Geometric Deep Learning - Graph Classification

Matan Birenboim
matan.birenboim@campus.technion.ac.il

Tal Peer
tal.peer@campus.technion.ac.il

Tsuf Bechor
tzufbechor@campus.technion.ac.il

November 2024

Introduction

Graph-structured data appears in diverse domains, including social networks, molecular biology, and knowledge graphs, posing unique challenges for machine learning. Graph Neural Networks (GNNs) have emerged as a powerful approach for tasks like graph classification, regression, and clustering. Among GNN architectures, spectral methods provide a robust mathematical framework for learning from graph signals, leveraging the graph Laplacian to encode structure and target specific frequencies [Bal+21]. In **molecular graphs**, node features (atoms) and edge attributes (bonds) using low-pass filters helps in capturing global chemical properties, which are often correlated across entire molecules. On the other hand, care must be taken to avoid excessive smoothing, which can lead to over-smoothing and the loss of discriminative features. In this project, we investigated the performance of various GNN architectures, focusing on **ChebConv**, a spectral-based convolutional layer that approximates graph convolutions using Chebyshev polynomials. By avoiding expensive eigenvalue decompositions, ChebConv efficiently captures localized and global graph information. Experiments were conducted to evaluate the impact of architectural variations, including the number of convolution layers, polynomial orders, and pooling strategies. Comparative analyses of ChebConv and other GNNs, such as GCN, GraphSAGE, and GAT, provide insights into the trade-offs between performance, efficiency, and scalability. [1]

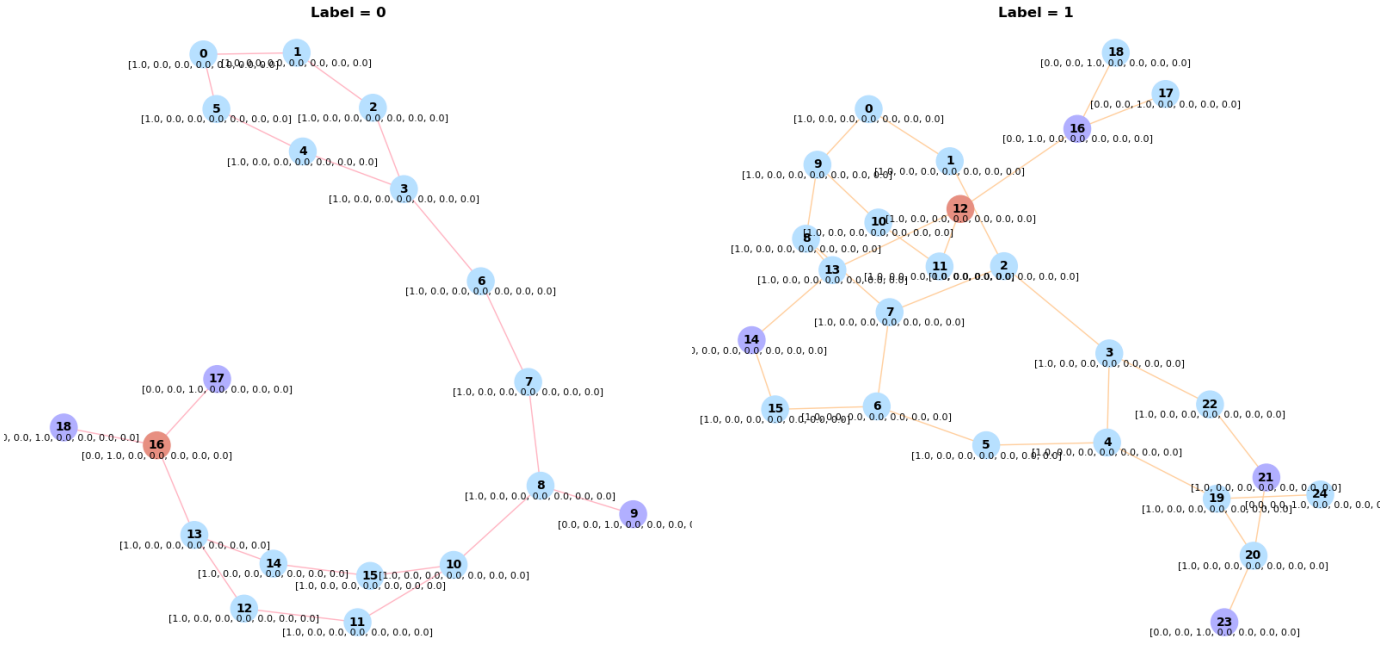


Figure 1: Samples graphs. each node colored by its node feature

[1] Project's GitHub Repository - *github.com/GraphClassification*

Methodology

Dataset EDA

We were supplied with 3 datasets - Train, Validation and Test. Since there is no previous information regarding the data, we performed Explantory Data Analysis 1. Each graph is described by:

- **Node features:** One-hot encoded vectors of size 7.
- **Edge attributes:** One-hot encoded vectors of size 4.
- **Graph-level descriptors:** The number of nodes, the number of edges, and other graph-specific properties.
- **Graph label:** Label of 0/1, the ground-truth for the classification task.

Hence, the graphs are heterogeneous, which means that nodes and edges are imbued with types. Given that, we can partition the set of nodes into disjoint sets:

$$V = V_1 \cup V_2 \cup \cdots \cup V_k \quad \text{where} \quad V_i \cap V_j = \emptyset, \forall i \neq j.$$

Edges in heterogeneous graphs generally satisfy constraints according to the node types, most commonly the constraint that certain edges only connect nodes of certain types, i.e.,

$$(u, \tau_i, v) \in E \rightarrow u \in V_j,\ v \in V_k.$$

After visualizing the graphs, we saw they exhibit characteristics similar to those in molecular features, where nodes features seems to indicate the types of the atoms constructing the molecule, and edges reflecting the chemical bonds between the atoms.

Graph ID	Nodes	Edges	Average Degree	Node/Edge Types
0	17	38	2.24	Nodes: {'type 0': '14 atoms', 'type 1': '1 atom', 'type 2': '2 atoms'} Edges: {'type 0': '32 edges', 'type 1': '4 edges', 'type 2': '2 edges'}
1	13	28	2.15	Nodes: {'type 0': '9 atoms', 'type 1': '2 atoms', 'type 2': '2 atoms'} Edges: {'type 0': '22 edges', 'type 1': '4 edges', 'type 2': '2 edges'}
2	13	28	2.15	Nodes: {'type 0': '9 atoms', 'type 1': '2 atoms', 'type 2': '2 atoms'} Edges: {'type 0': '22 edges', 'type 1': '4 edges', 'type 2': '2 edges'}
3	19	44	2.32	Nodes: {'type 0': '16 atoms', 'type 1': '1 atom', 'type 2': '2 atoms'} Edges: {'type 0': '34 edges', 'type 1': '8 edges', 'type 2': '2 edges'}
4	11	22	2.00	Nodes: {'type 0': '6 atoms', 'type 1': '1 atom', 'type 2': '2 atoms', 'type 3': '2 atoms'} Edges: {'type 0': '12 edges', 'type 1': '8 edges', 'type 2': '2 edges'}

Table 1: Sample of graph statistics summary for train set. We assume each type of node representing a specific atom from the set of 7 distinct atoms. We also assume edges representing molecular bonds, single, double,triple and aromatic.

Bond Type	Label 0 - Mean	Label 1 - Mean	Label 0 - Min/Max	Label 1 - Min/Max
1	15.372549	30.484	0.00 , 24.00	0.00 , 60.00
2	9.803922	10.868687	4.00 , 20.00	4.00 , 24.00
3	3.803922	3.595960	2.00 , 10.00	2.00 , 12.00
4	0.00	2.00	0.00 , 0.00	0.00 , 2.00

Table 2: Bond types - Mean and Min/Max per Label

In praticular, edges with type 0 are the most common, but 9 graphs in the train dataset (total of 150) contains 0 edges with this kind of bonds. We can see that 8 of them are label 0, and all of them contain bonds from type 1 and 2.

Models

Background - Graph Convolutional Networks and Connections to Message Passing

Kipf and Welling [2016a] built on the notion of graph convolutions to define one of the most popular GNN architectures, commonly known as the Graph Convolutional Network (GCN). The key insight of the GCN approach is that we can build powerful models by stacking very simple graph convolutional layers.

