Settings:** While classic GNNs typically assume homogeneous graphs (one node type, one edge type), GTNs natively handle multiple node/edge types by constructing meta-paths. This is particularly advantageous for tasks like node classification on multi-relational (heterogeneous) graphs.
- **Soft Meta-Path Selection:** GTNs employ a mechanism analogous to attention over edge types, enabling them to "softly pick" relevant paths for each layer. Traditional GNNs do not commonly include any step that *adapts* the underlying graph structure itself.
- **Interpretability Through Learned Paths:** Because GTNs produce explicit adjacency matrices for each newly discovered meta-path, users can trace important relationships, offering insights that go beyond the black-box nature of many GNNs.

In summary, Graph Transformer Networks offer a more adaptive approach than conventional GNNs by generating meta-path-based graph structures in an end-to-end fashion. This adaptability is particularly advantageous for handling heterogeneous or incomplete data, as it minimizes reliance on domain-specific preprocessing and can adjust to noisy or missing edges. Consequently, GTNs have shown promising results in complex relational tasks (e.g., node classification within citation networks).

In our project, we present an initial attempt to integrate portions of the official GTN implementation with the ZINC dataset. While the idea of producing multiple adjacency matrices based on bond types could potentially unveil deeper insights into molecular structures, our current experiments are still preliminary. Hence, no formal results from Graph Transformers are reported here due to the constrained time frame.

3 Experiments

This section details our experimental setup, including the dataset, model architectures, training protocol, and the specific variations we tested. We employed PyTorch Geometric for all graph operations and leveraged our custom Python scripts for data loading, model definition, and training loops.

3.1 Dataset and Preprocessing

We use the **ZINC** dataset, which comprises molecular graphs intended for property prediction tasks in drug discovery. Each molecule is represented as a graph, where:

- **Nodes** (atoms) contain various features (e.g., atom type).
- **Edges** (bonds) may include bond types as `edge_attr`.

By default, we use the smaller, `subset` version of ZINC to ensure faster experimentation. The subset contains 12,000 molecules, while the Full data set contains almost 250,000 molecules. For our initial analysis we test train and analyze our results based on the We split the data into training (`train`), validation (`val`), and test (`test`) sets, provided by the built-in PyTorch Geometric `ZINC` class. Listing 1 illustrates our data-loading utility.

```python
# src/data_utils.py
def get_zinc_dataset(root='../data/ZINC', batch_size=64, subset=True):
    train_dataset = ZINC(root, split='train', subset=subset)
    val_dataset   = ZINC(root, split='val',   subset=subset)
    test_dataset  = ZINC(root, split='test',  subset=subset)

    train_loader = DataLoader(train_dataset, batch_size=batch_size,
        shuffle=True)
    val_loader   = DataLoader(val_dataset,   batch_size=batch_size,
        shuffle=False)
    test_loader  = DataLoader(test_dataset,  batch_size=batch_size,
        shuffle=False)

    return train_loader, val_loader, test_loader
```

Listing 1: Data loader for ZINC.

The main task is to predict the value y (a regression task) for each molecular structure. Specifically, we focus on the *penalized* $\log P$, also called *constrained solubility* in some literature. Formally, this target is defined as

$$y = \log P - \text{SAS} - \text{cycles},$$

where $\log P$ is the water-octanol partition coefficient, SAS is the synthetic accessibility score, and cycles denotes the number of cycles containing more than six atoms.

We begin by performing exploratory analysis on the ZINC dataset to understand the underlying molecular structures. A typical example of a random molecule in the dataset can be represented as:

$$\texttt{Data(}\mathbf{x} = [27,\ 1],\ \text{edge\_index} = [2,\ 60],\ \text{edge\_attr} = [60],\ y = [1]\texttt{)},$$

where $\mathbf{x}$ contains integer-encoded atom types, so each unique atom type corresponds to a different integer. The edge_index structure maps which atom is connected to which other atom, and edge_attr specifies the bond type (integer values from 1 to 4), typically interpreted as single, double, triple, or aromatic bonds. Finally, y is the numerical target value defined above, which we aim to predict.

Below, in Figure 10, we illustrate some random molecules from the ZINC dataset. Each molecule is visualized as a graph where nodes correspond to atoms, colored according to their atom types, and edges correspond to bonds, colored by bond type:

In this figure, node i has an integer-coded feature $\mathbf{x}_i$ that identifies its atom type (e.g., atomic number), which determines the node's color from a discrete colormap. Meanwhile, bond information is stored in the edge_attr field, which appears to encode bond types (1 = single, 2 = double, 3 = triple, 4 = aromatic). These integer codes for atom and bond types originate from the preprocessed ZINC pickle

