Christopher Goh Wei Jin

Benchmarking Simplicial Neural Networks

Computer Science Tripos – Part II

Hughes Hall

May 12, 2022

Declaration

I, Christopher Goh Wei Jin of Hughes Hall, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed: *Christopher Goh Wei Jin*

Date: May 12, 2022

Acknowledgements

I would like to express my immense gratitude to the following people for their support, without whom this work would not have been possible:

- **Cristian Bodnar**, my supervisor, for his guidance and for being a constant source of support and encouragement.
- **Professor Pietro Liò** for his advice on the manuscript.
- **Dr John Fawcett**, my DoS, for his ceaseless encouragement throughout my 3 years in Cambridge.
- Friends and family, who have supported me along this journey.

Proforma

Candidate Number:	**2407G**
Project Title:	**Benchmarking Simplicial Neural Networks**
Examination:	**Computer Science Tripos – Part II, July 2022**
Word Count:	**11421**[1]
Line Count:	**6732**[2]
Project Originator:	Cristian Bodnar
Supervisor:	Cristian Bodnar

Original Aims of the Project

The aim of the project was to create a software library capable of benchmarking different simplicial neural networks. This software contained at least one task (graph classification) and three different neural network models (2 simplicial neural network, 1 graph neural network) were supposed to be implemented and tested on this task.

Work Completed

The core aims have been exceeded. The final software library consists of four different tasks (graph classification, trajectory classification, adversarial resistance and unsupervised representation learning) as well as the implementation of six different neural network models (4 simplicial neural network, 2 graph neural network) that can be tested on these tasks. Additionally, my novel work on simplicial attention networks has been accepted as a poster paper at ICLR 2022 workshop on Geometrical and Topological Representation Learning.

Special Difficulties

None.

[1]This word count was computed using Overleaf's inbuilt word counter

[1]This line count was computed using `find . -name '*.py' | xargs wc -l`.

Contents

Chapter 1

Introduction

1.1 Overview

In recent times, the field of machine learning on graph data structures has been receiving more attention due to the expressive powers of graphs. Graphs are found commonly in the real world such as social network interactions, molecular structures, and protein-protein interactions, to name a few. Being able to harness the power of learning interactions within graphs will help us understand more about the world around us. Thus, with their ability to learn representations of nodes or graphs, graph neural networks (GNNs) have emerged as one of the most popular machine learning topics. The most common tasks performed by GNNs include node classification, edge prediction and graph classification [1, 2].

Since then, the field has turned it's attention towards using higher order representations, such as hypergraphs and simplicial complexes, in order to exploit topological information about the graph with the aims being to learn richer data. This dissertation will focus on simplicial neural networks (SNNs), specifically, creating a software capable of benchmarking SNNs. While GNNs operate at the node level, SNNs operate at the level of simplicial complexes, where a simplicial k-complex is a set of simplexes considered as a collective unit with the largest dimension of any simplex being k (a k-simplex) and a k-simplex is a geometric object with $(k + 1)$ vertices. So a point is a 0-simplex, an edge is a 1-simplex, and a triangle a 2-simplex. A pictorial representation of a simplex can be seen in Figure 1.1.

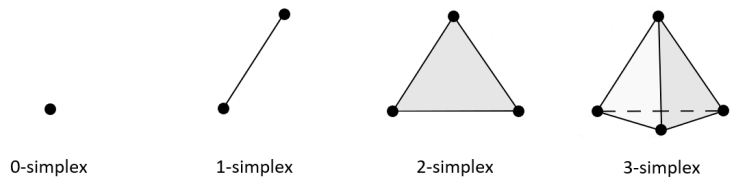


Figure 1.1: Example of a 0-simplex to 3-simplex.

1.2 Motivation

As mentioned in the proposal, while GNNs are limited to pair-wise interactions between vertices, SNNs can encode higher order information between them. A typical message-passing GNN works by aggregating information from neighbouring nodes through a graph convolution function and in doing so, each node can learn the structure and features of neighbouring nodes [3, 4, 5, 6, 7, 8, 9, 10]. A SNN works in a similar manner but instead of nodes, each simplex is able to learn the structure and features of neighbouring simplices. SNNs have recently been successfully applied to various problems such as missing data imputation [11], graph classification [12], edge prediction [13], trajectory classification [14] and prediction [15] as well as homology localisation [16]. However, the field of SNNs is still relatively new and there are no standard benchmarks or benchmarking tests available.

The three most important factors when it comes to creating tests for benchmarking is that they need to be rigorous to allow for fair comparison [17], reproducible due to the AI repliciation crisis [18], and diverse to prevent models from overfitting on specific tasks or datasets [19]. With these in place, we will be able to statistically separate the performance of different models and understand the types of tasks that each is more suited for. This is something I hope to implement and achieve.

1.3 Project summary

The final benchmarking pipeline includes four different types of tests for four different SNN models with two GNN models as a baseline.

1. The first benchmarking test is a graph classification test where the models are tested on their ability to classify pictures from the Modified National Institute of Standards and Technology database (MNIST) and Canadian Institute For Advanced Research-10 (CIFAR10) dataset when they are represented as graph data structures.
2. The second test involves trajectory classification. The models were tested on their ability to classify trajectories going through a graph. To make the test harder, all the training complexes use the same orientation for the edges, while the test trajectories use random orientations.
3. The third test involves comparing the different model's resistance to adversarial perturbation in the form of noise added to images from the MNIST dataset specifically tailored to each model with the aims of causing misclassification.
4. The last test is an unsupervised feature generation test where the models are scored on their ability to generate meaning node representations for nodes in the Cora citation network graph without knowing the node's true class. The expressiveness of these representations are measured by using a linear regression model to predict node classes from these new node representations.

The four SNN architectures implemented and tested were:

1. Simplicial convolutional network (SCN) proposed by Ebli, Defferrard, and Spreemann in "Simplicial neural networks" [11].
2. Simplicial 2-complex convolutional neural network (SCConv) proposed by Bunch et al. in "Simplicial 2-Complex Convolutional Neural Networks" [12].

3. The novel simplicial attention network (SAT) which is a generalisation of graph attention networks (GAT) applied to simplcial complexes. My paper on SAT has been accepted as a poster paper at ICLR 2022 workshop on Geometrical and Topological Representation Learning[1].
4. Simplicial attention neural network (SAN) proposed by Giusti et al. in "Simplicial Attention Neural Networks" [21]. This paper was posted on arXiv a few days after my paper on SAT. Although both use attention mechanisms, they use different approaches and thus, I felt SAN was worth including.

The two GNN baselines architectures implemented and tested were the graph convolutional network (GCN) proposed by Kipf and Welling [9] and GAT proposed by Veličković et al. [22].

Some of the content from my submission to ICLR will be recycled in this dissertation without reference in accordance with the blind submission principle.

[1]ICLR is the top artificial intelligence conference with a H5-index of 253 [20]

Chapter 2

Preparation

2.1 Graph neural networks

The main mechanism of a GNN is to take as input a set of input node features $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$, where N is the number of nodes in the graph and $\boldsymbol{x}_i \in \mathbb{R}^k$ where k is the number of features each node has and produce a new set of node features $\boldsymbol{X}' = \{\boldsymbol{x}'_1, \boldsymbol{x}'_2, \ldots, \boldsymbol{x}'_n\}$ where each new node feature vector has potentially different cardinality $\boldsymbol{x}'_i \in \mathbb{R}^{K'}$ and return this as output. Using these new node features, it is possible to perform the tasks of node, edge and graph classification.

A GNN works primarily by an information diffusion mechanism, which can be characterised by the convolution operation, an operation that computes the inner product between a weight vector and a n-dimension vectorised patch $\boldsymbol{I}^{(i,j)}$ centered around a point (i, j) [23]. The updated information of point (i, j) after the operation is given as:

$$O^{(i,j)} = \sum_{i=0}^{n} \boldsymbol{I}_i^{(i,j)} \boldsymbol{W}_i \tag{2.1}$$

In a graph, information diffusion via the convolutional operator happens between neighbouring nodes connected by edges. By utilising this mechanism, each node is able to aggregate information from neighbouring nodes [24]. This mechanism is not new and has been seen before in different architectures such as the convolution neural network [25] and cellular neural network [26].

GNN's that rely on the convolution operator are further broken down into two main categories which describes the way the way information is diffused. These two approaches are the spatial approach and spectral approach. The difference between spatial and spectral approaches will be discussed in this chapter, as well as how they relate to the GNNs chosen as part of the baseline. However, this chapter will not describe the implementation details of the SNNs used as part of the test.

2.1.1 Spectral approach

The motivation for the spectral approach comes from graph signal processing where given a graph signal f, we want a system H that transforms it into another signal $f' = \mathrm{H}(f)$.

Applying this to graph neural networks, we want to take a given node feature matrix and produce another node feature matrix $\boldsymbol{X}' = \mathrm{H}(\boldsymbol{X})$.

The spectral approach works with the spectral representation of the graph. The spectral representation of the graph resolves around the Laplacian matrix $\boldsymbol{L}$ of the graph, which is defined by

$$\boldsymbol{L}_{i,j} = \begin{cases} \deg(i) \text{ if } i = j \\ -1 \text{ if } i, j \text{ are connected by an edge} \\ 0 \text{ otherwise} \end{cases} \tag{2.2}$$

By utilising the eigenvectors/eigenvalues of the adjacency matrix, more properties about the graph can be revealed [27]. The equation for the Laplacian is as follows:

$$\boldsymbol{L} = \boldsymbol{D} - \boldsymbol{A} \tag{2.3}$$

where $\boldsymbol{A}$ represents the adjacency matrix of the graph, $\boldsymbol{D}$ represents the degree of each node in the graph, or the number of neighbours each node has, and $\boldsymbol{I}$ is the identity matrix. The spectral decomposition of the Laplacian can be further written as:

$$\boldsymbol{L} = \boldsymbol{U}\Lambda\boldsymbol{U}^\top \tag{2.4}$$

where $\boldsymbol{U}$ is the matrix of eigenvectors of the normalised graph Laplacian $\boldsymbol{L}$ and Λ is a diagonal matrix of the eigenvalues. We can also obtained the normalised version of the Laplacian, given by:

$$\hat{\boldsymbol{L}} = \boldsymbol{I} - \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2} \tag{2.5}$$

In order to transform the graph features into the spectral domain, we define two Fourier transforms:

$$\mathcal{F}(\mathbf{X}) = \boldsymbol{U}^\top\mathbf{X} \tag{2.6}$$
$$\mathcal{F}^{-1}(\mathbf{X}) = \boldsymbol{U}\mathbf{X} \tag{2.7}$$

The convolution operation on the graph is then defined as:

$$\begin{aligned} g \star (X) &= \mathcal{F}^{-1}(\mathcal{F}(g) \odot \mathcal{F}(\boldsymbol{X})) \\ &= \boldsymbol{U}(\boldsymbol{U}^\top g \odot \boldsymbol{U}^\top \boldsymbol{X}) \\ &= \boldsymbol{U} g_\theta(\Lambda)\boldsymbol{U}^\top \boldsymbol{X} \\ &= g_\theta(\boldsymbol{U}\Lambda\boldsymbol{U}^\top)\boldsymbol{X} \\ &= g_\theta(\boldsymbol{L})\boldsymbol{X} \end{aligned} \tag{2.8}$$

where the filter is defined as $g_\theta(\Lambda) = \mathrm{diag}(\boldsymbol{U}^\top g)$ and $\odot$ is the Hadamard operator. All spectral based GNN's follow this definition of $g_\theta(\boldsymbol{L})\boldsymbol{X}$ with the main difference being the choice of the filter g_θ [1, 28].

Defferrard, Bresson, and Vandergheynst [29] proposed a polynomial filter to approximate the filter function, given as:

$$g_{\theta'}(\boldsymbol{L}) \approx \sum_{k=0}^{K-1} \boldsymbol{w}_k T_k(\boldsymbol{L}) \tag{2.9}$$

where T_k is the Chebyshev polynomial recursively defined as $T_k(\boldsymbol{x}) = 2\boldsymbol{x}T_{k-1}(\boldsymbol{x}) - T_{k-2}(\boldsymbol{x})$, with $T_0(\boldsymbol{x}) = 1$ and $T_1(\boldsymbol{x}) = \boldsymbol{x}$, $\boldsymbol{w}_k$ is a vector of Chebyshev coefficients.

In the case of the graph convolutional network by Kipf and Welling [9], only the first two terms are used and the Laplacian operator is approximated as $\hat{\boldsymbol{L}} - \boldsymbol{I}$. Applying this to Equation 2.8:

$$\begin{aligned} g_{\theta'} \star \boldsymbol{X} &= g_{\theta'}(\hat{\boldsymbol{L}} - \boldsymbol{I})\boldsymbol{X} \\ &= (\boldsymbol{w}_o + \boldsymbol{w}_1(\hat{\boldsymbol{L}} - \boldsymbol{I}))\boldsymbol{X} \\ &= (\boldsymbol{w}_o + \boldsymbol{w}_1(-\boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}))\boldsymbol{X} \end{aligned} \tag{2.10}$$

In order to reduce the number of parameters, they set $\boldsymbol{w} = \boldsymbol{w}_0 = -\boldsymbol{w}_1$ to get:

$$g_{\theta'} \star \boldsymbol{X} = (\boldsymbol{w}(\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2}))\boldsymbol{X} \tag{2.11}$$

After further renormalisation of $\boldsymbol{I} + \boldsymbol{D}^{-1/2}\boldsymbol{A}\boldsymbol{D}^{-1/2} \rightarrow \hat{\boldsymbol{D}}^{-1/2}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-1/2}$ to avoid a vanishing gradient problem, we get the compact form of the GCN propagation step:

$$\boldsymbol{X}' = g_\theta \star \boldsymbol{X} = \phi(\hat{\boldsymbol{D}}^{-1/2}\hat{\boldsymbol{A}}\hat{\boldsymbol{D}}^{-1/2}\boldsymbol{X}\boldsymbol{W}) \tag{2.12}$$

where $\boldsymbol{W}$ is a learnable weight matrix and ϕ is an activation function. Common examples of spectral GNNs include graph convolutional networks mentioned above, and ChebNet [29].