A basic GCN layer is defined in Kipf and Welling [2016a] as:

$$H^{(k)} = \sigma\left(\tilde{A}H^{(k-1)}W^{(k)}\right),$$

where:

$$\tilde{A} = (D+I)^{-\frac{1}{2}}(I+A)(D+I)^{-\frac{1}{2}}$$

is a normalized variant of the adjacency matrix (including self-loops), $W^{(k)}$ is a learnable parameter matrix, and σ is a non-linearity.

This model was initially motivated as a combination of a simple graph convolution (based on the polynomial $I+A$), with a learnable weight matrix and a non-linearity. In general, if we consider combining a simple graph convolution defined via the polynomial $I+A$ with non-linearities and trainable weight matrices, we recover the basic GNN:

$$H^{(k)} = \sigma\left(AH^{(k-1)}W^{(k)}_{\text{neigh}} + H^{(k-1)}W^{(k)}_{\text{self}}\right),$$

where $W^{(k)}_{\text{neigh}}$ and $W^{(k)}_{\text{self}}$ are trainable weight matrices for neighbor and self-contributions, respectively.

In other words, a simple graph convolution based on $I+A$ is equivalent to aggregating information from neighbors and combining it with information from the node itself. Thus, the notion of message passing corresponds to a simple form of graph convolutions combined with additional trainable weights and non-linearities.

We conducted experiments for different known models.

- **GCN** The baseline model, since known for its simplicity and efficiency in tasks involving homogeneous graphs. it employs spectral convolutions to aggregate information from neighboring nodes. It relies on the graph Laplacian to encode structural information, and its global mean pooling enables effective graph-level representation.
- **GrpahSAGE** Instead of training individual embeddings for each node, SAGE convolution learn a function that generates embeddings by sampling and aggregating features from a node's local neighborhood. Its ability to incorporate diverse aggregation functions (mean, max, or LSTM-based) allows it to adapt to different datasets.
- **GAT** GAT is known for applications requiring attention to graph topology, and address the limitations of uniform neighborhood aggregation by introducing attention mechanisms. This enables the model to focus on the most relevant neighbors during feature aggregation. GATs uses multi-head attention enhances the model's stability and expressiveness, particularly for heterogeneous graphs.
- **EGCConv** EGC incorporates edge-specific features into the convolution operations, makes it suitable for tasks where edge properties play a role.
- **ChebConv** ChebConv leverage spectral graph theory to approximate convolutions using Chebyshev polynomials. This reduces computational costs while preserving essential graph structural information. Chebyshev polynomials can be computed fast by its recursive definition, and avoid direct computation of the Laplacian eigenvalues. This way it achieves computational efficiency while maintaining a global view of the graph structure.

Connection Between Chebyshev Polynomials and Graph Filters Graph filters are mathematical operators designed to modify graph signals in the spectral domain. Chebyshev polynomials provide an efficient means to implement such filters by approximating the spectral filtering operation without requiring costly computation of the Laplacian eigenvectors. Specifically:

Graph Laplacian Basis and Chebyshev Polynomials The process of transforming graph signals into the spectral domain is done by computing the eigenvectors of the graph Laplacian, which defines the graph Fourier basis. Chebyshev polynomials approximate filters $h(\lambda)$ in the spectral domain as:

$$h(\lambda) = \sum_{k=0}^{K-1} \theta_k T_k(\lambda),$$

where $T_k(\lambda)$ are Chebyshev polynomials, λ are the eigenvalues of the Laplacian, and θ_k are learnable coefficients.

By directly applying $T_k(\tilde{L})$ to the graph Laplacian $\tilde{L}$, Chebyshev filters bypassing the computationally expensive eigenvector decomposition.

Higher-Order Polynomials Capture Complex Dependencies Higher-order Chebyshev polynomials expand the receptive field of graph convolutions. Each polynomial order k corresponds to aggregating information from k-hop neighbors. Higher k includes features from nodes that are farther away in the graph. By stacking multiple polynomial terms $T_k(\tilde{L})$, the model captures both local and global structures by incorporating contributions from multiple graph scales.

Higher-order terms $T_k(\lambda)$ target finer details in the graph's frequency spectrum, allowing the model to distinguish subtle variations in graph structure. Additionally, complex structural dependencies, such as cycles (as seen in molecule graphs) or hierarchical subgraphs, become accessible as the receptive field grows.[2]

Experiments

The experiments were divided into 3 phases:

- **Baseline Comparison:** All models were trained with default settings to establish baseline performance. The initial hyperparameter settings included a random search over the following parameters (Table 3):

Hyperparameters	**Tested Values**
Epochs	[80, 125, 150, 200]
Batch Sizes	[24, 32, 48, 64]
Hidden Channels	[16, 24, 48, 64]
Learning Rates	[5e-4, 1e-3, 5e-3, 1e-2]
Weight Decay	[3e-3, 5e-5]

Table 3: Hyperparameters Tested During Baseline Comparison

- **Focused Exploration of ChebConv:** Based on the baseline results, ChebConv emerged as the best-performing model. Subsequent experiments concentrated on optimizing its architecture and hyperparameters. The explored configurations are detailed in (Table 4).

Hyperparameters	**Tested Values**
Polynomial Order (K)	[2,3,4]
Number of Convolution Layers	[2,3,5,7,9,11,13]
Number of Linear Layers	[1,2,3]
Pooling Strategies	['mean', 'add', 'max']

Table 4: Different Architectures Tested During ChebConv Finetune

- **Hyperparameters tuning for ChebConv:** We performed subsequent experiments for further-evaluations, by dividing the train set to train-test split and check for the confusion metrics terms(Table 5).

[2][HWW24] mentioned that GCN is a simplified version of ChebConv with only the first two Chebyshev polynomials and that ChebConv has more expressive capability than GCN in theory.

Hyperparameters	Tested Values
Polynomial Order (K)	[2,3,4]
Number of Convolution Layers	[2,3,5,7,9,11,13]
Pooling Strategies	['add', 'max']
Batch Sizes	[24, 32, 48, 64]
Hidden Channels	[16, 24, 48, 64]
Epochs	[90]
Number of Linear Layers	[1]
Learning Rates	[3e-3]
Weight Decay	[5e-5]

Table 5: Hyperparameters Tested During ChebConv Finetune

In the 1st and 2nd experiments phases, we train each model over the train set (150 graphs), and validate over the given validation set (19 graphs). For the 3rd experiment phase, we created splits of the original train-set to train-test sets, where 75% of the original train-set were used for train and the rest were used as test set. We used PyTorch Geometric DataLoader for data loaders instances and experimented different batch sizes. Wee leveraged the Adam optimizer and the CrossEntropy loss objective for optimization. We saved the experiments configurations, performance metrics and sessions results to a .csv file for later analyzing the results, together with the accuracies and loss scores, performance metrics (e.g. min, max& mean gained accuracies, and the epoch where the max & min accuracies were reached) for later hypertune.

Results

Performance Analysis Across Models

The experimental results align with the mathematical foundations of the models. ChebConv architecture, leveraging Chebyshev polynomials for spectral graph convolutions, achieved high validation accuracies consistent with its theoretical strength in capturing multi-hop dependencies and complex graph structures. Similarly, EGC's ability to model edge-specific features was reflected in its competitive performance. GraphSAGE demonstrated stable generalization, aligning with its inductive learning approach and scalability. GAT's use of attention mechanisms matched its observed ability to handle heterogeneous graph structures effectively. In contrast, GCN, while performing robustly, showed limitations in handling complex dependencies, consistent with its design for homogeneous graphs. Overall, the results confirm that the models performed as theoretically expected within their respective capabilities.