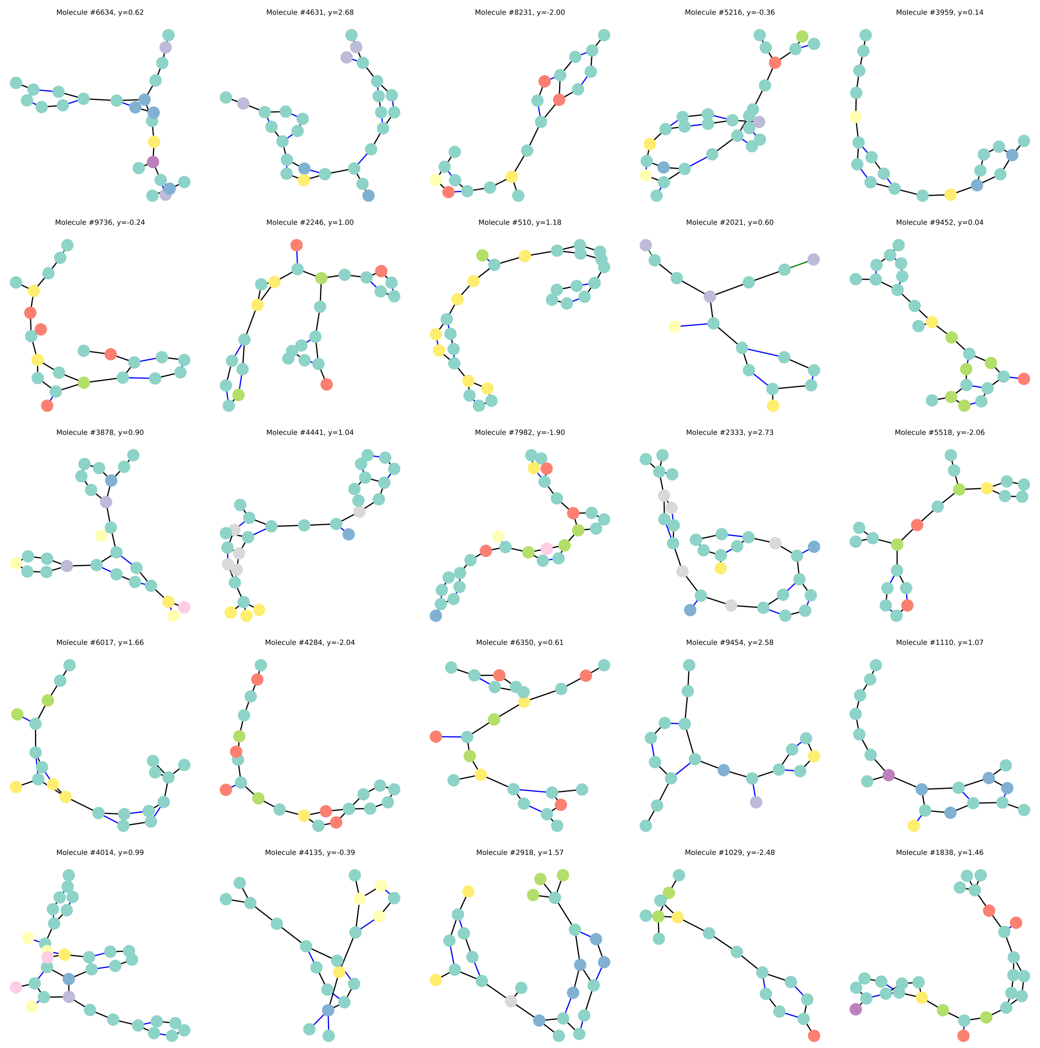


Figure 10: A graphical representation of selected random molecules from the ZINC dataset and their respective y values. Nodes represent atoms, and edges represent chemical bonds. Node colors correspond to atom types, and edge colors reflect bond types.

files provided with PyG and are inferred to be consistent with standard atomic numbers and common bond orders. However, the exact mapping itself is not fully documented in the PyG repository, since the encoding was generated externally (e.g., with RDKit-based scripts) prior to integration into PyG.

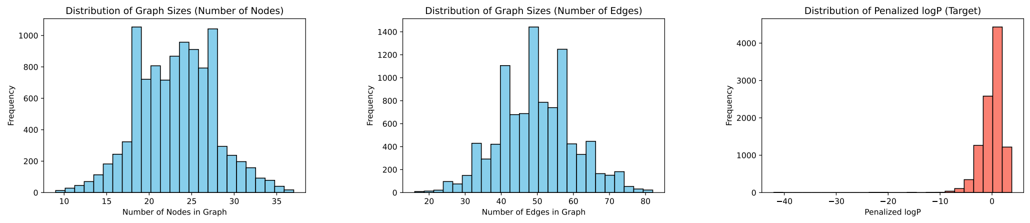
3.2 Dataset Exploration

We further explored the dataset to characterize the range of nodes, edges, and target values (y) in our sampled subset. Table 1 summarizes the basic statistics:

	Min	**Max**	**Average**
Nodes	9	37	23.17
Edges	16	82	49.86

Table 1: Basic statistics of the sampled dataset: minimum, maximum, and average counts of nodes and edges per graph.

Figure 11 presents histograms illustrating the distributions of the number of nodes, the number of edges, and the target variable y (the penalized $\log P$). As expected, most molecules in our sample fall within moderate sizes, and the target values exhibit a unimodal distribution typical for organic molecule datasets.



(a) Number of nodes (b) Number of edges (c) Penalized $\log P$ (y)

Figure 11: Histograms showing the distribution of nodes, edges, and the penalized $\log P$ target variable (y) in the sampled subset of the dataset.

In our training set of 10,000 molecules, the distributions of node and edge counts appear roughly unimodal and close to normal, with occasional spikes around certain atom and bond counts. By contrast, the target variable (penalized $\log P$) is left-skewed and contains multiple outliers, which can pose challenges for regression-based models. Rather than applying specialized outlier handling, we have chosen to retain the dataset in its original form, relying on our models to naturally learn from these variations.

For all experiments, we use the default ZINC split and adjust only the `batch_size` (64 in our case) to fit available GPU memory. We run the models on two different systems: an M1 Mac and an L4 GPU on Google Colab. Overall performance is comparable across both devices, except for GAT training, where the GPU proves to be about four times faster.

3.3 Model Variants

We evaluated five main families of graph neural networks, each offering distinct ways to perform message passing on graph data. Our `get_model(...)` function instantiates specific architectures by name:

- **GCN**: Standard graph convolutional layers **?**.
- **GIN**: Graph Isomorphism Network Xu et al. [2019].
- **GINE**: A GIN extension supporting edge attributes (e.g., bond types).
- **GAT**: Graph Attention Network Veličković et al. [2018], applying multi-head attention on node features.
- **SAGE**: Uses neighbor sampling and an aggregator to build node embeddings Hamilton et al. [2017].