2.1.2 Spatial approach

The spatial approach on the other hand, concerns itself more with the topology of the graph. In the case of spatial graphs, there is greater attention to the coordinates of the nodes, or at least the relative positions of the nodes, as well as the neighbours of each node [30]. Due to the use of the Laplacian/adjacency matrix in spectral models, the graph has to be viewed in its entirety. Spatial models on the other hand, can work on spatially local neighbourhoods of nodes. Since we are more interested in local neighbourhoods as opposed to the entire graph structure, a major challenge of spatial approaches is defining the convolution operation with differently sized neighbourhoods and treating the importance of each neighbour differently [1, 31].

One of the approaches used to solve this problem is the graph attention network. In this model, an attention mechanism is used to learn the relative weights between two connected nodes. The attention coefficient between node i and j denoted as $\alpha_{i,j}$ is calculated using the following formula:

$$\alpha_{i,j} = \text{softmax}_{j \in \mathcal{N}_i}(a(\boldsymbol{W}\boldsymbol{h}_i^k, \boldsymbol{W}\boldsymbol{h}_j^k)) \tag{2.13}$$

where $\boldsymbol{W}$ is a learnable weight matrix, a is a function for computing attention coefficients, and $\mathcal{N}_i$ refers to the neighbours of node i. The coefficient is normalised by a softmax such that all attention coefficient for node i sum to 1.

The convolution function is given as:

$$\boldsymbol{h}_i^{k+1} = \phi(\sum_{j \in \mathcal{N}_i} \alpha_{i,j}\boldsymbol{W}\boldsymbol{h}_j^k) \tag{2.14}$$

where $\alpha_{i,j}$ is the normalised attention coefficient mentioned earlier, and the new feature vector for node i, $\boldsymbol{h}_i^{k+1}$, is the sum of the products of attention coefficient applied to the feature vector of all the neighbours of i. This allows the model to assigning different importances to nodes of a same neighbourhood enabling a leap in model capacity and can generalise to generalise to graphs that were not seen in training. A multi-head attention mechanism can also be used to stabilise the learning process. The mechanism is formulated by processing K independent attention mechanisms execute the transformation of Equation 2.15, and concatenating their features, resulting in the following output feature representation:

$$\boldsymbol{h}_i^{k+1} = \Big\|_{k \leq K} \phi\Big(\sum_{j \in \mathcal{N}_i} \alpha_{i,j}^k \boldsymbol{W}^k \boldsymbol{h}_j^k\Big) \tag{2.15}$$

Common examples of spatial GNNs include GAT [22] and GraphSAGE [32].

2.2 Simplicial neural networks

SNNs are a generalisation of GNNs that operate at the level of simplicial complexes. Spectral neural network theory lends itself greatly to SNNs, with the key engine behind it being the Hodge–de Rham theory [33] that generalises the standard graph Laplacian, which describes node-to-node interactions via edges, to the Hodge-Laplacian. The Hodge-Laplacian allows us to model diffusion from edges-to-edges via nodes, edges-to-edges via triangles, triangles-to-triangles via edges and so on [13, 34].

2.2.1 Simplicial complexes

As mentioned in the introduction, simplicial complexes are a class of topological spaces that are made of simplices of various dimensions. Given a set of vertices V, a k-simplex is an unordered subset $\{v_0, v_1, \ldots, v_k\}$ where $v_i \in V$ and $v_i \neq v_j$ for all $i \neq j$. For a k-simplex $\sigma = \{v_0, v_1, \ldots, v_k\}$, we say its faces are all of the $(k-1)$-simplices that are also subsets of σ, while its cofaces are all $(k+1)$-simplices that have σ as a face. A simplex can also have an orientation, denoted by $[v_0, v_1, \ldots, v_k]$, where there is a chosen orientation for its vertices. An orientation is considered equivalent if they differ by an even permutation, or can be expressed by an even number of transposition. An example is:

$$[v_0, v_1, v_2, \ldots, v_k] = [v_1, v_2, v_0, \ldots, v_k] \tag{2.16}$$

On the other hand, a change in orientation will result in a change in the sign of the coefficient, where:

$$[v_0, \ldots, v_i, \ldots v_j, \ldots, v_k] = -[v_0, \ldots, v_j, \ldots v_i, \ldots, v_k] \tag{2.17}$$

The choices for orientation are mostly arbitrary and for book-keeping purposes [35, 36].

2.2.2 Adjacencies

Similar to how we consider two nodes to be adjacent if there exists an edge that connects the two of them together, there is a notion of adjacency for simplicial complexes. However,

adjacency can exist in two forms. Two k-simplices σ_i and σ_j are upper adjacent if both are faces of some $(k+1)$-simplex τ. If the complex is oriented, we further say σ_i and σ_j are similarly oriented with respect to τ if the orientations of σ_i and σ_j agree with the ones induced by τ. If not, they are dissimilarly oriented. Similarly, two k-simplices σ_i and σ_j are lower adjacent if both have a common face. We denote the upper adjacent simplices of σ by $\mathcal{N}_\sigma^\uparrow$ and the lower adjacent simplices by $\mathcal{N}_\sigma^\downarrow$. For the purposes of this paper, we also assume that $\sigma \in \mathcal{N}_\sigma^\uparrow$ and $\sigma \in \mathcal{N}_\sigma^\downarrow$. Given two adjacent d-dim simplices σ, τ, denote by $o_{\sigma,\tau} \in \{\pm 1\}$ the relative orientation between them with $o_{\sigma,\sigma} = 1$ for all σ. If the simplex is not oriented, we assume $o_{\sigma,\tau} = 1$ for all adjacent simplices σ, τ.

Referencing Figure 2.1 as an example, $[v_2, v_4]$ and $[v_2, v_1]$ are lower adjacent and similarly orientated with respect to each other, while $[v_3, v_2]$ and $[v_2, v_4]$ are upper adjacent and dissimilarly orientated with respect to each other and dissimilarly orientated with respect to $[v_3, v_4, v_2]$ [36].

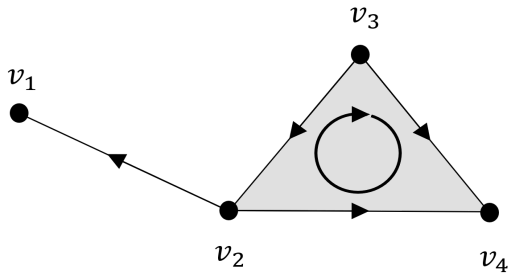


Figure 2.1: Example of a simplicial 2-complex

2.2.3 Boundary matrix

Consider a simplicial complex K. Denote by $\mathbf{C}_k$ the vector space with coefficients in $\mathbb{R}$ having the oriented k-simplices of K as its basis. Elements of this vector space are called k-chains. Then we can define a boundary operator $\partial_k : \mathbf{C}_k(X) \to \mathbf{C}_{k-1}(X)$ acting on the basis elements via $\partial_k[v_0, \ldots, v_k] := \sum_i (-1)^i [v_0, \ldots, \hat{v}_i, \ldots, v_k]$. This can be represented as a matrix $\mathbf{B}_k$ where the rows are indexed by $(k-1)$-simplices and the columns are indexed by k-simplices. Again, referencing Figure 2.1, we construct the two boundary matrices B_1 and B_2 shown in Equation 2.18.

$$\boldsymbol{B}_1 = \begin{array}{c|cccc} & [v_2,v_1] & [v_2,v_4] & [v_3,v_2] & [v_3,v_4] \\ \hline v_1 & 1 & 0 & 0 & 0 \\ v_2 & -1 & -1 & 1 & 0 \\ v_3 & 0 & 0 & -1 & -1 \\ v_4 & 0 & 1 & 0 & 1 \end{array} \qquad \boldsymbol{B}_2 = \begin{array}{c|c} & [v_3,v_4,v_2] \\ \hline [v_2,v_1] & 0 \\ [v_2,v_4] & -1 \\ [v_3,v_2] & -1 \\ [v_3,v_4] & 1 \end{array} \tag{2.18}$$

The rules for filling in the values for B_1 can be explained as:

$$\boldsymbol{B}_{1_{(v_i,[v_j,v_k])}} = \begin{cases} 0 \text{ if } v_i \neq v_j \text{ and } v_i \neq v_k \\ -1 \text{ if } v_i = v_j \\ 1 \text{ if } v_i = v_k \end{cases} \tag{2.19}$$

and the rules for filling in the values for $\boldsymbol{B}_2$ can be explained as:

$$\boldsymbol{B}_{2_{([v_i,v_j],[v_x,v_y,v_z])}} = \begin{cases} 0 \text{ if } \{v_i, v_j\} \not\subset \{v_x, v_y, v_z\} \\ -1 \text{ if } [v_i, v_j] \text{ does not have a similar orientation induced by } [v_x, v_y, v_z] \\ 1 \text{ if } [v_i, v_j] \text{ has a similar orientation induced by } [v_x, v_y, v_z] \end{cases} \tag{2.20}$$

The boundary operators also induces a co-boundary operator, which is the adjoint of $\boldsymbol{B}_k$, being $\boldsymbol{B}_k^\top$ which acts as an operator $\partial_k^\top : \boldsymbol{C}_{k-1}(X) \to \boldsymbol{C}_k(X)$

2.2.4 Hodge Laplacian

Based on the boundary and co-boundary matrix, we can define a Hodge Laplacian, which is a higher order generalisation of the graph Laplacian [34]. The k-th Hodge Laplacian is defined as:

$$\boldsymbol{L}_k = \boldsymbol{B}_k^\top \boldsymbol{B}_k + \boldsymbol{B}_{k+1}\boldsymbol{B}_{k+1}^\top \tag{2.21}$$

with L_k acting as a linear operator $L_k : \boldsymbol{C}_k(X) \to \boldsymbol{C}_k(X)$.

One interesting point to note is that $\boldsymbol{B}_k^\top \boldsymbol{B}_k$ corresponds to the lower adjacencies mentioned in section 1.2.2, whereas $\boldsymbol{B}_{k+1}\boldsymbol{B}_{k+1}^\top$ corresponds to the upper adjacencies. We can refer to them as $\boldsymbol{L}_k^\downarrow$ and $\boldsymbol{L}_k^\uparrow$ respectively.

We can rewrite Equation 2.21 as:

$$\boldsymbol{L}_k = \boldsymbol{L}_k^\downarrow + \boldsymbol{L}_k^\uparrow = \boldsymbol{B}_k^\top \boldsymbol{B}_k + \boldsymbol{B}_{k+1}\boldsymbol{B}_{k+1}^\top \tag{2.22}$$

Using the two boundary matrices from Equation 2.18 and setting $k = 1$, the up and down Laplacians for $\boldsymbol{L}_1$ are:

$$\boldsymbol{L}_1^\downarrow = \begin{pmatrix} 2 & 1 & -1 & 0 \\ 1 & 2 & -1 & 1 \\ -1 & -1 & 2 & 1 \\ 0 & 1 & 1 & 2 \end{pmatrix} \quad \boldsymbol{L}_1^\uparrow = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & -1 \\ 0 & 1 & 1 & -1 \\ 0 & -1 & -1 & 1 \end{pmatrix} \tag{2.23}$$

The rules for filling in the up and down Laplacian can be generalised to the following

$$(\boldsymbol{L}_k^\uparrow)_{\sigma,\tau} = \begin{cases} \deg(\sigma) \text{ if } \sigma = \tau \\ -1 \text{ if } \sigma \neq \tau, o_{\sigma,\tau} = 1, \text{ and } \mathcal{N}_{\sigma,\tau}^\uparrow \\ 1 \text{ if } \sigma \neq \tau, o_{\sigma,\tau} = -1, \text{ and } \mathcal{N}_{\sigma,\tau}^\uparrow \\ 0 \text{ otherwise} \end{cases} \tag{2.24}$$

$$(\boldsymbol{L}_k^\downarrow)_{\sigma,\tau} = \begin{cases} k+1 \text{ if } \sigma = \tau \\ -1 \text{ if } \sigma \neq \tau, o_{\sigma,\tau} = -1, \text{ and } \mathcal{N}_{\sigma,\tau}^\downarrow \\ 1 \text{ if } \sigma \neq \tau, o_{\sigma,\tau} = 1, \text{ and } \mathcal{N}_{i,j}^\downarrow \\ 0 \text{ otherwise} \end{cases} \tag{2.25}$$

In the case of this dissertation, since I am only focusing on simplicial 2 complexes, only B_1 and B_2 will be constructed. Thus, the three Laplacians used will be

$$\boldsymbol{L}_0 = \boldsymbol{B}_1\boldsymbol{B}_1^\top \tag{2.26}$$

$$\boldsymbol{L}_1 = \boldsymbol{B}_1^\top \boldsymbol{B}_1 + \boldsymbol{B}_2\boldsymbol{B}_2^\top \tag{2.27}$$

$$\boldsymbol{L}_2 = \boldsymbol{B}_2^\top \boldsymbol{B}_2 \tag{2.28}$$

with L_0 being the same as Equation 2.3. As such, we can see that the theory for simplicial neural networks is a generalisation of spectral graph theory.