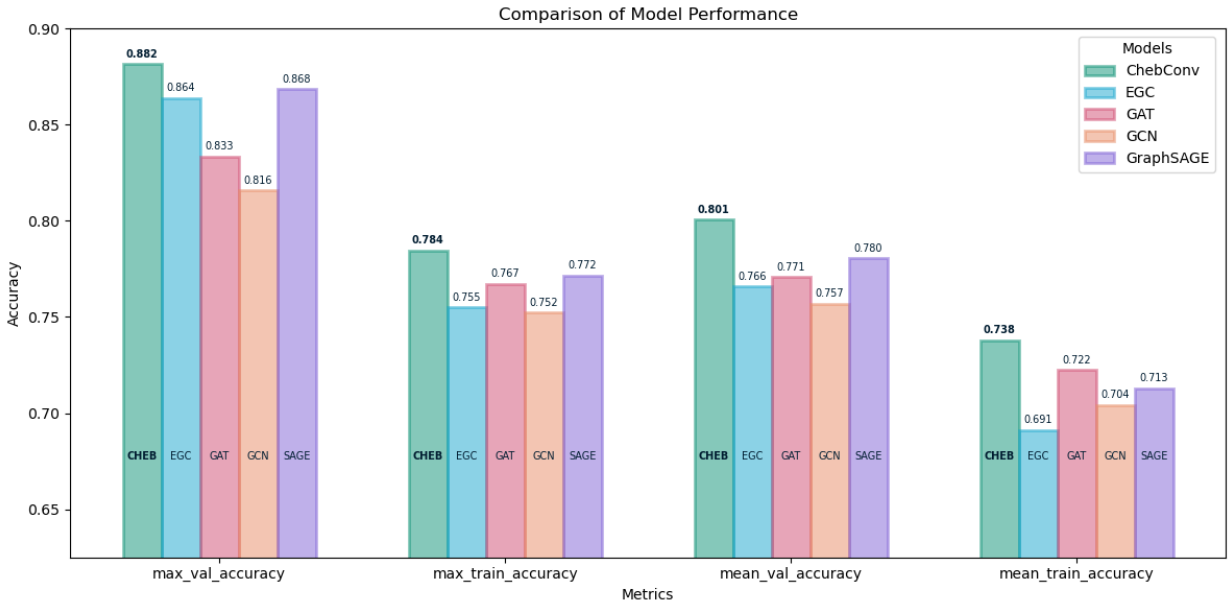


Figure 2: Combined analysis of tested models

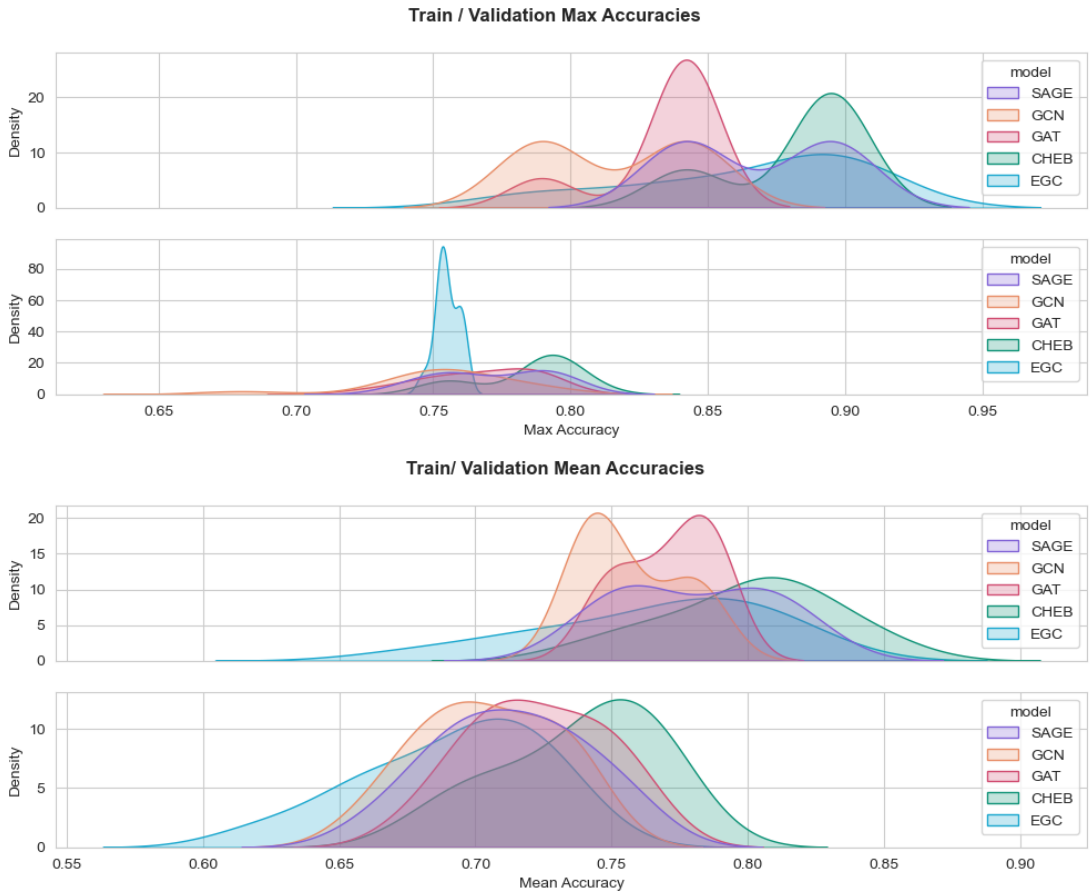


Figure 3: Combined analysis of ChebConv performance: (a) Maximum accuracy distribution, and (b) Mean accuracy distribution. In each figure, the upper subplot demonstrates the models performance over the **Validation set**, while the bottom subplot demonstrates the models performance over the **Train set**.

Experiments different Architecture for ChebConv models

For finding the optimal model configuration for ChebConv convolution layers, we initiated focused experiments with different configurations and explored architectural variations, including the number of convolution layers, polynomial orders, and pooling strategies. The results demonstrated that moderate-depth architectures (5-9 convolution layers) and lower polynomial orders ($k \in \{2, 3\}$), and "add" / "max" pooling methods outerperform other configurations. Adding more convolution layers allowed the model to capture multi-hop dependencies, but configurations with more than 9 layers showed diminishing returns due to over-smoothing. Among the pooling methods, "mean" pooling underperformed with lower validation accuracies and consistency, probably due to its disadvantage of demonstrating over-smoothing.

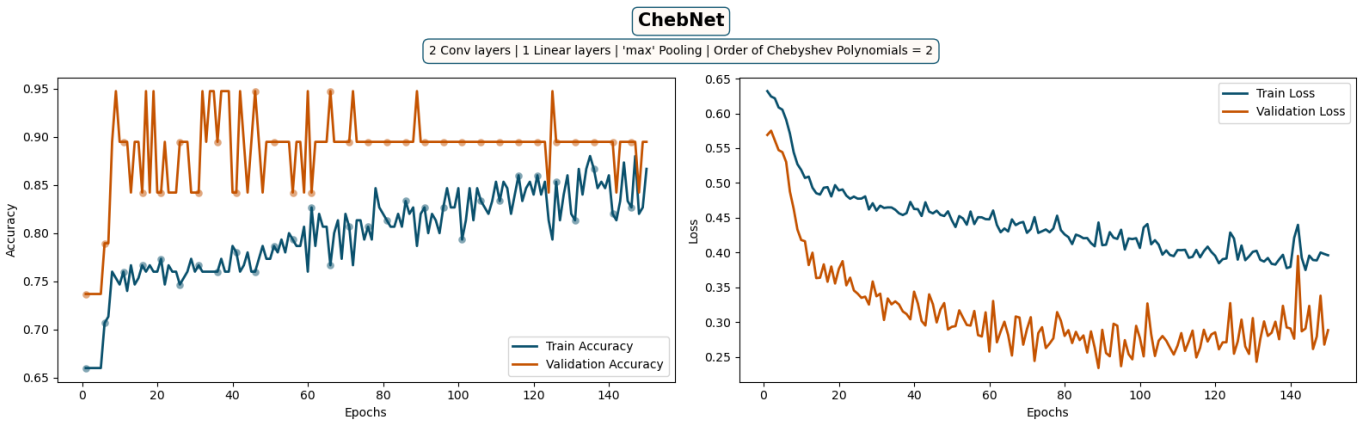


Figure 4: ChebNet model used for comparison with other models.

Early training converged typically around epoch 25, highlighted the rapid learning facilitated by Chebyshev filters, which efficiently approximate spectral convolutions. Overall, the results confirm the robustness and generalization ability of ChebConv while highlighting practical trade-offs in architectural design, such as the balance between depth and computational efficiency, and we decided to focus on ChebConv based architectures.