These models allow customization via parameters such as:

- **hidden_dim**: The hidden dimension size (e.g., 64).
- **dropout**: Dropout rate (e.g., 0.3).
- **activation**: Activation function (e.g., `relu` or `leakyrelu`).
- **pool**: Pooling method (`mean`, `max`, or `attention`).
- **residual**: Whether to add residual connections between layers.
- **batch_norm**: Whether to apply batch normalization after each layer.
- **heads**: Number of attention heads (for GAT/Transformer).
- **edge_dim**: Dimension of edge features (for GINE).

We emphasize that only GINE made use of edge attributes (bond types), providing it with an advantage in tasks where such features are critical. While GAT can also leverage edge information, in practice, we faced computational complexities, and thus only GINE actually utilized edge features in this study.

3.4 Training Setup

For all experiments, we used:

- **Optimizer:** Adam with learning rate $\alpha = 0.001$.
- **Loss Function:** Mean Squared Error (MSE).
- **Epochs:** 3000 (without early stopping).
- **Batch Size:** 64.

We chose MSE loss because the ZINC task involves regression to predict chemical properties. We also report other metrics like Mean Absolute Error (MAE) and R^2 to provide additional insights.

Listing 2 shows our simplified training loop for a single epoch, where we: 1. Set the model to `train` mode. 2. Iterate over each batch from the data loader. 3. Zero the gradients, perform a forward pass, compute the loss, and backpropagate. 4. Accumulate the total loss for monitoring.

```python
# src/train.py
def train_one_epoch(model, dataloader, optimizer, criterion, device):
    model.train()
    total_loss = 0
    for batch_data in dataloader:
        batch_data = batch_data.to(device)
        optimizer.zero_grad()

        out = model(
            x=batch_data.x.float(),
            edge_index=batch_data.edge_index,
            batch=batch_data.batch,
            edge_attr=batch_data.edge_attr.float() if batch_data.
                edge_attr is not None else None
        ).squeeze(-1)

        loss = criterion(out, batch_data.y.float())
        loss.backward()
        optimizer.step()

        total_loss += loss.item() * batch_data.num_graphs

    return total_loss / len(dataloader.dataset)
```

Listing 2: Training loop for one epoch.

We perform validation after each epoch using a similar routine but with the model in `eval` mode and no gradient updates.

3.5 Experimental Variations

We ran **15 experiments** (5 model families $\times$ 3 variations each) using the same splits of ZINC. Our goal was to observe how architectural tweaks like dropout, activation functions, attention pooling, and residual connections affect performance. Below is a concise summary:

GCN Variations

- **GCN_V1_Basic:** No dropout, ReLU, mean pooling, no residual or batch norm.
- **GCN_V2_Dropout_LeakyReLU:** 30% dropout, LeakyReLU, mean pooling.
- **GCN_V3_Attn_Residual_BN:** 30% dropout, LeakyReLU, `attention` pooling, residual connections, and batch norm.

GIN Variations

- **GIN_V1_Basic:** Similar to above but using GINConv layers without edge features.
- **GIN_V2_Dropout_LeakyReLU:** 30% dropout, LeakyReLU, mean pooling.
- **GIN_V3_Attn_Residual_BN:** 30% dropout, LeakyReLU, attention pooling, with residual and batch norm.

GINE Variations

- **GINE_V1_Basic:** Adapted GINConv layers using edge features.
- **GINE_V2_Dropout_LeakyReLU:** 30% dropout, LeakyReLU, mean pooling.
- **GINE_V3_Attn_Residual_BN:** 30% dropout, LeakyReLU, attention pooling, with residual and batch norm.

GAT Variations

- **GAT_V1_Basic:** 4 attention heads, no dropout, ReLU, mean pooling.
- **GAT_V2_Dropout_LeakyReLU:** 8 attention heads, 30% dropout, LeakyReLU, mean pooling.
- **GAT_V3_Attn_Residual_BN:** 8 attention heads, 30% dropout, LeakyReLU, attention pooling, residual, batch norm.

GraphSAGE Variations

- **SAGE_V1_Basic:** No dropout, ReLU, mean pooling.
- **SAGE_V2_Dropout_LeakyReLU:** 30% dropout, LeakyReLU, mean pooling.
- **SAGE_V3_Attn_Residual_BN:** 30% dropout, LeakyReLU, attention pooling, residual, batch norm.

In all cases, we keep the **hidden_dim** at 64 and **out_channels** at 1 (for regression). The `Models.ipynb` notebook automates these experiments and tracks training/validation loss across epochs. While all the models were not run on the same machine, all the results are stored in the expermints folder on git.

3.6 Evaluation Metrics and Logging

We evaluate our models using three key metrics:

- **MSE (Mean Squared Error)**: This is our primary loss function during training. Given N samples with ground-truth values y_i and predictions $\hat{y}_i$, the MSE is defined as

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \hat{y}_i\right)^2.$$

  Lower values indicate better predictive accuracy. MSE heavily penalizes large errors (outliers).

- **MAE (Mean Absolute Error)**: Often used for interpretability, since it measures the average magnitude of errors in the same units as the target. Formally,

$$\text{MAE} = \frac{1}{N}\sum_{i=1}^{N}\left| y_i - \hat{y}_i\right|.$$

  MAE is less sensitive to large outliers than MSE, making it easier to see how far, on average, predictions deviate from true values.