As an example, using the simplicial complex from Figure 2.1 and the 2 boundary matrices from Equation 2.18, our three Laplacians are

$$\boldsymbol{L}_0 = \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 3 & -1 & -1 \\ 0 & -1 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{pmatrix} \quad \boldsymbol{L}_1 = \begin{pmatrix} 2 & 1 & -1 & 0 \\ 1 & 3 & 0 & 0 \\ -1 & 0 & 3 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix} \quad \boldsymbol{L}_2 = \begin{pmatrix} 3 \end{pmatrix} \tag{2.29}$$

Combining our observation from Equation 2.24 and Equation 2.25, the general rule for filling in the k-th Hodge Laplacian can be described as:

$$(\boldsymbol{L}_k)_{\sigma,\tau} = \begin{cases} k+1+\deg(\sigma) \text{ if } \sigma = \tau \\ 1 \text{ if } \sigma \neq \tau, \mathcal{N}^{\downarrow}_{\sigma,\tau}, \neg\mathcal{N}^{\uparrow}_{\sigma,\tau}, \text{ and } o_{\sigma,\tau} = 0 \\ -1 \text{ if } \sigma \neq \tau, \mathcal{N}^{\downarrow}_{\sigma,\tau}, \neg\mathcal{N}^{\uparrow}_{\sigma,\tau}, \text{ and } o_{\sigma,\tau} = -1 \\ 0 \text{ if } \sigma \neq \tau, (\mathcal{N}^{\uparrow}_{\sigma,\tau} \text{ or } \neg\mathcal{N}^{\downarrow}_{\sigma,\tau}) \end{cases} \tag{2.30}$$

While the equation for filling in the k-th Hodge-Laplacian can be written as

$$\boldsymbol{L}_k = \boldsymbol{D} - \boldsymbol{A}^{\uparrow} + (k+1)\boldsymbol{I} + \boldsymbol{A}^{\downarrow}, k > 0. \tag{2.31}$$

Now we have three matrices that not only tell us the adjacencies between simplicial complexes, but also gives us information about the type of adjacency between them, capturing the different higher-order interactions between neighbouring simplices [36].

The existent simplicial convolutional networks rely on this Laplacian (and its normalised versions) to weight the different adjacencies between simplices. As one can see, the Hodge-Laplacian forms the backbone behind SNN theory. Further details about how they are used in the specific SNN models used will be covered in the implementation chapter.

2.3 Requirements analysis

Requirements. As set out in my proposal, the requirements of this project can be broken into three core (C) and three extension (E) requirements;

C1. Create a benchmarking software that is able to test GNNs and SNNs.

C2. Implement the SNN models by Bunch et al. [12] and Ebli et al. [11] as part of the software. Also add a GCN model as a baseline.

C3. Add one test based on identifying the ground truth labels of superpixel graphs generated from images.

E1. Implement a novel SNN based on the knowledge acquired from studying SNN and GNN architectures, or a new SNN that was published after the proposal was written.

E2. Add a second test that is able to further demonstrate the advantage/disadvantage of taking higher order interactions into account.

E3. Release the software to public as a usable benchmarking library, such as by releasing it as a Python library or adding the datasets into PyTorch-geometric.

2.4 Tools used

2.4.1 Development

Python 3.8 was the main language of choice, given the ease of use of the language and the multitude of machine learning libraries available. In order to ensure a consistent code style, Google `yapf` was used to format code to PEP 8 standard.

2.4.2 Libraries

1. `PyTorch` [37] was the main library used for machine learning given its flexibility, Pythonic code structure, and GPU support. An extension of `PyTorch`, `PyTorch-geometric` [38] was also used due to its support for graph neural networks.
2. `Scipy` [39], a scientific computing library, was also used due to having better support for sparse matrix operations compared to `PyTorch`.
3. `Scikit-image` [40] was used for image processing and to create the superpixel graphs from MNIST and CIFAR10 images.
4. Other common graph and scientific computing libraries such as `networkx` [41] and `NumPy` [42] were also used.
5. Python's `unittest` library was used to write unit tests.

   Table 2.1 contains a summary of the third-party libraries used, their purpose and license. Compatibly with these, My code is released publicly under the MIT Licence.

Library	Purpose	License
`PyTorch` [37]	Machine learning	BSD license
`PyTorch-geometric` [38]	Graph machine learning	MIT license
`Scikit-image` [40]	Image processing	BSD license
`Scipy` [39]	Sparse matrix implementation	BSD license
`NumPy` [42]	Scientific computing	BSD License
`tqdm` [43]	Progress report	MIT License
`joblib`	Parallel processing	BSD license

Table 2.1: Third-party libraries used.

2.4.3 Version control

Git was used for version control and I frequently pushed my code to GitHub in order to back it up. Other files such as log files, Python notebooks, and results were regularly stored in Google Drive.

2.4.4 Resources

A majority of the project code was written in PyCharm IDE due to familiarity and having in-built support for Git version control. However, validating parts of the pipeline, such as superpixel generation and GNN implementation, was done on Google Colab. Dataset

generation, model training and evaluation were done using the GPUs provided by the Computational Biology group.

2.5 Starting point

In terms of technical experience with the tools used in this project, I am quite familiar with the language Python and have used it for a few years. However, I have never used any of the major libraries required for this dissertation such as `PyTorch` and `Scipy` prior to this project. I have also never contributed to any open source projects or written software libraries in the past, though I am familiar with Git as a version control tool.

As for prior knowledge regarding the content covered in this project, I have never interacted with graph neural networks, nor covered graph theory before this. Naturally, this was a steep learning curve and time had to be spent learning these. However, I benefited from some of the content covered in different courses from the computer science tripos such as NST maths (part IA), machine learning and real world data (part IA), artificial intelligence (part IB), information theory (part II) and deep neural networks (part II) and they made learning these new concepts easier.

Chapter 3

Implementation

This chapter describes the implementation of the four SNNs (SCN, SCConv, SAT, and SAN), the four benchmarking test (superpixel classification, trajectory classification, adversarial resistance, and unsupervised representational learning), as well as a brief overview of the code structure and design principles used.

3.1 SNN models

Compared to GNNs, SNNs operate on three sets of features. Firstly, the node level residuals, represented as $\boldsymbol{H}_0$, is a $\mathbb{R}^{N\times K}$ matrix that describes the node features where N is the number of nodes. Secondly, the edge level residuals, represented as $\boldsymbol{H}_1$, is a $\mathbb{R}^{E\times K'}$ matrix that describes the edge features where E is the number of edges. Lastly, $\boldsymbol{H}_2$, the triangle level residuals, is a is a $\mathbb{R}^{T\times K''}$ matrix that describes the triangle features where T is the number of triangles.

3.1.1 SCN

Ebli, Defferrard, and Spreemann's SNN model (SCN) [11] is the first SNN implemented in this paper. It has the following propagation mechanism:

$$\boldsymbol{H}_i^{k+1} = \phi(\boldsymbol{L}_i'(\Big\|_{k\leq K} T_j(\boldsymbol{H}_i^k)\boldsymbol{W}_i^k)) \tag{3.1}$$

where i is in the range of 0 to 2, one for each of the Hodge Laplacians, $\boldsymbol{L}_i'$ refers to the i-th scaled Hodge Laplacian $\boldsymbol{L}$ referenced in the preparation chapter, $\boldsymbol{W}_i^k$ is a learnable weight matrix, and $(\Big\|_{k\leq K} T_j(\boldsymbol{H}_i^k)\boldsymbol{W}_i^k)$ refers to the concatenation of k-th Chebyshev polynomial of H_i^k multiplied with weight matrix $\boldsymbol{W}_i^k$ up to K.

In the original implementation by Ebli et al., the Laplacian is scaled in the following manner:

$$\boldsymbol{L}_i' = \boldsymbol{I} - \frac{2}{\max(\Lambda)}\boldsymbol{L}_i \tag{3.2}$$

where all elements in the Laplacian are multiplied by two before being divided by the maximum eigenvalue and subtracted from the identity matrix. This scaling is based on a approximated harmonic space projection operator and acts as a harmonic filter [44] and we follow this scaling throughout the paper. We also limit the concatenation

of Chebyshev polynomials to only the 1st polynomial in order to keep things constant between the different models [11].

3.1.2 SCConv

Bunch et al.'s SNN model (SCConv) [12] is implemented by first normalizing the matrices according to the following equations proposed by Schaub et al. [45]:

$$\begin{aligned}
\boldsymbol{D}_1 &= 2 \cdot \text{diag}(|\boldsymbol{B}_1|\boldsymbol{D}_2\mathbf{1}) \\
\boldsymbol{D}_2 &= \max(\text{diag}(|\boldsymbol{B}_2|\mathbf{1}), \boldsymbol{I}) \\
\hat{\boldsymbol{D}}_2 &= \max(\text{diag}(|\boldsymbol{B}_1|\mathbf{1}), \boldsymbol{I}) \\
\boldsymbol{D}_3 &= \frac{1}{3}\boldsymbol{I} \\
\hat{\boldsymbol{D}}_3 &= \boldsymbol{I} \\
\boldsymbol{D}_4 &= \boldsymbol{I} \\
\boldsymbol{D}_5 &= \text{diag}(|\boldsymbol{B}_2|\mathbf{1})
\end{aligned} \tag{3.3}$$

$\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ are the two boundary matrices described in the preparation chapter and $\text{diag}(|\boldsymbol{B}_k|\mathbf{1})$ is the diagonal matrix of 1 over the number of neighbours each node (when $k = 0$) or edge (when $k = 1$) has. Using these, we obtain the normalised Laplacians as:

$$\begin{aligned}
\hat{\boldsymbol{L}}_0^{\uparrow} &= \boldsymbol{B}_1\hat{\boldsymbol{D}}_3\boldsymbol{B}_1^{\top}\hat{\boldsymbol{D}}_2^{-1} \\
\hat{\boldsymbol{L}}_1^{\uparrow} &= \boldsymbol{D}_2\boldsymbol{B}_1^{\top}\hat{\boldsymbol{D}}_1^{-1}\boldsymbol{B}_1 \\
\hat{\boldsymbol{L}}_1^{\downarrow} &= \boldsymbol{B}_2\boldsymbol{D}_3\boldsymbol{B}_2^{\top}\boldsymbol{D}_2^{-1} \\
\hat{\boldsymbol{L}}_2^{\downarrow} &= \boldsymbol{D}_4\boldsymbol{B}_2^{\top}\boldsymbol{D}_5^{-1}\boldsymbol{B}_2
\end{aligned} \tag{3.4}$$

Where $\boldsymbol{L}_k^{\uparrow}$ is the k-th normalised up Laplacian and $\boldsymbol{L}_k^{\downarrow}$ is the k-th normalised down Laplacian. Instead of using the normalised Laplacian, Bunch et al. instead opts for using the normalised adjacency matrices which are defined as:

$$\begin{aligned}
\boldsymbol{A}_0^{\uparrow} &= \hat{\boldsymbol{D}}_2 - \hat{\boldsymbol{L}}_0^{\uparrow}\hat{\boldsymbol{D}}_2 \\
\boldsymbol{A}_1^{\uparrow} &= \boldsymbol{D}_2 - \boldsymbol{D}_2\hat{\boldsymbol{L}}_1^{\uparrow} \\
\boldsymbol{A}_1^{\downarrow} &= \hat{\boldsymbol{D}}_2^{-1} - \hat{\boldsymbol{D}}_2^{-1}\hat{\boldsymbol{L}}_1^{\downarrow} \\
\boldsymbol{A}_2^{\downarrow} &= \boldsymbol{D}_4^{-1} - \boldsymbol{D}_4^{-1}\hat{\boldsymbol{L}}_2^{\downarrow}
\end{aligned} \tag{3.5}$$

Lastly, self-loops are added in a style similar to what Kipf and Welling [9] proposed to prevent a vanishing gradient.

$$\begin{aligned}
\hat{\boldsymbol{A}}_0^{\uparrow} &= (\boldsymbol{A}_0^{\uparrow} + \boldsymbol{I})(\hat{\boldsymbol{D}}_2 + \boldsymbol{I})^{-1} \\
\hat{\boldsymbol{A}}_1^{\uparrow} &= (\boldsymbol{A}_1^{\uparrow} + \boldsymbol{I})(\boldsymbol{D}_2 + \boldsymbol{I})^{-1} \\
\hat{\boldsymbol{A}}_1^{\downarrow} &= (\boldsymbol{D}_2 + \boldsymbol{I})(\boldsymbol{A}_1^{\downarrow} + \boldsymbol{I}) \\
\hat{\boldsymbol{A}}_2^{\downarrow} &= (\boldsymbol{D}_4 + \boldsymbol{I})(\boldsymbol{A}_2^{\downarrow} + \boldsymbol{I})
\end{aligned} \tag{3.6}$$

Using these equations, the propagation steps are then defined as:

$$\begin{aligned}
\boldsymbol{H}_0^{k+1} &= \phi(\hat{\boldsymbol{A}}_0^{\uparrow}\boldsymbol{H}_0^h\boldsymbol{W}_{0,0}^k \;||\; \boldsymbol{D}_1^{-1}\boldsymbol{B}_1\boldsymbol{H}_1^k\boldsymbol{W}_{1,0}^k) \\
\boldsymbol{H}_1^{k+1} &= \phi(\boldsymbol{D}_2\boldsymbol{B}_1^{\top}\boldsymbol{D}_1^{-1}\boldsymbol{H}_0^k\boldsymbol{W}_{0,1}^k \;||\; (\hat{\boldsymbol{A}}_1^{\uparrow} + \hat{\boldsymbol{A}}_1^{\downarrow})\boldsymbol{H}_1^k\boldsymbol{W}_{1,1}^k \;||\; \boldsymbol{B}_2\boldsymbol{D}_3\boldsymbol{H}_2^k\boldsymbol{W}_{2,1}^k) \\
\boldsymbol{H}_2^{k+1} &= \phi(\boldsymbol{D}_4\boldsymbol{B}_2^{\top}\boldsymbol{D}_5^{-1}\boldsymbol{H}_1^k\boldsymbol{W}_{1,2}^k \;||\; \hat{\boldsymbol{A}}_2^{\downarrow}\boldsymbol{H}_2^k\boldsymbol{W}_{2,2}^k)
\end{aligned} \tag{3.7}$$

where $||$ is a horizontal concatenation, ϕ is an activation function and $\boldsymbol{W}_{i,j}^h$ once again refers to a learnable weight matrix [12].