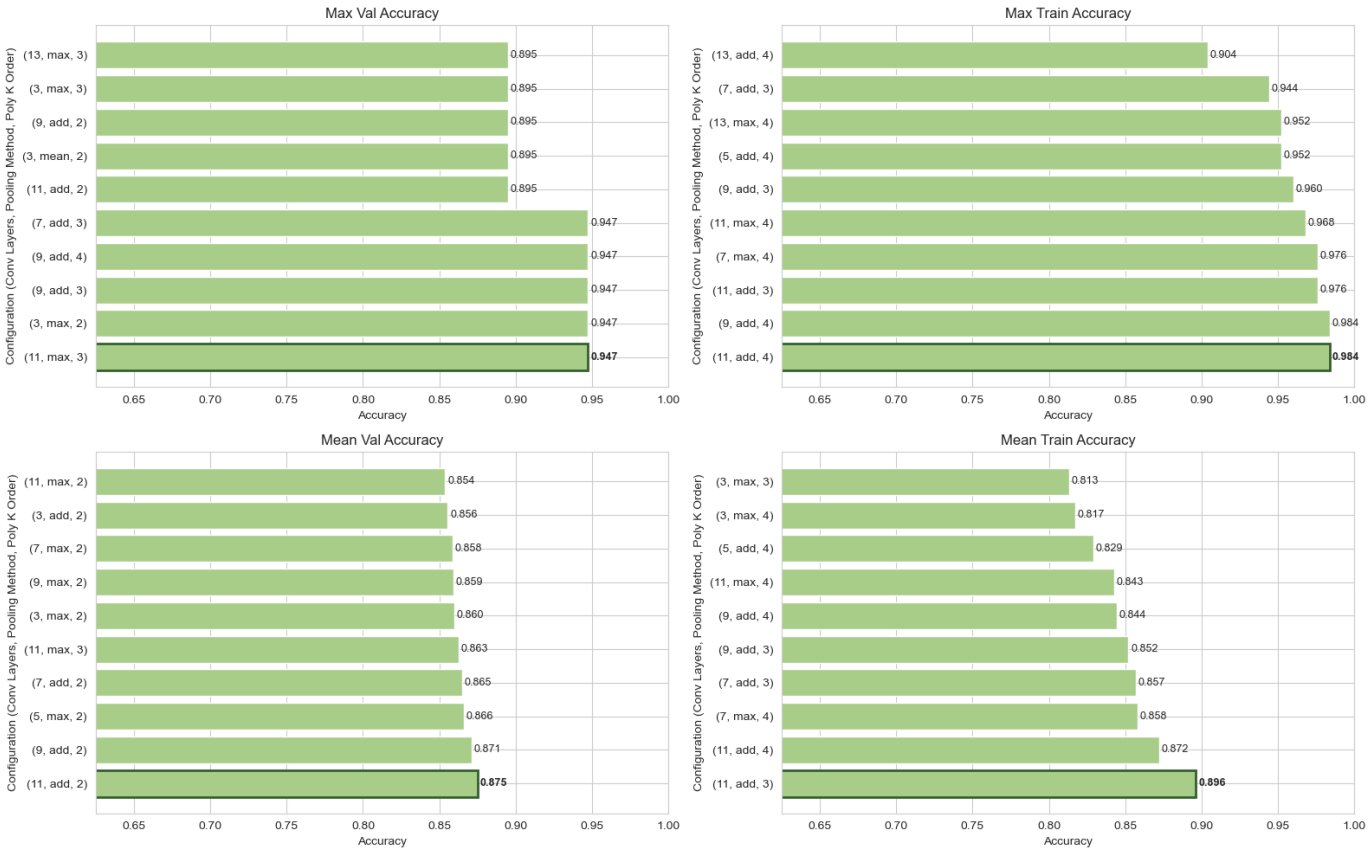


Figure 5: ChebConv Optimization

Discussion

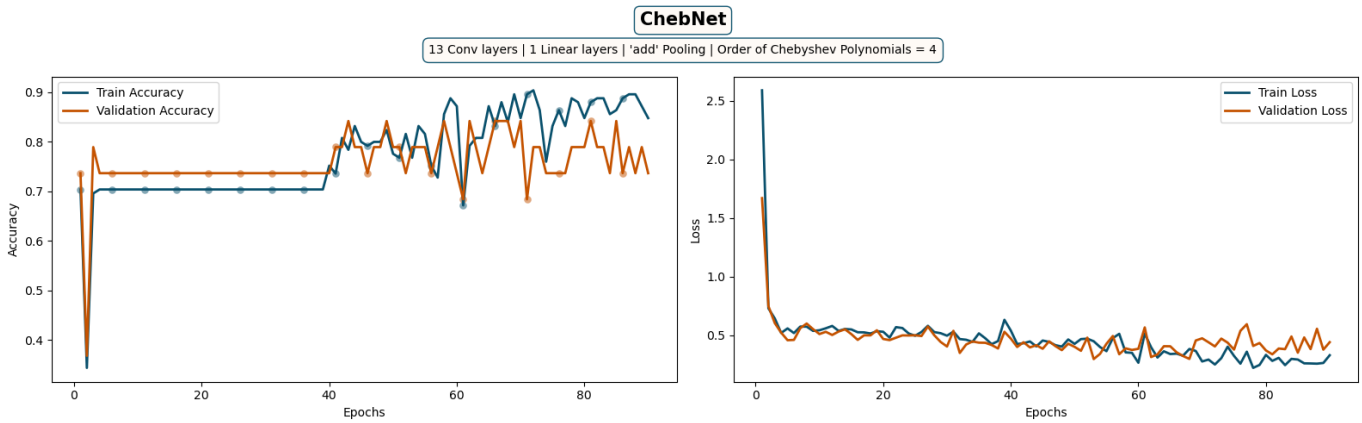
The evaluation results highlight the effectiveness of ChebConv architectures across various configurations. The evaluation highlights that configurations with moderate depth (e.g., 5 or 9 convolution layers) and higher polynomial orders (3) strike a balance between sensitivity, specificity, and precision. "Max" pooling methods excel at identifying dominant features and achieving high specificity, while "add" pooling provides robustness for sensitivity at the cost of more false positives, which performed better for Extremely deep models (e.g., 11-13 convolution layers). However, this kind of configurations were sensitive to different settings of hyperparameters, and suffered from over-smoothing which reduced their ability to generalize. These findings confirm the theoretical strengths of ChebNet but also emphasize practical trade-offs when selecting architectural components. Configurations with shallow networks ($\leq$ 5 layers) achieved lower False positive rates, indicating better specificity but at the cost of reduced expressiveness. On the other hand, deeper networks ($\geq$ 10 layers) exhibited signs of over-smoothing, as we mentioned previously.

Optimal Balance: Moderate Depth and Higher Polynomial Order / Higher Depth and Moderate Polynomial Order (5, 'max', 3), (13, 'add', 2) configurations achieved the best balance across all metrics, as can be viewed in Figure 6. Moderate depth (5–9 convolution layers) ensures a sufficiently wide receptive field to capture graph-level features without over-smoothing. A higher polynomial order (3) enhances the model's ability to extract multi-hop dependencies, which might be relevant for graphs with complex relationships, but overfitting observed for incorporating higher polynomial orders with the addition of linear layers.