- R^2 **Score (Coefficient of Determination)**: Shows how well the model's predictions correlate with the ground truth, relative to a simple baseline of always predicting the mean. Formally,

$$R^2 = 1 - \frac{\sum_{i=1}^{N}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{N}(y_i - \bar{y})^2},$$

  where $\bar{y} = \frac{1}{N}\sum_{i=1}^{N} y_i$ is the mean of the ground-truth values. An R^2 of 1.0 indicates perfect predictions, 0 means the model is on par with simply predicting $\bar{y}$, and negative values suggest the model is worse than that simple baseline.

All metrics are logged during training and validation phases. After training, we compute Test Loss $=$ MSE$(\hat{y}, y)$, MAE, and R^2 on the test split to compare model performance. The following code snippet (Listing 3) demonstrates how we evaluate and collect predictions for further analysis:

```python
# src/train.py
@torch.no_grad()
def evaluate(model, dataloader, criterion, device):
    model.eval()
    total_loss = 0
    for batch_data in dataloader:
        batch_data = batch_data.to(device)
        out = model(
            x=batch_data.x.float(),
            edge_index=batch_data.edge_index,
            batch=batch_data.batch,
            edge_attr=batch_data.edge_attr.float() if batch_data.
                edge_attr is not None else None
        ).squeeze(-1)

        loss = criterion(out, batch_data.y.float())
        total_loss += loss.item() * batch_data.num_graphs

    return total_loss / len(dataloader.dataset)
```

Listing 3: Evaluation function.

```python
# src/train.py
def predict(model, dataloader, device):
    model.eval()
    all_preds, all_targets = [], []
    for batch_data in dataloader:
        batch_data = batch_data.to(device)
        out = model(...).squeeze(-1)
        all_preds.append(out.detach().cpu())
        all_targets.append(batch_data.y.cpu())

    return torch.cat(all_preds), torch.cat(all_targets)
```

Listing 4: Collect predictions for plotting and error analysis.

After having run our experiments, we choose our best performing model to be run on the whole dataset. We expect the `GINE` with additional parameters to be our best performer as it has access to `edge_attr` and also it is less prone to over fitting.

4 Results

In this section, we present and discuss the performance of our *15* GNN variations across several metrics, visualized in five main comparative plots. We first start by examining the training and validation loss through training across 3000 epochs.

4.1 Training and Validation Loss Curves

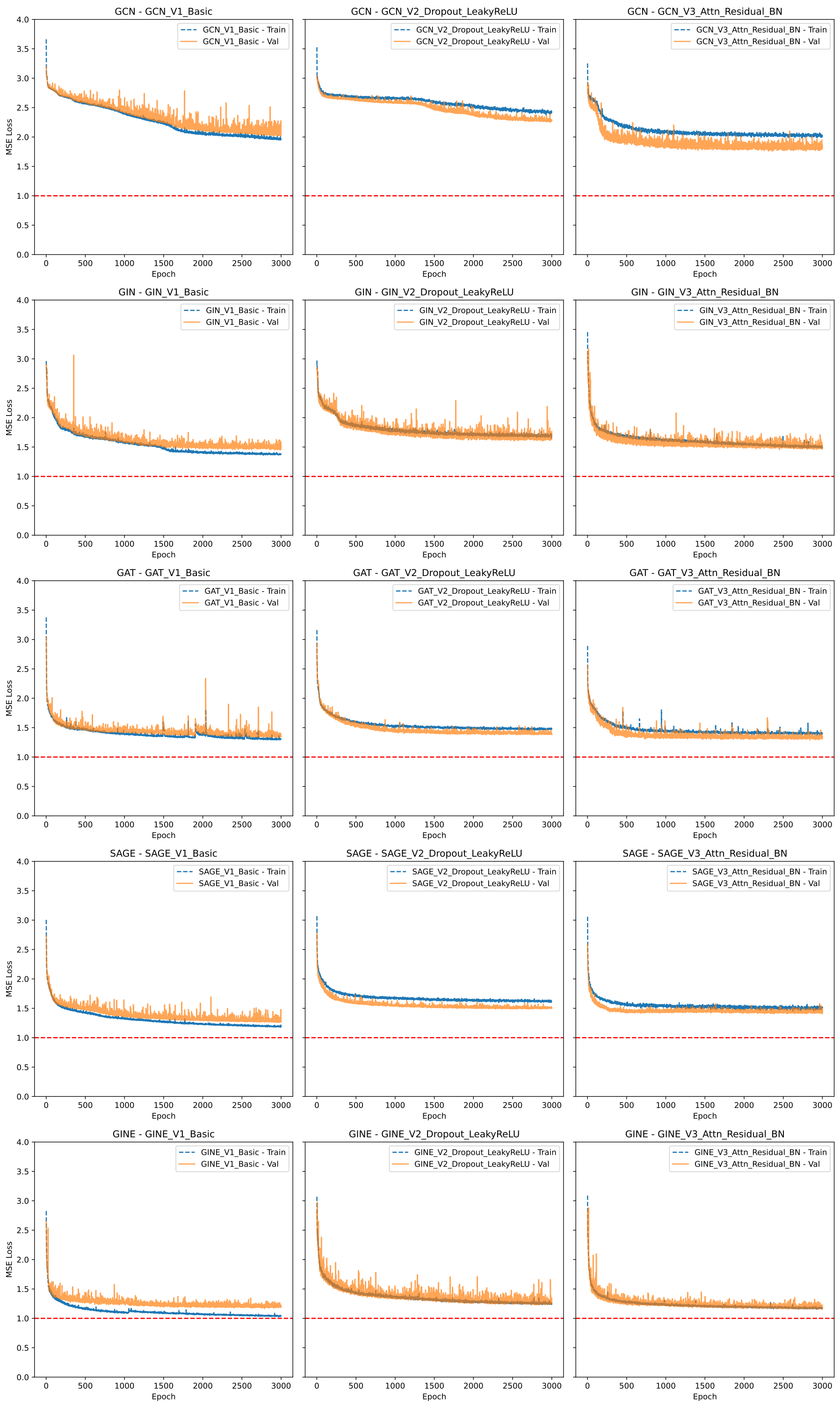


Figure 12: Comparison of MSE loss during training (blue curve) and validation (orange curve) for each of the 15 models across 3000 epochs, including the newly added GINE variants. Each sub-plot corresponds to a particular architecture configuration (e.g., `GCN_V1_Basic`, `GCN_V2_Dropout_LeakyReLU`, ..., `GINE_V3_Attn_Residual_BN`).