SCConv is currently the only SNN in which each propagation step takes into account features from adjacent dimensions. The node level features $\boldsymbol{H}_0^{k+1}$ is updated with information from $\boldsymbol{H}_0^k$ and $\boldsymbol{H}_1^k$, the edge-level features $\boldsymbol{H}_1^{k+1}$is updated with information from all three layers and the triangle-level features $\boldsymbol{H}_2^{k+1}$ is updated with information from $\boldsymbol{H}_1^k$ and $\boldsymbol{H}_2^k$. This is in contrast to SCN in which dimensions do not interact with each other.

3.1.3 SAT (Extension)

The simplicial attention network (SAT) is a novel SNN that was designed and implemented based on my understanding of GAT and SNNs. The motivation for the creation of such a model was to leverage the success of attention mechanisms in structured domains in order to dynamically weigh the interactions between neighbouring simplices and support a model that can readily adapt to novel structures.

Let K be a simplicial complex. The model will be described for an arbitrary dimension of the complex $d \leq \dim(K)$. Attention coefficients are computed for the up $\alpha_{\sigma,\tau}^{\uparrow}$ and down $\alpha_{\sigma,\tau}^{\downarrow}$ adjacencies via the following equations:

$$\alpha_{\sigma,\tau}^{\uparrow} = o_{\sigma,\tau} \cdot \mathrm{softmax}_{\tau \in \mathcal{N}_{\sigma}^{\uparrow}}(a(\boldsymbol{W}_1\boldsymbol{h}_{\sigma}^k, \boldsymbol{W}_1\boldsymbol{h}_{\tau}^k)), \tag{3.8}$$

$$\alpha_{\sigma,\tau}^{\downarrow} = o_{\sigma,\tau} \cdot \mathrm{softmax}_{\tau \in \mathcal{N}_{\sigma}^{\downarrow}}(a(\boldsymbol{W}_2\boldsymbol{h}_{\sigma}^k, \boldsymbol{W}_2\boldsymbol{h}_{\tau}^k)), \tag{3.9}$$

where a is a function for computing attention coefficients. This effectively becomes a form of signed attention in an oriented simplicial complex.

SAT layers are described by the following message passing equation weighting the neighbours by the attention coefficients:

$$\boldsymbol{h}_{\sigma}^{k+1} = \phi\Big(\sum_{\tau \in \mathcal{N}_{\sigma}^{\uparrow}} \alpha_{\sigma,\tau}^{\uparrow}\boldsymbol{W}_1^k\boldsymbol{h}_{\tau}^k, \sum_{\tau \in \mathcal{N}_{\sigma}^{\downarrow}} \alpha_{\sigma,\tau}^{\downarrow}\boldsymbol{W}_2^k\boldsymbol{h}_{\tau}^k\Big) \tag{3.10}$$

Here, ϕ is an update function that aggregates the two incoming messages from the lower and upper adjacencies and updates the representation of σ. More generally, each of these arguments can effectively be augmented with Z attention heads:

$$\boldsymbol{h}_{\sigma}^{k+1} = \Big\|_{z \leq Z}\phi\Big(\sum_{\tau \in \mathcal{N}_{\sigma}^{\uparrow}} \alpha_{\sigma,\tau}^{\uparrow,z}\boldsymbol{W}_1^k\boldsymbol{h}_{\tau}^k, \sum_{\tau \in \mathcal{N}_{\sigma}^{\downarrow}} \alpha_{\sigma,\tau}^{\downarrow,z}\boldsymbol{W}_2^k\boldsymbol{h}_{\tau}^k\Big) \tag{3.11}$$

Note that when working at the node-level, where the relative orientation between nodes is trivial and only upper adjacencies are present, one recovers GAT [22].

A pictorial representation of the attention coefficient being calculated can be seen in Figure 3.1.

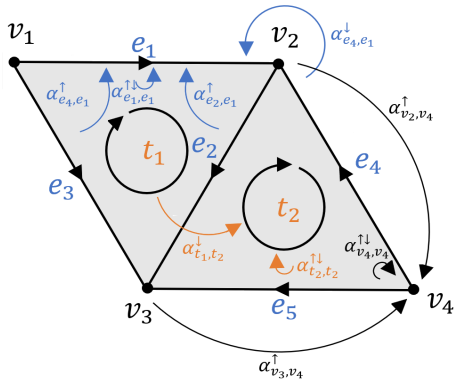


Figure 3.1: Pictorial diagram of attention coefficients being calculated for node v_4, edge e_1, and triangle t_2.

3.1.4 SAN (Extension)

Despite sharing similar names and the use of the attention mechanisms to increase performance, SAN, proposed by Giusti et al. [21], differs fundamentally from SAT. SAT is based solely on an attentional framework. SAN on the other hand is based on the Hodge decomposition and combines both convolutional and attentional mechanisms to approximate signals through the graph [21].

The Hodge decomposition is used to process the signals within a k-simplex and states that the the space of a k-simplex can be decomposed into three orthogonal subspaces:

$$\mathbb{R}^{N_k} = \mathbf{image}(\boldsymbol{B}_{k+1}) \oplus \mathbf{image}(\boldsymbol{B}_k^\top) \oplus \mathbf{kernel}(\boldsymbol{L}_k) \tag{3.12}$$

where $\oplus$ represents the union of orthogonal subspaces and N_k represents the cardinality of signals with N_1 representing nodes, N_2 representing edges and so on [46, 47].

We refer to $\mathbf{image}(\boldsymbol{B}_{k+1})$ as the *solenoidal* component, $\mathbf{image}(\boldsymbol{B}_k^\top)$ as the *irrotational* component and $\mathbf{kernel}(\boldsymbol{L}_k)$ as the *harmonic* component. In order to approximate the three different components, the following propagation mechanism is used.

$$\boldsymbol{H}_i^{k+1} = \phi\big(GAT(\boldsymbol{H}_i^k, \boldsymbol{L}_i^\uparrow) + GAT(\boldsymbol{H}_i^k, \boldsymbol{L}_i^\downarrow) + \boldsymbol{L}_i' \boldsymbol{H}_i^k \boldsymbol{W}\big) \tag{3.13}$$

GAT$(\boldsymbol{H}, \boldsymbol{L})$ refers to the graph attention network propagation mechanism with features $\boldsymbol{H}$ on neighbourhood $\boldsymbol{L}$, and $\boldsymbol{L}_i'$ once again refers to the approximated harmonic space projection operator that acts as a harmonic filter. GAT$(\boldsymbol{H}_i^k, \boldsymbol{L}_i^\uparrow)$ corresponds to the *solenoidal* component (based on upper neighbourhoods), GAT$(\boldsymbol{H}_i^k, \boldsymbol{L}_i^\downarrow)$ corresponds to the *irrotational* component (based on lower neighbourhoods) and $\boldsymbol{L}_i' \boldsymbol{H}_i^k \boldsymbol{W}$ corresponds to the *harmonic* component.

3.2 Benchmarking tests

3.2.1 Superpixel image classification

A superpixel graph is the graph representation of an image in which pixels are grouped into nodes representing perceptually meaningful regions, such as a region of similar

intensity [48]. By representing the image as a graph, it is possible to change the task of image classification to graph classification. This application of using graph neural networks on superpixel graphs for the task of image classification was first done by Monti et al. [49] and has since been a popular framework for testing graph neural networks. A form of graph image classification was also used by Bunch et al. when they produced their simplicial neural network model [12], but instead of a superpixel graph they grouped pixels at fixed intervals. Thus, this method was chosen as the first benchmarking test.

The images used are from the Modified National Institute of Standards and Technology database (MNIST) [50] and Canadian Institute For Advanced Research-10 (CIFAR10) [51] dataset. The MNIST dataset is a dataset consisting of handwritten digits from 0 to 9 while the CIFAR10 dataset consists of images from 10 classes including birds, planes and frogs.

There are many different superpixel algorithm that exist, such as Simple Linear Iterative Clustering (SLIC) [48], Superpixels Extracted via Energy-Driven Sampling (SEEDS) [52], and others. SLIC was chosen as it is fast, stable and readily available in the Skimage library. The superpixel nodes are then connected to directly adjacent nodes by means of a region adjacency graph to get the final graph. From this graph we construct a simplicial 2-complex. Triangles are of particular importance in this graph as they tend to encode tightly connected regions constituting a form of higher-order interaction between the superpixels.

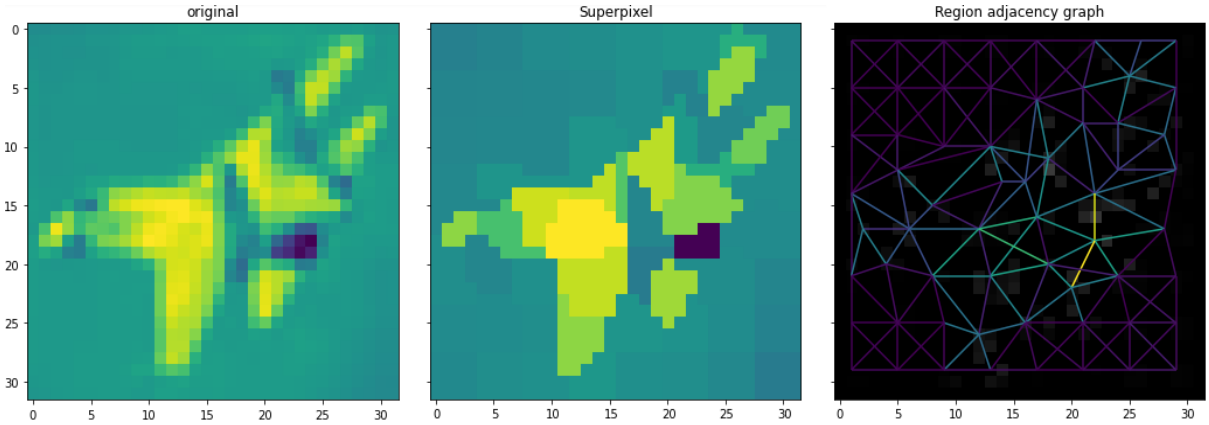


Figure 3.2: The stages from image to graph of a greyscaled CIFAR10 image.

We follow the work of Long, Yan, and Chen [53] in setting up the node features and how the GNN layers are used. The node features of node cluster j are $\boldsymbol{h}_j = [1/n \sum_i^{n_j} (a_i, b_i, r_i, g_i, b_i)]$ where the a_i and b_i are the x and y coordinate of pixel i respectively, and r_i, g_i, b_i are the red, green, and blue pixel values for pixel i. Since MNIST is greyscale, $r_i = g_i = b_i$ in the MNIST superpixel graph. The node feature is then the average of all the pixels associated with the j-th superpixel cluster. The features for the 1-simplices and 2-simplices will be the concatenation of the node features that make up the simplex in the order given by pixel values.

Our layers are arranged in the same architecture proposed by Long, Yan, and Chen known as a hierarchical GNN architecture. This architecture comprises of three GNN/SNN layer. The resulting node features from each layer, also known as the residual, are concatenated, resulting in the penultimate feature vector comprising of features vectors from previous

layers. This is then passed through a mean pooling function, before being fed to a multi-layer perceptron (MLP) with 10 outputs followed by a softmax activation. A pictorial representation of the architecture is shown in Figure 3.3.

The reason for combining the residuals is because nodes in a GNN tend to over-smooth, where nodes of the same subgraph end up having the same values or features with deeper layers, resulting in a drop in performance with increased depth. Thus, relying only on the residuals from the final layer might result in limited performance. This is also known as the over-smoothing problem [54, 55].

In order to include edge and triangle features, which are also being calculated in parallel, the residual edge features and the residual triangle features are concatenated separately. This is followed by a mean pooling function and then an MLP layer for both. The result is three different vectors of size 10, one being the predictions of the class by using node features, the second by edge features and the last by triangle features. These three vectors are combined and a softmax is applied to get the likelihood of each class.

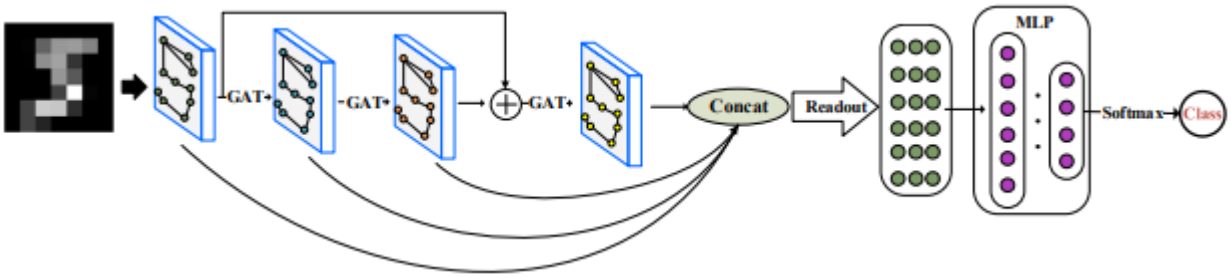


Figure 3.3: Hierarchical GNN architecture proposed by Long, Yan, and Chen with 3 GAT layers in their paper "A Graph Neural Network for superpixel image classification" [53]. In the benchmarking test, the GAT layer is swapped with the model we want to test.