Figure 6: Evaluation Metrics Comparison - ChebConv Architecture

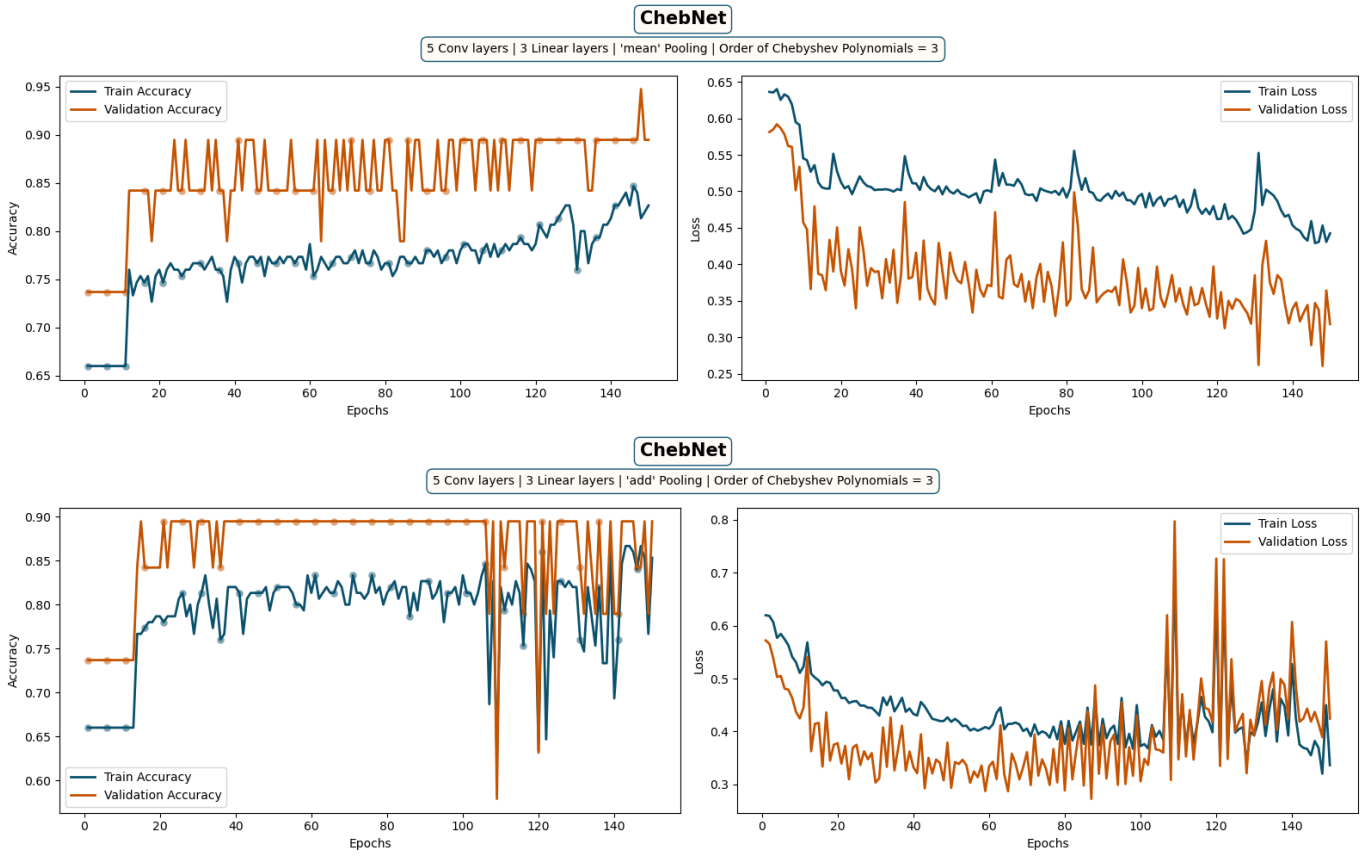
Different Depth Configurations

Deep configurations in general, suffered from reduced sensitivity (63.64% in some cases) and overall accuracy ($\tilde{6}4\%$). But 13 convolution layers and 'add' pooling and minimal order for the Chebyshev polynomials outerperformed other models with the right setting of hyperparameters.



Pooling methods

'max' pooling configurations ((5, max, 3), (9, max, 2)) consistently achieved high specificity (100%) and precision (100%) but occasionally at the expense of sensitivity. On the other hand, **'add' pooling** configurations ((3, add, 2), (7, add, 3)) Showed strong sensitivity (85%–92%) and balanced F1-scores but slightly lower specificity due to a higher rate of false positives. **An interesting result** was for the configuration of 7-conv layers, 'max' pooling and 2-order polynomials, achieved perfect sensitivity (100%) but with **0%** specificity. This indicates an extreme bias toward positive (label 0) predictions.



Polynomial Order k

The results demonstrates that deeper configurations required setting of higher k for stabilized performance. Also, setting early stopping led to better performance over the validation set, but the model seems to overfit the train set very fast (accuracy reaching to 1 and loss reaching 0 at epoch 20).

Conclusion Key findings demonstrate that ChebConv outer-performs known models performance, particularly with moderate depth (6-9 convolution layers), "add" pooling for 1-linear layer (or "max" pooling for 2-3 linear layers), and polynomial orders of 2 or 3. These configurations balance sensitivity and specificity, while deeper architectures face challenges from over-smoothing. It is important to note that different tasks would end up in considering different architectures. Finaly, our chosen model is ChebNet with 13 convolution layers with $k = 2$, 1 linear layer, 'add' pooling. The model is trained for 80 epochs, with learning rate of 0.003 and weight decay of 0.005. [3]

[3] all results files can be found in the git repo.

```
student@gdlgpu2023s-0022:~$ cd GeometricDeepLearning
student@gdlgpu2023s-0022:~/GeometricDeepLearning$ python main.py --seed=42

==================================================

Model = ChebNet ->
        Number of Convolution Layers = 13
        Number of Linear Layers = 1
        Hidden Channels = 32
        Order of Chebyshev Polynomials = 2
        Pooling Type = 'add'

        Total Epochs = 80
        Batch Size = 24
        Optimizer = torch.optim.Adam
        Criterion = torch.nn.CrossEntropyLoss
        Learning Rate = 0.003
        Weight Decay = 0.005


-> Train mean loss = 0.4930 | Validation mean loss = 0.3869
-> Train mean accuracy = 0.7558 | Validation mean accuracy = 0.8362
-> Train max accuracy = 0.8133 | Validation max accuracy = 0.8947


Predictions saved to predictions.txt
```

A Over-Smoothing as a Low-Pass Convolutional Filter

In Chapter 5, [Ham20] introduced the problem of over-smoothing in GNNs. This phenomena occures when, after too many rounds of message passing, the embeddings for all nodes become indistinguishable and lose meaningful information. We can now interpret over-smoothing through the lens of graph signal processing.

The key intuition is that stacking multiple rounds of message passing in a basic GNN is analogous to applying a low-pass convolutional filter, which smooths the input signal on the graph. To illustrate this, we simplify the basic GNN update equation (Equation 7.40) to:

$$H^{(k)} = A_{\text{sym}} H^{(k-1)} W^{(k)},$$

where $A_{\text{sym}} = D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$ is the symmetric normalized adjacency matrix. Compared to Equation 7.40, this model omits the non-linearity and the addition of "self" embeddings at each step, simplifying both mathematical analysis and numerical stability. This formulation is similar to the simple GCN approach proposed by Kipf and Welling [2016a], effectively averaging neighbor embeddings at each message-passing step.

After K rounds of message passing using this simplified update rule, the representation depends on the K-th power of the adjacency matrix:

$$H^{(K)} = A_{\text{sym}}^{K} X W,$$

where W is a linear operator, and X is the matrix of input node features.

Connection to Convolutional Filters

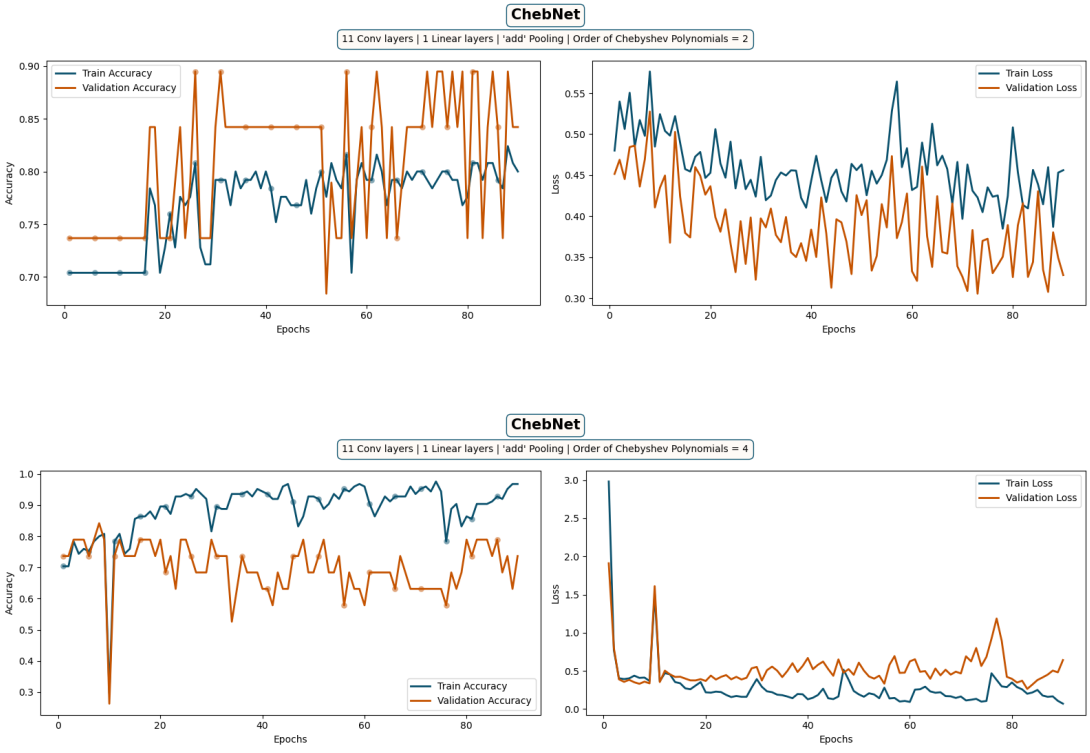
The multiplication $A_{\text{sym}}^{K} X$ can be interpreted as applying a convolutional filter based on the lowest-frequency signals of the graph Laplacian. This explains why over-smoothing occurs: as K increases, the repeated application of A_{sym} causes node embeddings to converge, effectively eliminating higher-frequency components and leaving only the low-frequency, smoothed signals.