Figure 12 presents the mean squared error (MSE) loss trajectories for both training and validation across all 3,000 epochs. For ease of reference, a red horizontal line is drawn at $\texttt{MSE} = 1$.

Key observations include:

- **Overall Convergence:** All models exhibit a downward trend over 3,000 epochs, indicating effective learning.
- **Basic Model (V1):** The basic GCN model displays the highest MSE loss in both training and validation among all variations. Notably, in V3, the validation loss dips below the training loss, which is unusual but can occur due to small data splits or noise. Overall, the basic GCN model maintains an MSE above 2.
- **Variations (V2, V3):** These typically show slightly higher training loss, suggesting that they are less prone to overfitting.
- **Attn + Residual + BN Variations (V3):** Across architectures (GCN, GIN, GAT, GraphSAGE, and GINE), the V3 configurations converge to lower MSE and often exhibit more stable final epochs.
- **Edge-Attribute Awareness (GINE):** By incorporating bond attributes into message passing, GINE generally achieves faster convergence and lower validation loss than many edge-agnostic models. Notably, GINE V3 attains the lowest MSE among all variants.

4.2 Test MAE Comparison

Next we examine the MAE loss across all models

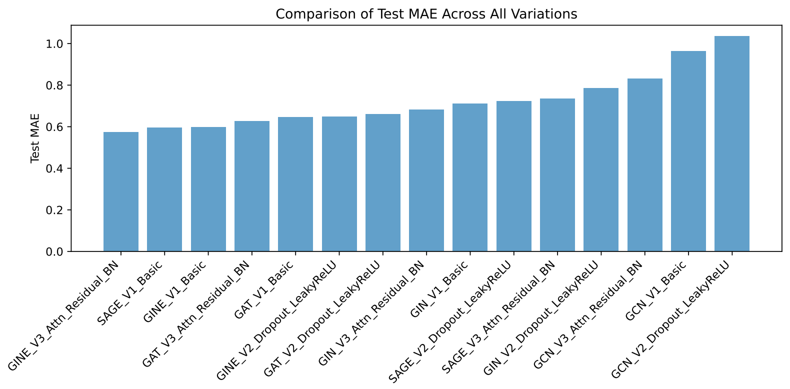


Figure 13: Comparison of final Test MAE (mean absolute error) for all 15 model variations (including the three GINE variants). Lower MAE values indicate more accurate predictions on the ZINC test set.

We plot the final test **MAE** of each model in Figure 13. Notable trends include:

- **GINE with Edge Features Tends to Excel:** The GINE variants exhibit noticeably lower MAE, likely because the bond attributes they incorporate provide critical chemical information that enriches the model's representations.
- **Comparisons to Non-Edge Models:** While GIN, GAT, and GraphSAGE remain strong contenders, GINE frequently outperforms them by leveraging explicit bond-type data. In contrast, the basic GCN models record higher MAE values, echoing the trends observed in the MSE curves.

4.3 R^2 Performance

We then examine the R^2 metric to gauge how much of the variability in y is explained by our models.

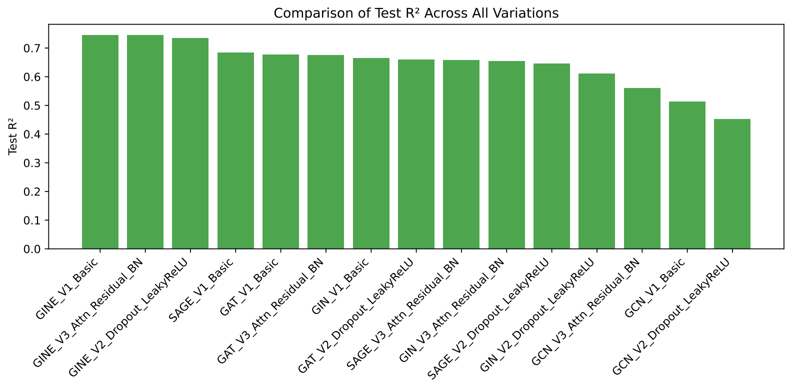


Figure 14: Comparison of the final test R^2 metric for each of the 15 models. Higher R^2 indicates tighter alignment between predictions and ground truth.

Figure 14 presents the coefficient of determination (R^2) for each model. An R^2 value approaching 1.0 implies that the predictions align closely with the true targets. Key observations include:

- **GINE_V3_Attn_Residual_BN Achieves Top Scores:** By integrating both node and edge features, in addition to attention, residual connections, and batch normalization, this configuration often attains the highest R^2 in the group, hovering around 0.75.
- **Other High Performers:** Models such as GIN_V3, GraphSAGE_V3, and selected GAT variants also demonstrate robust R^2 values, although they generally lag slightly behind the best GINE settings.
- **Basic GCN Model:** The basic GCN model reaches an R^2 of around 0.50, which is relatively low for our prediction goals.