Although the primary metric of concern would be the classification accuracy of the model on the test dataset, other metrics such as time taken per epoch and top 5 error would also be measured to give a more well rounded view of model performance.

3.2.2 Orientation Flow (Extension)

The orientation flow test is another classification task that was implemented as part of this project. In contrast to superpixel graph classification, this test consists of two different classes of trajectories represented as simplex orientations and the metric we are interested in measuring is the classification accuracy of the models.

As mentioned in the preparation chapter, simplicial complexes are given an orientation. Although the choice of orientation is usually arbitrary, it can be used to define signals within a graph such as an information flow [45], as demonstrated in this test.

The orientation flow dataset is a synthetic dataset constructed by sampling 1000 points within a unit square. Two holes within the graph are created by removing nodes and the edges connected to these nodes. Delauney triangulation is then used to construct triangles from the remaining nodes. Trajectories are generated by sampling a point randomly from the top-left corner of the complex and the end point is a point randomly sampled from the bottom right of the complex. Two different types of trajectories between the top-left corner and the bottom-right corner are generated, with trajectories from one class

traversing via the top-right corner, and trajectories from the second class traversing via the bottom-left corner. 1000 train and 200 test trajectories are constructed. This test is not new and has been covered in other works [21, 35, 46, 56]. An example of the dataset and some trajectories can be seen in Figure 3.4.

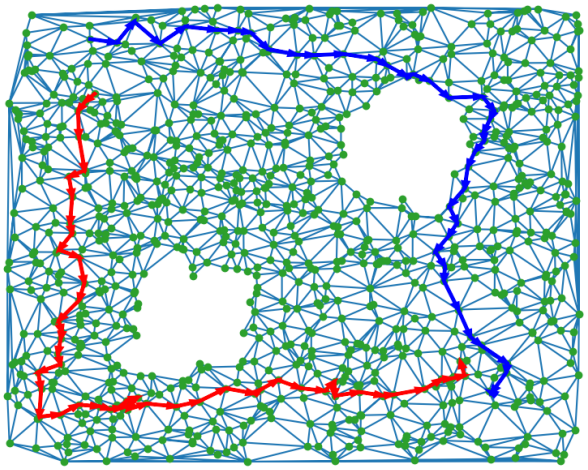


Figure 3.4: Example of the two different types of trajectories generated in this test

A side effect of this test is that is able to verify if models satisfy the desirable property of orientation equivariance. Glaze, Roddenberry, and Segarra introduced the idea of an admissible simplicial neural network architecture [35]. In order to consider an SNN architecture admissible, the architecture has to satisfy the following three properties

1. **Permutation equivariance**. This is a property extended from graph neural networks, in which the choice of node labels should not affect the result of the neural network architecture. In order to achieve this, the aggregation function of neighbours has to be independent of the order in which neighbours are presented to it, such as using a sum or mean aggregation function.
2. **Orientation equivariance**. This is a property in which the choice of simplex orientation should not affect the result of the neural network architecture.
3. **Simplicial awareness**. This is a property in which an architecture that uses k-simplicial layers has to use all its layers. If the architecture performs just as well without any of the layers then it violates this property.

In order for the test to be challenging for non-orientation invariant models, all trajectories from the training dataset use the same orientation for the edges, while trajectories from the test dataset use random orientations.

SCN, SCConv and SAT have the property of orientation equivariance as long as an odd activation such as Tanh is used. SAN on the other hand does not have the property due to it's solenoidal and irrotational filters. The test will be run on the models using the TanH, identity, and ReLu activation functions, where the models with orientation equivariance are expected to perform well with the odd activation functions and poorly with ReLu.

3.2.3 Adversarial resistance (Extension)

Adversarial examples are attacks against machine learning models aimed at making the model produce an incorrect answer. These attacks do not interfere with the training of the model, or the weights after training. Instead, inputs are drawn from a dataset that a machine learning model was trained on and have small but intentional perturbations added it. It is these small perturbations that result in the incorrect answer being produced. A classic example of an algorithm that can be used to produce adversarial examples is the fast gradient sign method (FGSM) proposed by Goodfellow, Shlens, and Szegedy [57].

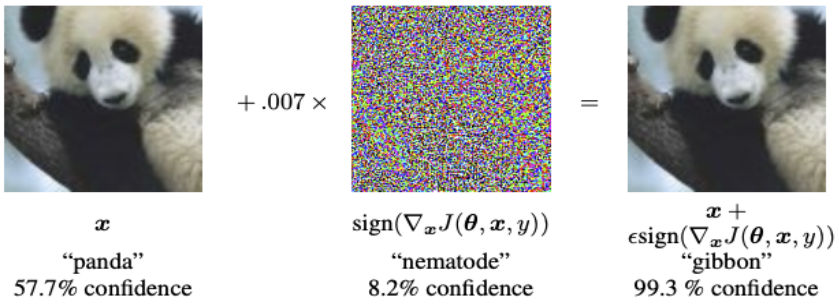


Figure 3.5: An example of untargeted fast gradient sign method in action [57].

There are two different versions of FGSM, untargeted and targeted. In the untargeted version, the objective of the algorithm is to make the model make a wrong prediction, whereas targeted FGSM aims to make the model make a specific prediction which is usually not the correct prediction [58].

The formula for untargeted FGSM is given as follows:

$$\text{FGSM}_{\text{untargeted}} : \mathbf{X}' = \mathbf{X} + \epsilon \times \text{sign}(\nabla_{\mathbf{X}} \mathbf{J}(\theta, \mathbf{X}, \mathbf{Y})) \tag{3.14}$$

In this equation, $\mathbf{X}'$ is the adversarial example, $\mathbf{X}$ is the original input, ϵ is a small constant that signifies the level of perturbation, J is the loss function, $\nabla_{\mathbf{X}}$ is the gradient with respect to $\mathbf{X}$, θ is the model being attacked, and Y is the true label. We can see that in order to produce our adversarial example $\mathbf{X}'$, we first need to compute the loss of the model with respect to the input and label. Once we have the loss, we are able to derive the direction needed to bring the model towards a minimum through backpropagation and thus achieve better performance via gradient descent. However, instead of subtracting it, we add it to the input which results in gradient ascent and the model going further from the minimum. In order to make the perturbation unnoticeable, we take only the sign of the gradient and multiply it by ϵ, the direction alone being enough to conduct the attack. This can be seen in Figure 3.5.

The formula for targeted FGSM is given as:

$$\text{FGSM}_{\text{targeted}} : \mathbf{X}' = \mathbf{X} - \epsilon \times \text{sign}(\nabla_{\mathbf{X}} \mathbf{J}(\theta, \mathbf{X}, \mathbf{Y}')) \tag{3.15}$$

The difference between targeted and untargeted is that we compute the loss with respect to a target label Y' that we want our model to output as the misclassified label. We then subtract the gradient rather than add it to decrease the loss so that the prediction moves towards Y' instead.

FGSM is known as a one-shot attack, since there is only 1 step required to generate the adversarial example. A stronger version of FGSM, also known as iterative FGSM (I-FGSM) applies the perturbation multiple times but with a smaller epsilon that is averaged over the number of iterations.

$$\text{I-FGSM}_{\text{untargeted}} : \mathbf{X'_{n+1}} = \mathbf{X'_n} + \frac{\epsilon}{\mathbf{N}} \times \text{sign}(\nabla_{\mathbf{X'_n}} \mathbf{J}(\theta, \mathbf{X'_n}, \mathbf{Y})), \quad \mathbf{X'_0} = \mathbf{X} \tag{3.16}$$

where n will run from 0 to N. The adversarial example will have to be clipped to ensure it stays within a certain range, for example, a pixel value can only be between 0 and 255. We will be using this attack in the test, since it also gives the added benefit of allowing us to visualise the strength of the attack on the model per epoch as a time series. Stronger attacks in this family do exist, such as momentum boosted I-FGSM [59], but such attacks are beyond the scope of this project.

The adversarial examples are constructed from the superpixel graphs mentioned in subsection 3.2.1 that are constructed from images that come from the MNIST dataset. Although it is possible that by changing the average pixel value in a superpixel cluster, clusters may have to be reformed, in this test we assume the pixels that make up a node clusters remain the same throughout and thus the coordinates of the cluster do not change as a result. We also assume that the attack can only affect the RGB values of the node and not the x and y coordinates. Both targeted and untargeted I-FGSM are used to construct adversarial examples.

Applying the attack on GNNs is straightforward since they only utilise features at a node level. Applying the attack on SNNs is slightly more complicated since the edge and triangle features are dependent on what the node features are. Once the gradients are calculated and the attack is applied on node level features, the edge and triangle features have to be reconstructed to use the adversarial pixel values from the updated node features.

In addition to testing each model's resistance to both targeted and untargeted attacks, another feature that will be investigated is the transferability of such attacks. One thing that has been noted is that adversarial examples often generalise, and that an adversarial example that was supposed to target a certain model also performs well on another model, even if this other model was trained on a different training set. The reason given was that models training on similar datasets end up learning similar functions, and because FGSM targets the gradient, it is able to generalise across models. We will also test the transferability of such attacks to see the similarity between the different models tested.

3.2.4 Unsupervised representational learning (Extension)

The unsupervised representational learning task is based on the Deep Graph Infomax (DGI) proposed by Veličković et al. [60] which is an approach for learning node representations in an unsupervised manner based on maximising local mutual information. The concept is based on Deep InfoMax (DIM), a framework that relies on a convolutional neural network to maximise mutual information between local features and global features in the context of image data [61]. DGI works in a similar fashion except applied to graphs.

Given a graph with a set of node features, $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$ where N is the number of nodes and $\boldsymbol{x}_i \in \mathbb{R}^K$ where K is the number of features each node has, as well as

an adjacency matrix $\boldsymbol{A} \in \mathbb{R}^{N\times N}$, the objective of DGI is to learn an encoder function $\mathcal{E} : \mathbb{R}^{N\times K} \times \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times K'}$. We denote the resulting $\mathbb{R}^{N\times K'}$ term as $\boldsymbol{H}$ where $\boldsymbol{H}$ is the set of new node embeddings $\{\boldsymbol{h}_1, \boldsymbol{h}_2, \ldots, \boldsymbol{h}_N\}$ and $\boldsymbol{h}_i \in \mathbb{R}^{K'}$. The difference between $\boldsymbol{x}_i$ and $\boldsymbol{h}_i$ is that $\boldsymbol{x}_i$ represents the features of the individual node i, while $\boldsymbol{h}_i$ represents the patch summary of the graph centered around node i. This new representation can be used for other downstream tasks such as node classification.

A readout function $\mathcal{R} : \mathbb{R}^{N\times K} \to \mathbb{R}^K$ is used to summarise the patch representation of $\boldsymbol{H}$ into a graph representation $\boldsymbol{s}$ which summarises the entire graph. As an equation, $\boldsymbol{s}$ is obtained as $\boldsymbol{s} = \mathcal{R}(\mathcal{E}(\boldsymbol{X}, \boldsymbol{A}))$.

In order to maximise local mutual information, a discriminator function $\mathcal{D} : \mathbb{R}^{N\times K'} \times \mathbb{R}^K \to \mathbb{R}^{K'}$ is used. For all patch representations $\boldsymbol{h}_i \in \boldsymbol{H}$, the discriminator $\mathcal{D}$ gives the probability that patch $\boldsymbol{h}_i$ was used to construct $\boldsymbol{s}$, with actual patches containing a higher probability score.

In order to facility noise contrastive sampling, a corruption function $\mathcal{C} : \mathbb{R}^{N\times K} \times \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times K} \times \mathbb{R}^{N\times N}$ is used to obtain negative samples from the original graph, where $\mathcal{C}(\boldsymbol{X}, \boldsymbol{A}) = \tilde{\boldsymbol{X}}, \tilde{\boldsymbol{A}}$. In a single graph setting, this corrupted graph can be obtained by permutating node features such that node j is now represented by $\boldsymbol{x}_i$, removing or adding edges with some probability p, or a mix of both. We can obtain the set of corrupted patch summaries $\tilde{\boldsymbol{H}}$ by feeding the corrupted graph through the encoder.

Lastly, we introduce the loss function $\mathcal{L} : \mathbb{R}^{K'} \times \mathbb{R}^{K'} \times \mathbb{R}^K \to \mathbb{R}$ which is defined as:

$$\mathcal{L} = \frac{1}{K' + K'} \left(\mathbb{E}_{(\boldsymbol{X},\boldsymbol{A})} \left[\sum \log \mathcal{D}(\boldsymbol{H}, \boldsymbol{s}) \right] + \mathbb{E}_{(\tilde{\boldsymbol{X}},\tilde{\boldsymbol{A}})} \left[\sum \log(1 - \mathcal{D}(\tilde{\boldsymbol{H}}, \boldsymbol{s})) \right] \right) \tag{3.17}$$

Which is based on a noise-contrastive type objective based on a Jensen-Shannon divergence and uses a standard binary cross-entropy loss between the samples from the original graphs and the corrupted graph to maximise mutual information between $\boldsymbol{h}_i$ and $\boldsymbol{s}$.

In order to ensure that the test can be applied to both GNN and SNNs, a change must be applied to the encoder function. Namely that θ, the model being tested, has to be a parameter. This is because the way patch representations are calculated will differ between GNNs and SNNs.

In GNNs, the patch representation can be obtained by a single pass of the GNN with the node features and adjacency matrix. Recall that the generalisation of the GNN propagation function can be described as $\boldsymbol{H}^{k+1} = \phi(\boldsymbol{A}\boldsymbol{H}^k\boldsymbol{W})$[1]. By replacing $\boldsymbol{H}^{k+1}$ and $\boldsymbol{H}^k$ from the GNN propagation step with our patch representation $\boldsymbol{H}$ and node features $\boldsymbol{X}$ terms respectively, we see that the GNNs fits the role of our encoder.