For instance, if we let K grow sufficiently large, the embeddings $H^{(K)}$ converge to a fixed point satisfying:

$$A_{\text{sym}} H^{(K)} = H^{(K)}.$$

This fixed point is attainable because the dominant eigenvalue of A_{sym} is equal to one. At this point, the node embeddings are dominated by the graph's lowest-frequency components, rendering them nearly identical and uninformative.

Over-smoothing highlights a trade-off in GNN depth: while deeper architectures capture more global structure, excessive depth leads to homogeneous embeddings and a loss of discriminative power. This underscores the importance of architectural choices, such as residual connections or limiting the number of layers, to mitigate over-smoothing in practice.



We can observe that at this fixed point, all node features converge to be completely defined by the dominant eigenvector of A_{sym}. More generally, higher powers of A_{sym} emphasize the largest eigenvalues of this matrix. Furthermore, the largest eigenvalues of A_{sym} correspond to the smallest eigenvalues of its counterpart, the symmetric normalized Laplacian L_{sym}. These observations imply that multiplying a signal by high powers of A_{sym} corresponds to applying a convolutional filter based on the lowest eigenvalues (or frequencies) of L_{sym}. In other words, this process produces a low-pass filter.

Thus, from this simplified model, we see that stacking multiple rounds of message passing leads to convolutional filters that are inherently low-pass. In the worst case, these filters cause all node representations to converge to constant values within connected components of the graph, corresponding to the "zero-frequency" of the Laplacian.

B Theoretical Motivation for ChebConv

Graph Convolutional Networks (GCNs) have become a cornerstone of geometric deep learning, enabling effective learning on graph-structured data. The Chebyshev Convolutional Network (ChebConv) extends traditional GCNs by incorporating spectral graph theory and Chebyshev polynomials for localized graph filtering. This section outlines the theoretical principles underlying ChebConv and its advantages.

Chebyshev Polynomials and Spectral Graph Convolutions

Chebyshev polynomials are particularly effective in approximating spectral graph convolutions. The recursive definition of Chebyshev polynomials offers an efficient way to approximate functions of the Laplacian, where the Laplacian's eigenvalues serve as the arguments of the polynomials. Similar to Fourier analysis, which decomposes a signal into sinusoids, orthogonal polynomials decompose graph signals into components that align with the spectral properties of the graph.

Graph Fourier Transform

We are interested in processing signals defined on undirected and connected graphs $G = (V, E, W)$, where V is a finite set of $|V| = n$ vertices, E is a set of edges, and $W \in \mathbb{R}^{n\times n}$ is a weighted adjacency matrix encoding the connection weight between two vertices. A signal $x : V \to \mathbb{R}$ defined on the nodes of the graph may be regarded as a vector $x \in \mathbb{R}^n$ where x_i is the value of x at the i-th node.

An essential operator in spectral graph analysis is the graph Laplacian, whose combinatorial definition is $L = D - W \in \mathbb{R}^{n\times n}$, where $D \in \mathbb{R}^{n\times n}$ is the diagonal degree matrix with $D_{ii} = \sum_j W_{ij}$. The normalized definition is $L = I_n - D^{-1/2}WD^{-1/2}$, where I_n is the identity matrix.

As L is a real symmetric positive semidefinite matrix, it has a complete set of orthonormal eigenvectors $\{u_l\}_{l=0}^{n-1} \in \mathbb{R}^n$, known as the graph Fourier modes, and their associated ordered real nonnegative eigenvalues $\{\lambda_l\}_{l=0}^{n-1}$, identified as the frequencies of the graph. The Laplacian is indeed diagonalized by the Fourier basis $U = [u_0, \ldots, u_{n-1}] \in \mathbb{R}^{n\times n}$ such that:

$$L = U\Lambda U^T$$

where $\Lambda = \mathrm{diag}([\lambda_0, \ldots, \lambda_{n-1}]) \in \mathbb{R}^{n\times n}$.

The graph Fourier transform of a signal $x \in \mathbb{R}^n$ is then defined as:

$$\hat{x} = U^T x \in \mathbb{R}^n,$$

and its inverse as:

$$x = U\hat{x}.$$

As in Euclidean spaces, this transform enables the formulation of fundamental operations such as filtering.

Orthogonal Polynomials as Graph Filters

Message passing in GNNs can be interpreted as applying a spectral filter operation on the eigenvectors of the graph Laplacian, where the eigenvectors represent various modes of variation on the graph (i.e., capturing the "smoothness" of signals across the graph). The eigenvalues of the Laplacian matrix are closely related to the graph's connectivity, as smaller eigenvalues correspond to low-frequency modes that capture the global structure of the graph, while larger eigenvalues correspond to higher-frequency modes that capture more local details.

This message passing mechanism can be viewed as a convolution operation in the spectral domain, transforming the message-passing framework into a frequency-based message-passing. The spectral filter aggregates information from the graph in the frequency domain, where low-frequency modes encode global information and high-frequency modes focus on local details. The eigenvectors of the Laplacian matrix are linked to the Fourier basis of the graph, enabling the expression of smoothness and variations on the graph.

Spectral Graph Theory and Convolution

In the spectral domain, a graph is represented by its Laplacian matrix, L, which encapsulates the connectivity structure. The graph Laplacian can be decomposed as $L = U\Lambda U^\top$, where U is the matrix of eigenvectors, and Λ is the diagonal matrix of eigenvalues.

The convolution operation on graphs is defined in the spectral domain as:

$$g_\theta * x = U g_\theta(\Lambda) U^\top x$$

where x represents the input signal (e.g., node features), and $g_\theta(\Lambda)$ is a learnable filter parameterized by θ. This operation involves costly eigendecomposition, which scales poorly for large graphs.

Chebyshev Polynomials for Efficient Convolution

ChebConv addresses the computational inefficiency by approximating $g_\theta(\Lambda)$ using Chebyshev polynomials, $T_k(\Lambda)$, up to order K:

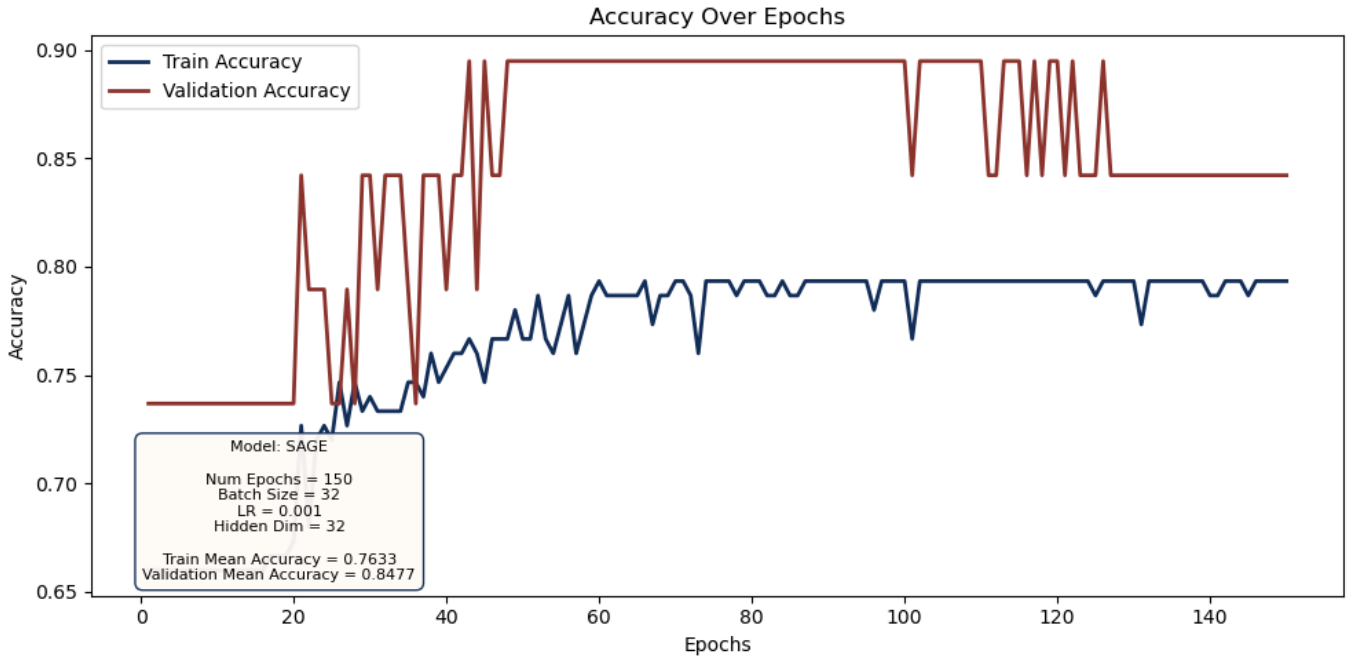
$$g_\theta(\Lambda) \approx \sum_{k=0}^{K} \theta_k T_k(\tilde{L})$$