4.4 Predicted vs. True Values

We then run a visual inspection of the predicted and actual values across all models.

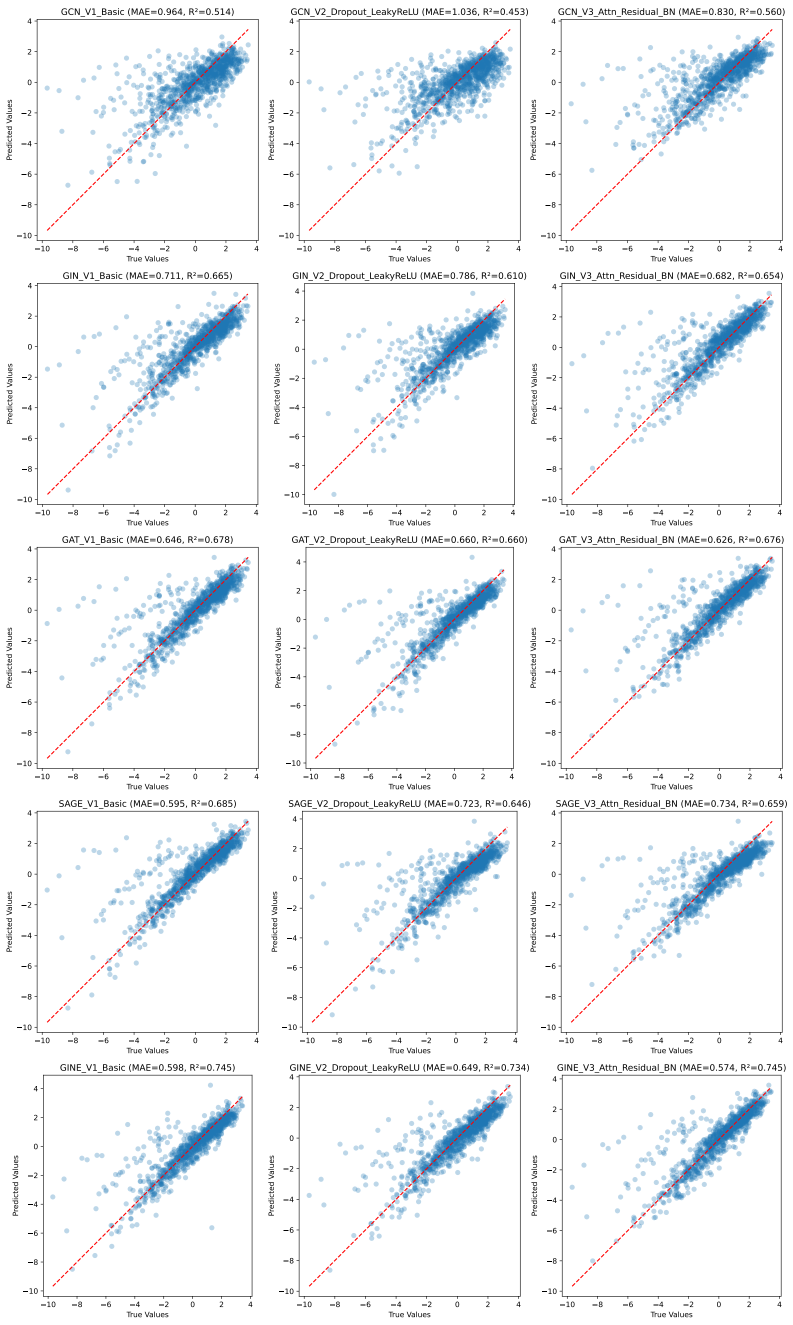


Figure 15: Scatter plots comparing predicted vs. true values on the test set for all 15 models. Each sub-figure includes its MAE and R^2 for quick reference. The red dashed line indicates the ideal $y = x$ identity line.

Figure 15 shows the predicted vs. actual values for each model on the ZINC test set:

- **Near-Diagonal Clusters:** GINE_V3 and other strong variants show predictions closely aligned with the red dashed line, reflecting high accuracy.

- **Residual + BN Effects:** V3 versions of GCN, GIN, GAT, and SAGE typically exhibit lower scatter than their Basic (V1) or Dropout-only (V2) counterparts.
- **Model Underprediction:** We notice that all models tend to underestimate some of the y values, while some models are better than others, this pattern is consistent across all models with varying degreees. This could be due to the fact that the prior distribution is very left skewed.