However, it is different in SNNs. Given $\boldsymbol{X}$ and $\boldsymbol{A}$, the model will first need to transform the graph into a simplicial complex that it can operate on, followed by aggregating the patch representation at the different complex levels back into the node patch representation.

The transformation function can be described as $\mathcal{T} : \mathbb{R}^{N\times K} \times \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times K} \times \mathbb{R}^{E\times K} \times \mathbb{R}^{T\times K} \times \mathbb{R}^{N\times E} \times \mathbb{R}^{E\times T}$ where $\boldsymbol{X}, \boldsymbol{A}$ are transformed into the node, edge and triangle

[1]This only works as we will be conducting a transductive learning task. If it were a different task such as inductive learning, a different propagation rule will have to be used, which is beyond the scope of this dissertation.

features $\boldsymbol{X}_0, \boldsymbol{X}_1, \boldsymbol{X}_2$ as well as the two boundary matrices $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ which can be used to construct all other matrices necessary for propagation.

After performing the SNN propagation step, the aggregation function, which can be explained as $\mathcal{A} : \mathbb{R}^{N\times K'} \times \mathbb{R}^{E\times K'} \times \mathbb{R}^{T\times K'} \times \mathbb{R}^{N\times E} \times \mathbb{R}^{E\times T} \to \mathbb{R}^{N\times K'}$, is applied. The aggregation function takes the node, edge, and triangle patch representations and aggregates them to a node level patch representation. This can be described by the following equation:

$$\mathcal{A} : \frac{1}{3}(\boldsymbol{H}_0 + \boldsymbol{B}_1\boldsymbol{H}_1 + \boldsymbol{B}_1\boldsymbol{W}\boldsymbol{B}_2\boldsymbol{H}_2) \tag{3.18}$$

An additional non-linearity $\boldsymbol{W}$ is applied between $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$ before being applied to $\boldsymbol{H}_2$ because $\boldsymbol{B}_1\boldsymbol{B}_2 = 0$.

Our new encoder function is described as $\mathcal{E} : \mathbb{R}^{N\times K} \times \mathbb{R}^{N\times N} \times (\mathbb{R}^{N\times K} \times \mathbb{R}^{N\times N} \to \mathbb{R}^{N\times K'}) \to \mathbb{R}^{N\times K'}$.

The full algorithm is described in the following pseudocode 9.

Algorithm 1 Deep Graph Infomax

function DGI($\boldsymbol{X}, \boldsymbol{A}, \theta$)
  for $i \leftarrow 1$ to J **do**
    $\boldsymbol{H} \leftarrow \mathcal{E}(\boldsymbol{X}, \boldsymbol{A}, \theta)$ ▷ Obtain patch representations
    $\boldsymbol{s} \leftarrow \mathcal{R}(\boldsymbol{H})$ ▷ Obtain patch summary
    $\tilde{\boldsymbol{X}}, \tilde{\boldsymbol{A}} \leftarrow \mathcal{C}(\boldsymbol{X}, \boldsymbol{A})$ ▷ Obtain negative examples
    $\tilde{\boldsymbol{H}} \leftarrow \mathcal{E}(\tilde{\boldsymbol{X}}, \tilde{\boldsymbol{A}}, \theta)$ ▷ Obtain negative patch representations
    loss = $\mathcal{L}(\boldsymbol{H}, \tilde{\boldsymbol{H}}, \boldsymbol{s})$ ▷ Calculate loss
    Do backpropagation on θ
  return $\mathcal{E}(\boldsymbol{X}, \boldsymbol{A}, \theta)$

In constructing the simplicial 2-complex, nodes, edges, and triangles are constructed based on the adjacency matrix, with the orientation of edges and triangles being constructed in the direction of increasing node index value. Edge and triangle features are formed by taking the logical-AND between node features.

We are interested in measuring how useful the generated patch representations are. This will be measured by training an MLP layer on the patch representations and measuring the accuracy of predicting node classes from the representations.

The test will be performed on the Cora citation network graph [62]. The Cora dataset consists of 2708 scientific publications classified into one of seven classes from different fields in computer science. Each publication in the dataset is described by a 0/1-valued word vector indicating the absence/presence of the corresponding word from a dictionary of 1433 unique words. A bi-direction link between publications exists if one is cited by the other. Cora is part of the Planetoid dataset in `PyTorch-geometric` which also features two other citation networks, Citeseer and PubMed. Although the test could very easily be extended to those two as well, it is outside the scope of this project.

On the other hand, a second graph called Cora-struct was added to the test. Cora-struct is a graph with a similar structure to Cora except all node features are identical and set to 1. Nodes belong to one of three classes that describe the structure of neighbours,

where nodes either belong to a tetrahedron clique, a triangle clique, or are not part of a clique complex. It has been proven that GNNs are unable to detect structures within a graph if all nodes are initialised to the same value [63]. Thus Cora-struct was added, not only as another dataset to be used to compare the performance of SNNs, but also to see if the higher order structures that SNNs operate on allow them to perform tasks that GNNs fail at.

3.3 Dataset class

The simplest way to represent a simplicial complex would be the cochain. As an object, this would consist of the three feature matrices $\boldsymbol{X}_0, \boldsymbol{X}_1$, and $\boldsymbol{X}_2$ and the two boundary matrices $\boldsymbol{B}_1$ and $\boldsymbol{B}_2$. We represent the `Cochain` object as the tuple $(\boldsymbol{X}_0, \boldsymbol{X}_1, \boldsymbol{X}_2, \boldsymbol{B}_1, \boldsymbol{B}_2)$.

We could just give the `Cochain` as input to our SNN models and then let the models process them. This, however, would be inefficient. After being given the `Cochain` as input, each model would have to perform their operations to calculate their own specialised terms, such as normalised Hodge-Laplacians and normalised boundary matrices. With each epoch, these terms will have to recalculated again and again resulting in lots of re-computation. The summary of terms required by each model is given in Table 3.1.

In order to get around this, we make use of `PyTorch`'s `InMemoryDataset` class to create dataset classes that are capable of storing datasets from tests as objects which are individually customised to each model to minimise the amount of re-computation needed. The trade-off is that a lot of memory is used to store these individually customised datasets.

Model	`SimplicialComplex`	Feature matrices	Laplacian matrices	Others
GCN	`GraphComplex`	$\boldsymbol{X}_0$	$\boldsymbol{L}_0$	
GAT	`GraphComplex`	$\boldsymbol{X}_0$	$\boldsymbol{L}_0$	
SCN	`SCNComplex`	$\boldsymbol{X}_0, \boldsymbol{X}_1, \boldsymbol{X}_2$	$\boldsymbol{L}'_0, \boldsymbol{L}'_1, \boldsymbol{L}'_2$	
SCConv	`SCConvComplex`	$\boldsymbol{X}_0, \boldsymbol{X}_1, \boldsymbol{X}_2$	$\hat{\boldsymbol{A}}_0^{\uparrow}, \hat{\boldsymbol{A}}_1, \hat{\boldsymbol{A}}_2^{\downarrow}$	$\boldsymbol{D}_1^{-1}\boldsymbol{B}_1, \boldsymbol{D}_2\boldsymbol{B}_1^{\top}\boldsymbol{D}_1^{-1}, \boldsymbol{B}_2\boldsymbol{D}_3, \boldsymbol{D}_4\boldsymbol{B}_2^{\top}\boldsymbol{D}_5^{-1}$
SAT	`SATComplex`	$\boldsymbol{X}_0, \boldsymbol{X}_1, \boldsymbol{X}_2$	$\boldsymbol{L}_0 \quad \boldsymbol{L}_2$	$\boldsymbol{L}_1^{\uparrow}, \boldsymbol{L}_1^{\downarrow}$
SAN	`SANComplex`	$\boldsymbol{X}_0, \boldsymbol{X}_1, \boldsymbol{X}_2$	$\boldsymbol{L}'_0, \boldsymbol{L}'_1, \boldsymbol{L}'_2$	$\boldsymbol{L}_1^{\uparrow}, \boldsymbol{L}_1^{\downarrow}$

Table 3.1: Table showcasing the matrices each model needs, as well as the name we refer the row-wise collection of matrices by; i.e., a `SCNComplex` consists of the tuple $(\boldsymbol{X}_0, \boldsymbol{X}_1, \boldsymbol{X}_2, \boldsymbol{L}'_0, \boldsymbol{L}'_1, \boldsymbol{L}'_2)$. All matrices in the table can be formed from terms in the cochain. In order to cut down on re-computation, these terms are calculated and saved to memory beforehand.

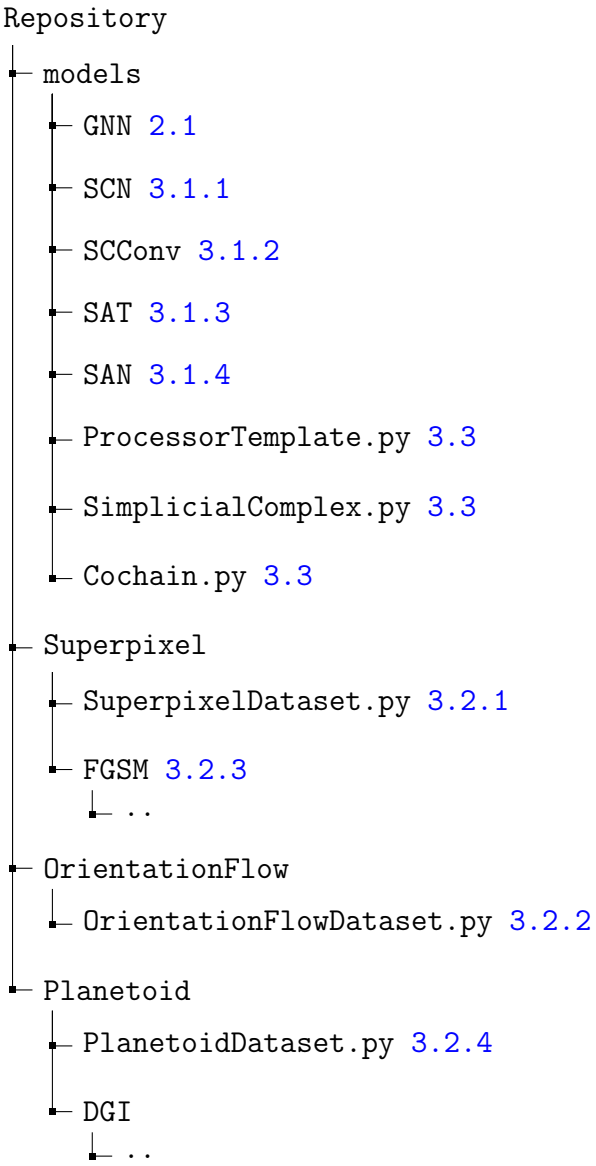
To facilitate the processing of cochain into the different `SimplicialComplex` objects, the strategy pattern is employed as seen in Figure 3.6. Each `Dataset` class, a subclass of `InMemoryDataset`, contains a `SimplicialProcessor` class. Each `SimplicialProcessor` has to implement methods required to process a `Cochain` into the required `SimplicialComplex`, store and retrieve individual `SimplicialComplex` objects to and from memory, and batch `SimplicialComplex` objects together.

This allows for other practices that make the program more time and space efficient, such as mini-batching to let training of the model scale with dataset size, the use of sparse matrices as opposed to dense matrices which reduces the computational complexity from $\mathrm{O}(N^2)$ to $\mathrm{O}(\xi N)$ where ξ is the density factor of the matrix, and the use of

`PyTorch`'s `DataLoader` class to bypass Python's Global Interpreter Lock and load data using multiple processes.

3.4 Repository overview

The two main components of this project are model implementation and dataset construction. The dataset for orientation flow 3.2.2 and DGI 3.2.4 reside in their own folder, while the dataset for superpixel 3.2.1 and adversarial superpixel 3.2.3 are in the same folder, since both use the same dataset. On the other hand, the implementation of all the models and their corresponding `SimplicialProcessor` classes can be found in a single folder.


Repository
models
GNN 2.1
SCN 3.1.1
SCConv 3.1.2
SAT 3.1.3
SAN 3.1.4
ProcessorTemplate.py 3.3
SimplicialComplex.py 3.3
Cochain.py 3.3
Superpixel
SuperpixelDataset.py 3.2.1
FGSM 3.2.3
..
OrientationFlow
OrientationFlowDataset.py 3.2.2
Planetoid
PlanetoidDataset.py 3.2.4
DGI
..