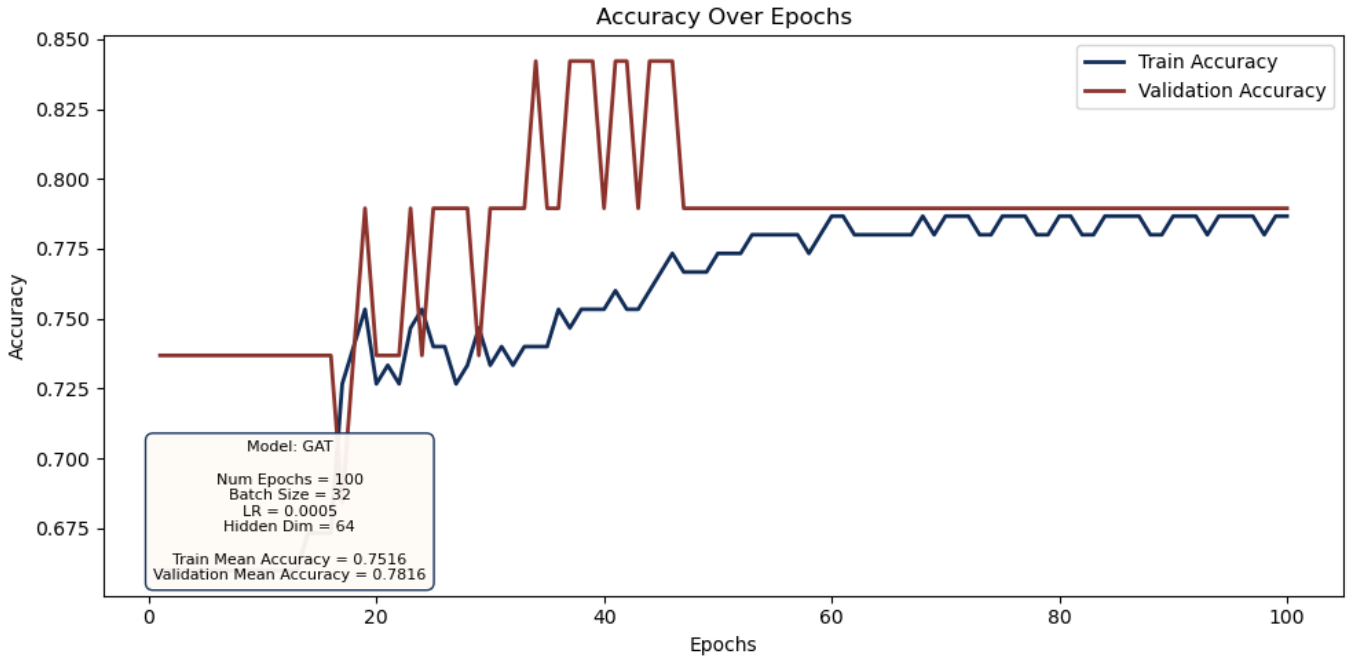
Here, $\tilde{L} = 2L/\lambda_{\max} - I$ is the normalized graph Laplacian, and T_k are the Chebyshev polynomials recursively defined as:

$$T_0(x) = 1, \quad T_1(x) = x, \quad T_k(x) = 2xT_{k-1}(x) - T_{k-2}(x)$$

This approximation eliminates the need for eigendecomposition, reducing computational complexity to $O(|E|)$, where $|E|$ is the number of edges. Additionally, the Chebyshev approximation is localized, meaning that the convolutional kernel operates over a K-hop neighborhood. Consequently, spectral filters represented by K-th order polynomials of the Laplacian are exactly K-localized. Furthermore, their learning complexity is $O(K)$, the support size of the filter, which is the same complexity as classical CNNs.