4.5 Distribution of Prediction Errors

We then examine the distribution of errors

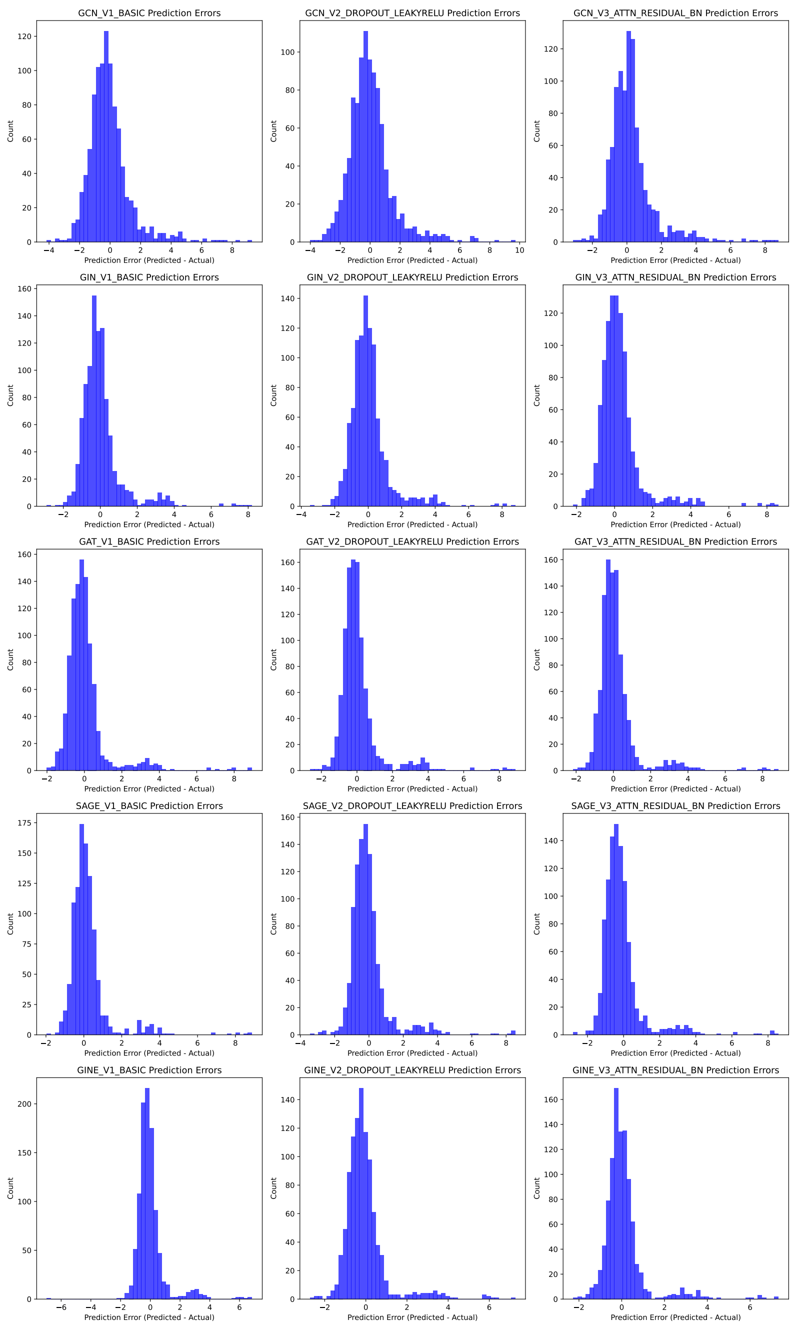


Figure 16: Histogram of test-set prediction errors (Predicted − Actual) for the 15 models, including the three new GINE variants. Taller bars around zero indicate more consistent predictions.

Figure 16 shows the distribution of prediction errors across the test set:

- **Tight Peaks for GINE:** The GINE variants exhibit sharp, high peaks near zero, confirming their lower MAE and strong predictive fidelity.

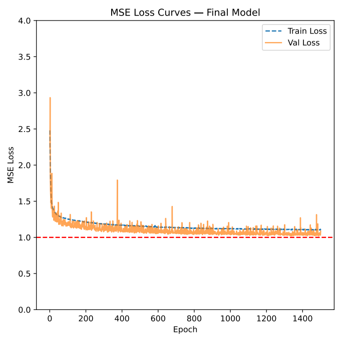
- **Reduced Tails in V3 Models:** The addition of attention pooling, residual connections, and batch normalization narrows the distribution of errors for many architectures, as evident in fewer large outlier errors.

- **Comparisons with Non-Edge Models:** Basic GCN or GIN haveslightly longer tails, suggesting they occasionally mispredict certain molecules that rely heavily on bond-type distinctions.

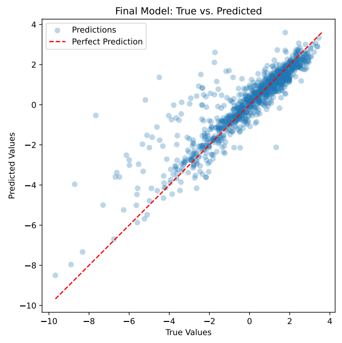
Over all, GINE(V3) is our best perfomer and we use it to train on the whole data set.

4.6 GINE-V3 Full Data Set Training:

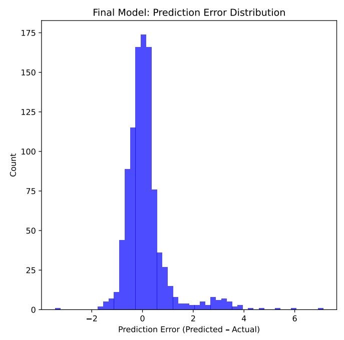
We train our **GINE-V3 model** on the entire dataset for $1,500$ epochs only due to resource limitations, expecting improved performance due to the additional training examples that allow the model to better learn underlying patterns.



(a) MSE Loss Curves



(b) True vs. Predicted



(c) Error Distribution

Figure 17: Visualizations of the final model: (a) training/validation MSE loss curves, (b) scatter plot of true vs. predicted values, and (c) histogram of prediction errors.

As shown in Figure 17, the MSE loss hovers near 1.0, representing an improvement over previous models. The scatter plot of true vs. predicted y values shows tighter clustering around the $x = y$ diagonal with fewer underpredicted samples. Furthermore, the error distribution histogram displays a shorter right tail, suggesting fewer large positive errors.

We also observe a test MAE of 0.513 and an R^2 of 79.6%, which is approximately five percentage points higher than the same model trained on a smaller subset of the dataset.

4.7 Summary of Key Findings

Below we look at the numerical results for our models:

Table 2: Summary of final test metrics for the 15 model variants on ZINC trained on a training dataset of size 10000.