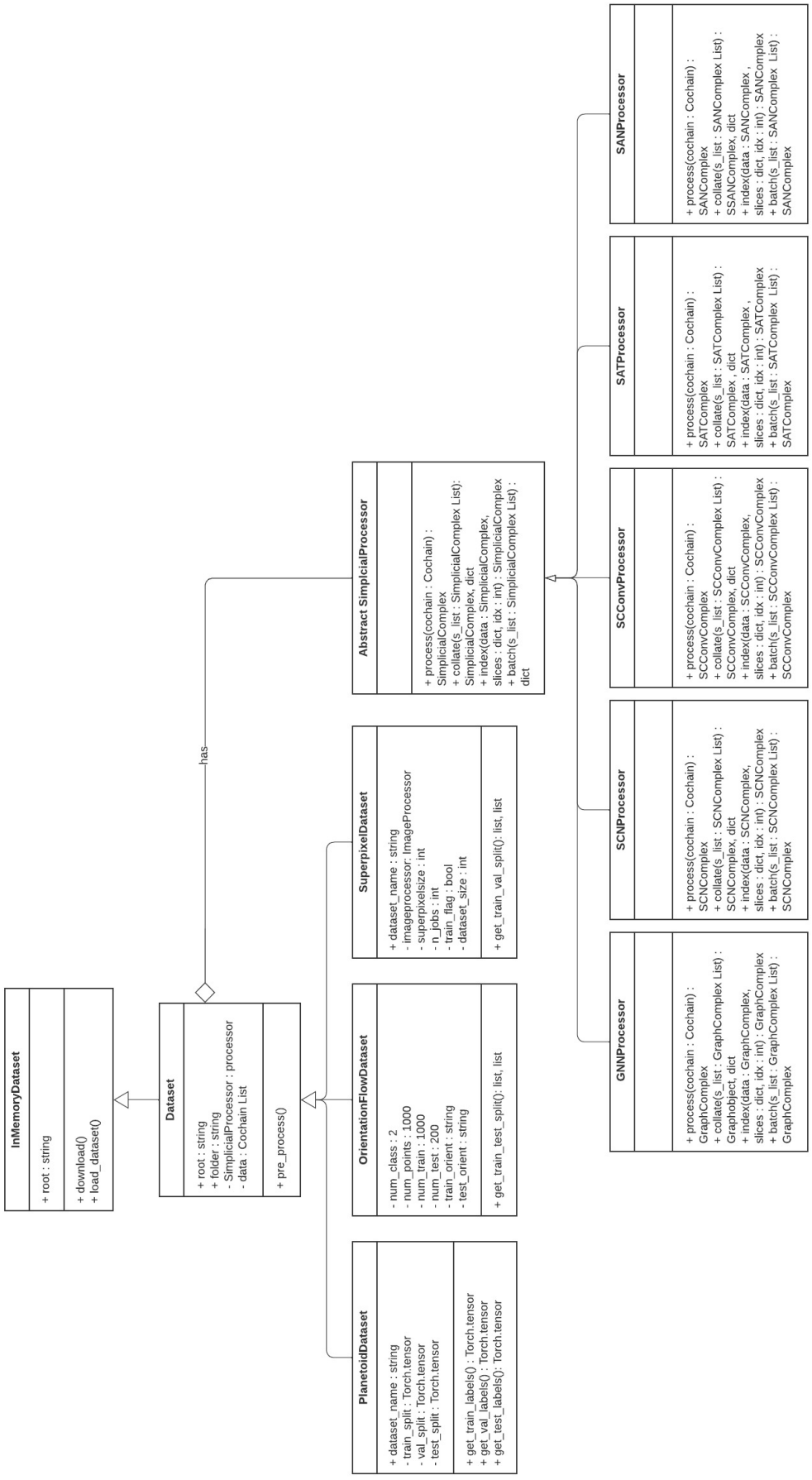


Figure 3.6: UML class diagram of dataset and processor classes

Chapter 4

Evaluation

In this chapter, I will begin with an evaluation of the models, followed by an evaluation of the models' performance in the different test implemented as part of this project.

4.1 Model statistics

Using the `InMemoryDataset` described in section 3.3, we can compare the memory footprint of the different models by analyzing their respective dataset sizes when stored in memory. This is shown in Table 4.1. As expected, the SNNs require more memory compared to the GNNs across all datasets. SAN consistently produces the datasets with the highest memory requirements, followed by SCConv and then SAT.

	In memory dataset size		
Model	CIFAR10	Trajectory Flow	Cora
GCN	2.5 Gb	-	15.7 Mb
GAT	2.5 Gb	-	15.7 Mb
SCN	3.9 Gb	663 Mb	17.3 Mb
SCConv	4.7 Gb	975 Mb	17.6 Mb
SAT	4.4 Gb	879 Mb	17.4 Mb
SAN	5.9 Gb	1.3 Gb	18.8 Mb

Table 4.1: Size of dataset for the different models

4.2 Superpixel classification

4.2.1 Setup

For the experiment, we set the superpixel algorithm SLIC to generate approximately 75 nodes. For all experiments we set a budget of 100 epochs for training, with a batch size of 32 images, using a split of 55k images for training and 5k images for validation for MNIST, and a split of 45k images for training and 5k images for validation for CIFAR10. The model is trained for 100 epochs and the model corresponding to the epoch with the best performance on the validation set is saved and tested against the test set of 10k

images. For the optimisation, we use the Adam optimiser, a weight decay of 0.0005, and a learning rate of 0.001 for all models tested. The number of feature channels for each model were set such that the total number of parameters across all models was roughly 10k. All models use a ReLU activation function except SCN which uses Leaky ReLU as in the original paper. Both GAT and SAT use two attention heads. The training procedure described is repeated across five seeds.

4.2.2 Results

We start off with the easier MNIST dataset. Most models usually perform well on this dataset and thus it acts as a sanity check to verify that the dataset, models and training loop have been implemented correctly. The results of MNIST can be seen in Table 4.2. We can see that the SNNs in general have a better performance compared to the GNNs, although GAT also has a strong performance. However, this results in a tradeoff in speed where the SNNs all take longer to compute compared to the GNNs.

	MNIST			
Model	Parameters	Train accuracy	Test accuracy	Seconds per epoch
GCN	10634	62.32 ± 2.86	63.65 ± 1.82	**11.0 ± 0.0**
GAT	9862	88.41 ± 1.66	88.95 ± 0.99	73.5 ± 2.5
SCN	10612	83.24 ± 0.52	84.16 ± 1.23	97.3 ± 0.4
SCConv	10315	88.66 ± 0.92	89.06 ± 0.47	112.0 ± 3.3
SAT	10186	91.41 ± 0.70	**92.99 ± 0.71**	276.8 ± 5.7
SAN	10582	91.36 ± 0.71	92.64 ± 0.24	258.3 ± 6.6

Table 4.2: MNIST image classification accuracy. SAT has the best performance but takes the longest time to compute. A higher test accuracy is better, while a lower seconds per epoch is better.

The CIFAR10 dataset is considerably harder, and this is reflected in the performance of the models, as seen in Table 4.3. As mentioned in the implementation chapter, we add the metrics of top 3 error and top 5 error, where top n error indicates the percentage of times the correct choice did not appear in the top n choices. Surprisingly, SCConv outperforms all other models in this test, possibly due to its unique propagation mechanism that exchanges information between adjacent simplicial dimensions.

4.2.3 Comparisons to related work

As mentioned in the proposal, classification of superpixel graphs as a test is not new and has been used in other works. We compare our test implementation and results to these other works, considering factors such as the number of parameters in the model, number of layers in the model, method for constructing graph from nodes (k-nearest neighbours or region adjacency graph), and superpixel node budget. This can be seen in Table 4.4. There are many potential factors that limit the performance of my models, such as the number of parameters, with the closest comparable implementation having 5 times more parameters, or the way the layers are arranged which is different in each paper. I do not claim that my models are state of the art or that the test was implemented without error.

	CIFAR10			
Model	Parameters	Train accuracy	Test accuracy	Top 3 error	Top 5 error
GCN	10634	28.28 ± 0.32	29.07 ± 0.95	41.54 ± 0.31	24.99 ± 0.18
GAT	9862	35.37 ± 2.44	36.45 ± 1.75	38.05 ± 1.21	23.93 ± 1.21
SCN	10612	31.54 ± 0.28	31.79 ± 0.35	39.25 ± 0.37	24.15 ± 0.37
SCConv	10315	42.24 ± 0.71	**43.80 ± 0.39**	**24.02 ± 0.59**	**11.01 ± 0.49**
SAT	10186	38.20 ± 1.02	40.20 ± 1.16	29.11 ± 1.50	15.42 ± 1.77
SAN	10582	39.56 ± 0.63	40.62 ± 0.35	28.26 ± 0.41	13.84 ± 0.15

Table 4.3: CIFAR10 image classification accuracy. SCConv achieves the best performance across the 3 accuracy metrics of test accuracy, top 3 error and top 5 error. A higher test accuracy is better, while a lower top 3 and top 5 error is better.

However, the results that were collected were not too far off from other similar papers which provides confidence in the test and the models.

Paper	Model	Dataset	Parameters	Layers	Graph construction	Nodes	Test accuracy
[17]	GCN	MNIST	101365	4	KNN	75	90.71 ± 0.22
[17]	GCN	CIFAR10	101657	4	KNN	150	55.71 ± 0.38
[17]	GAT	MNIST	110400	4	KNN	75	95.54 ± 0.21
[17]	GAT	CIFAR10	110704	4	KNN	150	64.22 ± 0.46
[64]	GAT	MNIST	55364	3	RAG	75	96.19 ± ?
[64]	GAT	CIFAR10	55364	3	RAG	75	45.93 ± ?
[53]	GAT	MNIST	?	3	KNN	75	97.11 ± ?
[12]	SCConv	MNIST	771370	1	RAG	49	91.10 ± 0.40
Mine	GCN	MNIST	10634	3	RAG	75	63.65 ± 1.82
Mine	GAT	MNIST	9862	3	RAG	75	88.95 ± 0.99
Mine	SCConv	MNIST	10315	3	RAG	75	89.06 ± 0.47

Table 4.4: Comparison with similar works.

4.3 Trajectory classification

4.3.1 Setup

For this test, all models use the same hyperparameters. Each model uses 4 layers, with the residual size set to 32 for all layers. Following the 4 layers, we take the absolute of the output and do a mean-pooling to make it orientation invariant. We then pass this through two MLP layers before finally taking the softmax of the output. For all experiments, we set a budget of 100 epochs for training, with a batch size of 4. Since there is no validation set, the model corresponding to the epoch with the best performance on the training set is saved and used for testing. For the optimisation, we once again use the Adam optimiser, a weight decay of 0.0005, and a learning rate of 0.001 for all models tested. The training procedure described is repeated across five seeds. The resulting graphs produced had 932.74 ± 8.82 nodes and 2730.27 ± 27.43 edges.

Since SAN is not orientation equivariant, I decided to also create and test an orientation equivarient version of it dubbed combined-SAN (CSAN) which features the orientation

equivariant attention mechansim of SAT but with the harmonic filter of SAN. CSAN is only featured in this test. I also added the results that were reported from another paper in which the test was conducted in a similar environment. The results can be found in Table 4.5.

	Activation function			
Model	Identity	ReLu	Tanh	Orientation equivariant
SCN	53.10 ± 2.27	49.70 ± 2.77	52.80 ± 3.11	✓
SCCONV	62.80 ± 3.11	50.80 ± 1.63	62.30 ± 3.97	✓
SAT	**92.90 ± 2.22**	49.70 ± 0.60	93.80 ± 1.33	✓
SAN	53.30 ± 2.54	**55.80 ± 4.69**	55.40 ± 4.68	
CSAN	91.25 ± 0.43	50.50 ± 0.10	83.40 ± 0.72	✓
MPSN [56]	82.6 ± 3.0	50.0 ± 0.0	**95.2 ± 1.8**	✓

Table 4.5: Trajectory classification accuracy. Models that use odd activation functions perform better on the test set, while models that use an activation function that break the invariance, like ReLu, perform similar to randomly.

4.3.2 Results

We see that the models perform very similar to randomly when ReLu is used, and perform better when the odd Tanh and Identity activation functions are used. Though curiously SAN performs the best when using ReLu. It is also the only model that performs statistically better than randomly with ReLu. One would expect that SAN would perform similar to random across all three activation functions. Potentially, models that are inherently non-invariant might play by different rules compared to models that are inherently invariant but made non-invariant via the activation function. This could be confirmed by implemented and testing another non-invariant model but that would be beyond the scope of the project.

4.4 Adversarial resistance

4.4.1 Setup

We mostly use the same setup in this test as with superpixel classification, with a small change to the size of the dataset. Here, a split of 10k images for training and 2k images for validation from MNIST is used, with another 2k images for testing and generating adversarial graphs. Models are trained for 200 epochs and once again, the model corresponding to the epoch with the best performance on the training set is saved and used for testing. Adversarial graphs are generated via I-FGSM over 250 epochs with a perturbation of 0.001 per epoch.

4.4.2 Untargeted attack results

We can see the drop in performance of the different models in Table 4.6 which displays the drop in performance compared to the test set, and Figure 4.1 which helps us visualise the relative drop in performance of the models compared to their initial starting accuracies.

		Accuracy at set epoch intervals				
Model	Initial accuracy	50 epochs	100 epochs	150 epochs	200 epochs	250 epochs
GCN	56.37 ± 2.24	6.68 ± 0.72	1.07 ± 0.38	0.30 ± 0.30	0.16 ± 0.17	0.10 ± 0.11
GAT	73.92 ± 5.66	26.96 ± 3.61	5.25 ± 1.37	1.34 ± 0.49	0.38 ± 0.29	0.13 ± 0.12
SCN	63.46 ± 2.39	**60.20 ± 2.62**	**57.31 ± 3.16**	**54.02 ± 3.86**	**50.96 ± 4.45**	**47.61 ± 4.99**
SCConv	79.16 ± 2.11	44.78 ± 3.35	14.55 ± 3.75	74.71 ± 2.14	2.08 ± 1.40	1.05 ± 1.011
SAT	81.29 ± 3.81	59.23 ± 4.87	31.32 ± 6.40	13.27 ± 4.23	4.92 ± 1.86	2.18 ± 0.64
SAN	81.83 ± 1.44	71.74 ± 2.90	59.11 ± 6.91	41.18 ± 5.90	28.78 ± 7.07	19.75 ± 6.33

Table 4.6: Performance of models when subjected to untargeted I-FGSM attack. The smaller the decrease from the initial accuracy, the more resistant the model is.

The SNN models are a lot more robust against adversarial attacks compared to the GNNs. However, it is interesting to note that the model's performance on the MNIST do not correlate with their robustness against attacks. SCN, which performed the weakest among the SNNs, has the strongest resistance while there is a sizeable difference in resistance between SAT and SAN although both have similar performance on MNIST.