C Visualizations





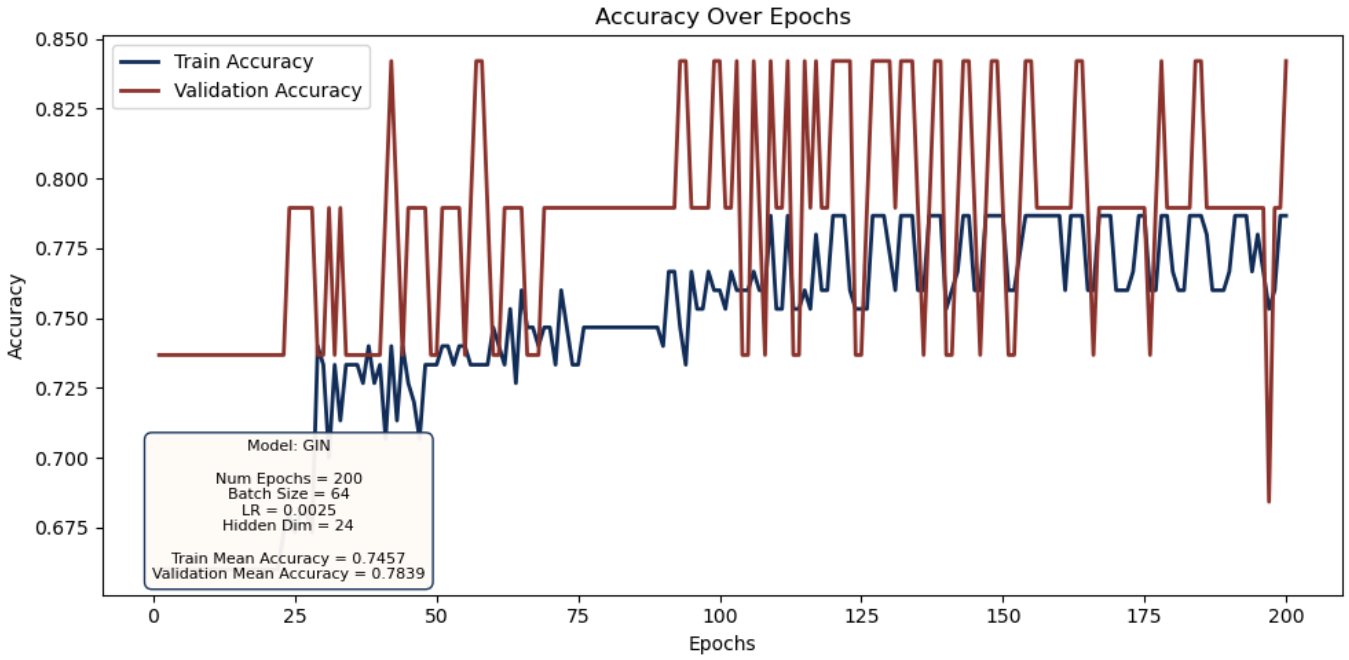
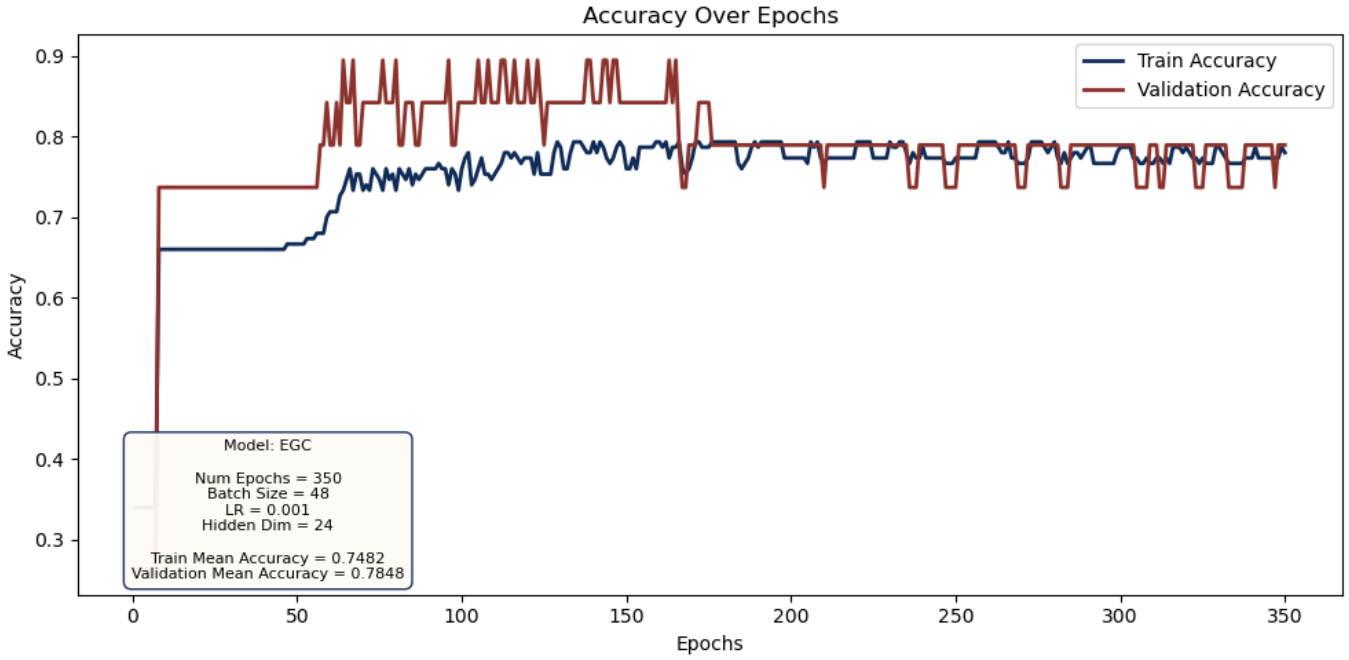
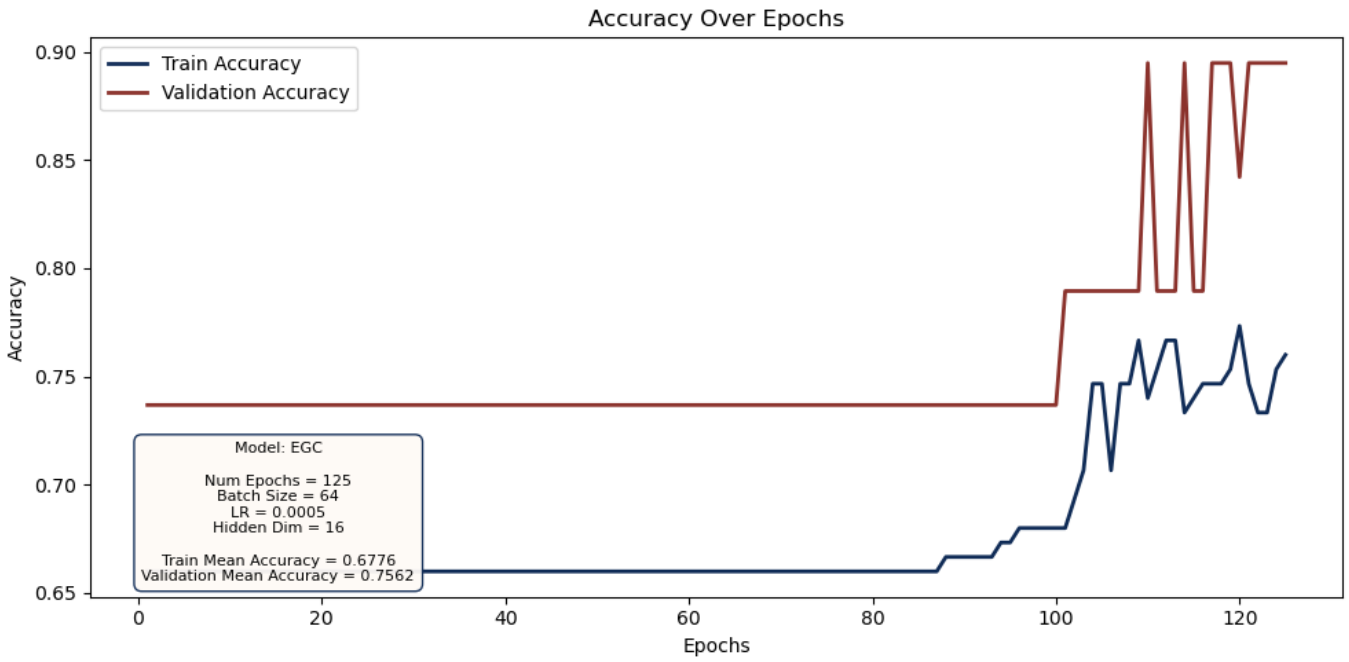
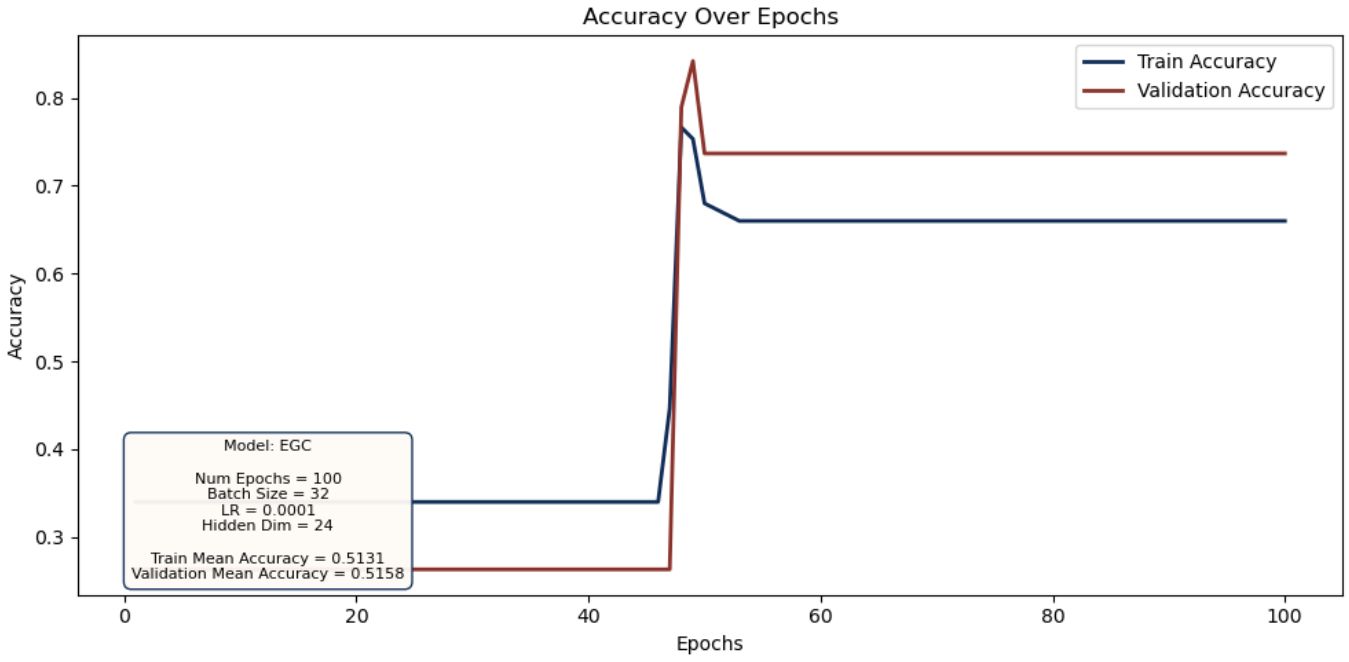


Figure 7: GIN was tested as a starting benchmark, but omitted later for experiments.




Accuracy Over Epochs
Train Accuracy
Validation Accuracy
Model: EGC
Num Epochs = 125
Batch Size = 64
LR = 0.0005
Hidden Dim = 16
Train Mean Accuracy = 0.6776
Validation Mean Accuracy = 0.7562
Accuracy
Epochs


Accuracy Over Epochs
Train Accuracy
Validation Accuracy
Model: EGC
Num Epochs = 100
Batch Size = 32
LR = 0.0001
Hidden Dim = 24
Train Mean Accuracy = 0.5131
Validation Mean Accuracy = 0.5158
Accuracy
Epochs