Model Variant	**Test MSE**	**Test MAE**	**R^2**	**Time/Epoch (s)**	**Machine**	**Dataset**
GCN_V1_Basic	1.98	0.96	0.51	0.93	CPU	Sample
GCN_V2_Dropout_LeakyReLU	2.23	1.04	0.45	1.13	CPU	Sample
GCN_V3_Attn_Residual_BN	1.79	0.83	0.56	1.35	CPU	Sample
GIN_V1_Basic	1.36	0.71	0.66	0.72	CPU	Sample
GIN_V2_Dropout_LeakyReLU	1.59	0.79	0.61	0.92	CPU	Sample
GIN_V3_Attn_Residual_BN	1.41	0.68	0.65	1.13	CPU	Sample
GAT_V1_Basic	1.31	0.65	0.68	2.28	GPU	Sample
GAT_V2_Dropout_LeakyReLU	1.38	0.66	0.66	1.62	GPU	Sample
GAT_V3_Attn_Residual_BN	1.32	0.63	0.68	1.77	GPU	Sample
SAGE_V1_Basic	1.28	0.59	0.68	0.98	CPU	Sample
SAGE_V2_Dropout_LeakyReLU	1.44	0.72	0.65	1.19	CPU	Sample
SAGE_V3_Attn_Residual_BN	1.39	0.73	0.66	1.39	CPU	Sample
GINE_V1_Basic	1.04	0.60	0.75	1.26	CPU	Sample
GINE_V2_Dropout_LeakyReLU	1.08	0.65	0.73	1.30	CPU	Sample
GINE_V3_Attn_Residual_BN	1.04	0.57	0.74	1.52	CPU	Sample
GINE_V3_Attn_Residual_BN	**0.80**	**0.51**	**0.79**	**31.49**	**CPU**	**Full**

From the table above and from the results discussed ealier we can make the following conclusions.

- **Edge-aware GNNs Excel:** Models leveraging explicit bond attributes (i.e., GINE) consistently surpass edge-agnostic architectures such as GCN, GIN, GAT, and GraphSAGE. Among the GINE variations, `GINE_V3_Attn_Residual_BN` achieves the best overall performance, with the highest R^2 and lowest MAE across most runs.
- **Advanced Techniques Improve Generalization:** Features like dropout, attention-based pooling, residual connections, and batch normalization each contribute to lower loss and more stable training. In particular, the "V3" variations, which combine all of these strategies, converge faster, handle outliers better, and ultimately reach higher R^2 scores.
- **Larger Training Sets Yield Better Results:** Training `GINE_V3` on the full dataset (rather than a subset) further improved performance, reducing the final MSE closer to 1.0 and pushing R^2 up to nearly 80%. The tighter scatter of predictions around the diagonal and a shorter right tail in the error histogram underscore the benefit of additional data.

Overall, these findings confirm that *edge-aware* GNN models, such as GINE, hold a clear advantage in predicting molecular properties from the ZINC dataset. Furthermore, the synergy of attention pooling, residual connections, and batch normalization in the V3 configurations provides the strongest results to date, emphasizing how multiple architectural enhancements can yield state-of-the-art performance in graph-based chemistry applications.

5 Conclusion

In this report, we explored the use of **Graph Neural Networks (GNNs)** for molecular property prediction on the ZINC dataset, drawing on fundamental concepts from graph theory and extending them through modern deep-learning techniques. Our investigations covered multiple GNN architectures—*GCN*, *GIN*, *GINE*, *GAT*, *GraphSAGE*, and an optional *Graph Transformer*—each configured under various design choices (dropout, attention pooling, residual connections, and batch normalization).

Key Takeaways:

- *Edge-aware GNNs* (notably `GINE`) *consistently outperform* edge-agnostic models on this molecular dataset, highlighting the importance of explicitly incorporating bond features into the message-passing process.
- Employing *advanced techniques* such as attention-based pooling, skip connections, and batch normalization (*V3* variants) systematically improves generalization, stability, and convergence speed for all evaluated GNN families.

- *Training on larger subsets* of the ZINC dataset further boosts performance, exemplified by `GINE_V3`, which achieved an MSE near 1.0, a MAE of 0.513, and an R^2 approaching 80%. This underscores the value of comprehensive data coverage when modeling complex chemical structures.
- Basic models like GCN without additional features (i.e., *V1* variants) tend to *underfit* or converge more slowly and exhibit higher residual errors, especially when bond-specific distinctions play a central role in property determination.

Implications and Future Work: Our findings indicate that leveraging *edge attributes* (e.g., bond types) is vital for accurate property prediction in molecular graphs, and that combining multiple improvements (dropout, attention pooling, residual connections, batch normalization) helps overcome common challenges like over-smoothing, overfitting, and unstable training. Future directions may involve:

- Investigating **deeper GNN stacks** (beyond two layers) while mitigating over-smoothing through advanced residual or gating schemes.
- Exploring **Graph Transformer** variants that relax the reliance on explicit neighborhood definitions, enabling more global attention across the graph.
- Extending our models to **heterogeneous or larger-scale molecular datasets**, potentially involving 3D structures or additional chemical properties.

Overall, our experiments confirm that GNN-based approaches, and particularly those that incorporate edge features, are highly effective for molecular property prediction in chemistry applications. By uniting rigorous graph-theoretic foundations with the flexibility of deep learning, GNNs continue to show substantial promise for tasks ranging from drug discovery to broader material design. The results of this work suggest that carefully chosen architectures and sufficient training data are paramount in achieving state-of-the-art performance.

References

William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 1024–1034, 2017.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations (ICLR)*, 2018.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations (ICLR)*, 2019.

Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J. Kim. Graph transformer networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 11983–11993, 2019.

Jie Zhou, Guang Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 2:25–66, 2021.