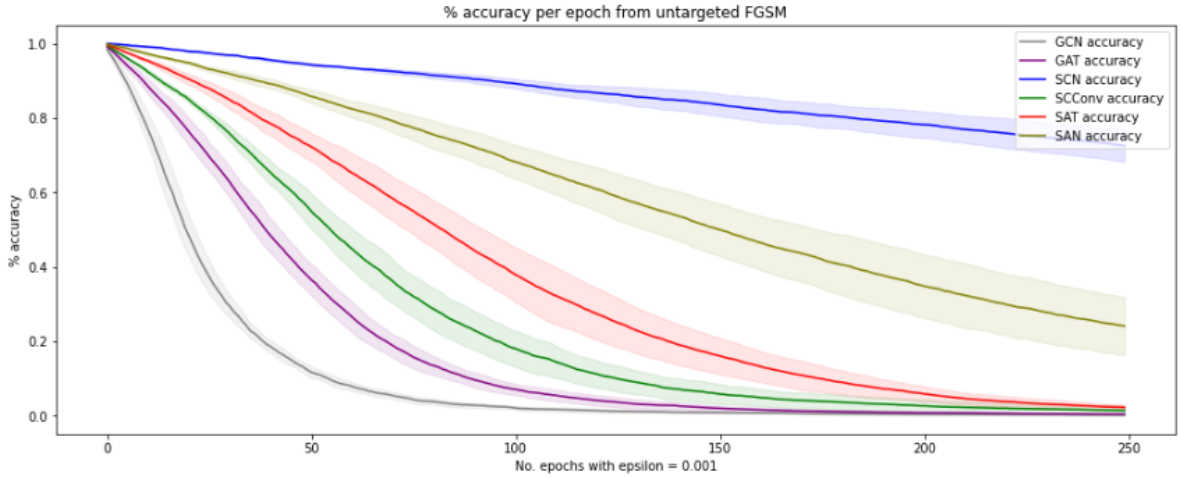


Figure 4.1: Decrease in accuracy percentage per epoch. If GCN starts off with an accuracy of 60.0, within 20 epochs it drops by half to about 30.0.

We can see the magnitude of the gradients of the models in Figure 4.2. With the size of the magnitude proportional to the model's resistance. GCN which has the earliest peak, performs the worst while SCN which has almost no gradient magnitude, is the most resistant. This correlation is expected, since FGSM targets the gradient of the model using fixed epsilons, thus a larger gradient means a bigger decrease in performance.

The reason why SCN and SAN have the two lowest gradients is primarily due to the use of the harmonic filter, which both models use, though other reasons may also exist. If we were to change the normalisation operation to something else, such as leaving the Laplacian un-normalised or using SCConv's normalisation technique, the magnitude of the gradients will increase and result in a model that is less resistant. A single run of SCN with SCConv's normalisation can be seen in Figure 4.2. However, this finding also raises questions about the properties that the harmonic filter provides, for example why it results in models that are harder to attack.

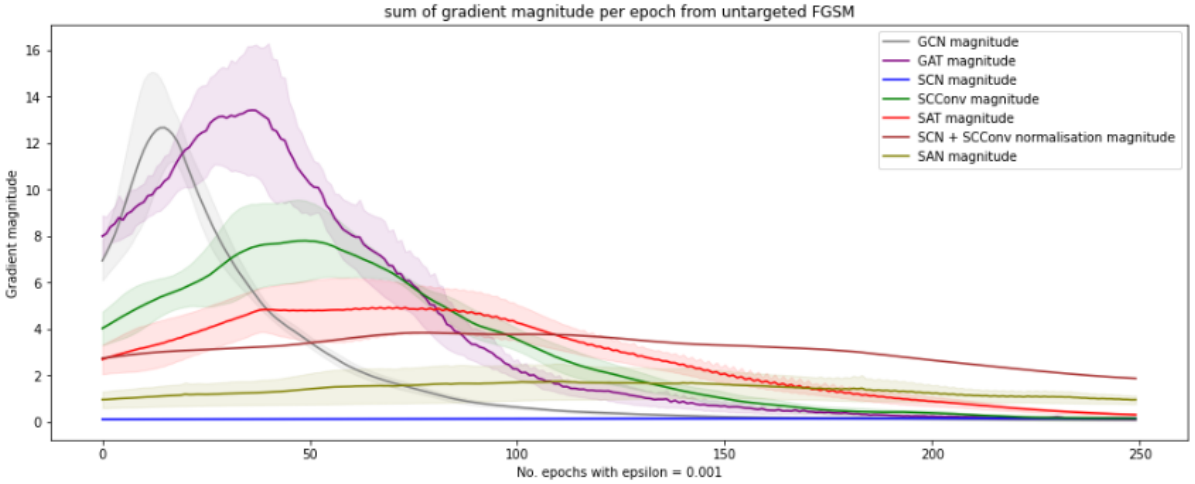


Figure 4.2: Magnitude of gradient per epoch with the addition of SCN using SCConv's normalisation in brown. Note that the magnitude of the gradient is greater than before.

4.4.3 Targeted attack results

We see a similar pattern when measuring the resistance of models to targeted attacks, where SCN is the most resistant followed by SAN. This can be seen in Table 4.7.

	Percentage of targeted mispredictions at set epoch intervals				
Model	50 epochs	100 epochs	150 epochs	200 epochs	250 epochs
GCN	37.24 ± 1.77	50.37 ± 1.51	55.22 ± 1.81	57.51 ± 1.89	58.67 ± 2.14
GAT	18.87 ± 2.25	35.43 ± 2.29	43.53 ± 1.54	47.50 ± 1.34	49.40 ± 0.93
SCN	4.85 ± 0.31	**5.74 ± 0.45**	**6.87 ± 0.82**	**7.85 ± 1.01**	**8.86 ± 1.29**
SCConv	10.09 ± 1.52	20.53 ± 3.37	26.78 ± 2.51	30.38 ± 1.90	32.39 ± 1.56
SAT	7.43 ± 0.87	16.17 ± 2.15	24.11 ± 2.67	30.09 ± 2.24	33.98 ± 1.63
SAN	**3.60 ± 0.60**	6.12 ± 1.87	9.46 ± 2.95	12.82 ± 3.92	15.48 ± 4.67

Table 4.7: Performance of models when subjected to targeted I-FGSM attack. The smaller the increase, the more resistant the model is.

4.4.4 Transferability attack results

The last metric to consider is the transferability of attacks. As mentioned in the implementation chapter, subsection 3.2.3, models trained on similar datasets end up learning similar functions. Thus hopefully we can see the similarities between models using this metric, where similar models should result in more transferable attacks. To generate the adversarial graphs, each model is subjected to targeted I-FGSM for 100 epochs with a perturbation of 0.001 per epoch. The other models are then given these adversarial graphs as input and one can measure the impact is has on their performance. The same pretrained models from earlier are reused in this test.

We see the results of the decrease in accuracy in Table 4.8. SCConv, SAT, and SAN all claim at least one instance of having the most transferable attack. As one can see from the results, it does appear that models that are similar result in better transferability.

	Accuracy of different models after transfer attack					
Model	GCN	GAT	SCN	SCConv	SAT	SAN
GCN	-	61.94 ± 1.93	62.69 ± 1.93	71.64 ± 0.79	77.13 ± 3.77	80.19 ± 0.31
GAT	32.13 ± 1.45	-	63.02 ± 2.07	75.17 ± 1.84	78.19 ± 4.34	80.82 ± 0.21
SCN	23.94 ± 0.06	54.03 ± 3.05	-	71.30 ± 0.85	76.97 ± 4.58	80.11 ± 0.68
SCConv	**22.49 ± 3.04**	57.23 ± 3.01	62.62 ± 2.09	-	75.66 ± 4.42	**79.22 ± 0.84**
SAT	27.28 ± 1.86	**52.99 ± 1.86**	62.59 ± 2.04	71.43 ± 2.29	-	80.80 ± 0.94
SAN	23.26 ± 3.28	53.39 ± 3.59	**62.14 ± 1.93**	**65.67 ± 3.55**	**75.02 ± 4.54**	-

Table 4.8: Performance of models when subjected to targeted I-FGSM attack. The lower the accuracy, the more transferable the attack. As an example, compared to the other models, SCConv produces adversarial graphs that have the greatest effect on GCN.

We can see attacks generated from SAT affect GAT the most, which is to be expected since both use attention mechanisms. We can also see the converse is true, where models that are distant result in worse transferability. GAT consistently produces the least transferable attacks against all convolution based models. It is unexpected to see GCN outperforming GAT when attacking SAT, though there is no significant difference between their performances. In other models such as SCN and SAN, there is also no significant difference in performance when comparing the top attacks generated by the other models, where the performances are all very similar. This could be due to the fact that they are very resistant in the first place and thus the differences are not as obvious.

	Percentage of targeted mispredictions of different models after transfer attack					
Model	GCN	GAT	SCN	SCConv	SAT	SAN
GCN	-	**6.94 ± 1.79**	3.92 ± 0.51	**6.46 ± 2.30**	**3.90 ± 0.65**	2.22 ± 0.54
GAT	10.90 ± 1.93	-	3.77 ± 0.21	3.36 ± 0.47	2.58 ± 0.53	2.00 ± 0.06
SCN	9.01 ± 0.06	5.19 ± 0.22	-	3.84 ± 0.57	2.76 ± 0.34	2.41 ± 0.19
SCConv	**11.49 ± 0.29**	5.87 ± 0.73	**4.15 ± 0.12**	-	3.65 ± 0.19	**2.58 ± 0.39**
SAT	8.47 ± 0.24	5.70 ± 0.18	3.85 ± 0.38	4.27 ± 0.60	-	2.47 ± 0.06
SAN	9.84 ± 0.94	5.29 ± 0.57	4.10 ± 0.20	5.18 ± 1.23	3.08 ± 0.49	-

Table 4.9: Performance of models when subjected to targeted I-FGSM attack. Models that have the most transferable attacks on other models are bolded.

Next, we consider how successful the transfer attacks were at making the model predict the targeted label. This is shown in Table 4.9. Surprisingly, the results are very different from Table 4.8, with GCN and SCConv producing attacks that results in the most targeted misclassification.

4.5 Unsupervised representational learning

4.5.1 Setup

For the last test, we measure the ability of the models at the task of unsupervised representational learning. The models are trained on Cora and Cora-struct for a maximum of 500 epochs, though this was cut short if loss does not improve over 20 consecutive epochs. Corrupted graphs are obtained by swapping node features (node corruption), adding or removing edges with some probability p (edge corruption), or both simultaneously.

Once again, for the optimisation, we use the Adam optimiser, a weight decay of 0.0005, and a learning rate of 0.001 for all models tested.

Once the patch representations are obtained, a single MLP layer is used to determine the usefulness of the representations by recording the accuracy of the MLP at predicting a node's class from the given representation. It is trained for 100 epochs using a train-val-test split of 140, 400, and 1000 for Cora, and 60, 300, and 1000 for Cora-struct. For optimisation, we use the Adam optimiser, a weight decay of 0 and a learning rate of 0.01.

Patch representations are obtained over 5 different seeds and 3 different corruption functions for all models. With these representations, the MLP is repeated over 50 seeds for each representation and the mean is taken as the prediction accuracy. The mean and standard deviation of these means are given as the final result.

4.5.2 Cora results

The results from the test on the Cora dataset are shown in Table 4.10. In the original paper, GCN achieves a score of 82.3 ± 0.6 on the Cora dataset when only node corruption is applied [22], a result that has been reproduced in this test. Unfortunately, no numerical results are given for edge corruption and the combined corruption.

	Corruption function		
Model	Node corruption	Edge corruption with $p = 0.005$	Node + Edge corruption with $p = 0.005$
GCN	**82.55 ± 0.60**	**81.72 ± 0.25**	80.56 ± 0.39
GAT	78.75 ± 1.17	77.40 ± 0.57	79.90 ± 1.81
SCN	61.04 ± 4.98	48.54 ± 1.12	60.98 ± 1.08
SCConv	82.02 ± 1.27	80.51 ± 1.18	80.90 ± 0.49
SAT	81.28 ± 0.76	81.41 ± 0.69	**81.55 ± 0.97**
SAN	79.01 ± 0.64	78.54 ± 0.76	78.62 ± 1.34

Table 4.10: Classification accuracy of Cora classes

The expectation when creating this test was that SNNs, that can exploit features from higher order structures, should be able to perform better than GNNs as they should have the potential to learn better local information. This does not appear to be the case. In fact, an MLP trained on the raw node features has a performance of 69.3 ± 1.4, which is better than the embeddings produced by SCN over all three corruption types.

A possible explanation could be that SNNs suffer to a greater extent from the heterophily problem. The heterophily problem is a problem inherent in GNNs where models perform poorly on graphs where nodes tend to be connected to nodes belonging to other classes [65]. The initial simplicial complex from Cora is created via the clique complex. This would have resulted in the grouping of nodes that are otherwise unrelated into common cliques which will affect the model's ability to learn from them, potentially making the learning problem harder right from the beginning and unfortunately was not something I considered when making the test. However, only creating cliques for nodes that have the same class may not be ideal as this could result in information leakage.

4.5.3 Cora-struct results

Even if the heterophily problem affects SNNs to a greater extent compared to GNNs, it should not result in problems for the Cora-struct dataset, since node classes are based on the type of clique the node is a part of. Since all nodes have initial features set to 1, there is no reason to do node corruption. However, I will still add it to confirm that the test has been implemented correctly, in which case all models should expect to see accuracies of about 33%. The main focus will be on the results from edge corruption and whether SNNs outperform GNNs at this task.

From the results in Table 4.11, we see that this is the case. With the exception of SAT, all other SNNs are able to perform better than random. This result also allows us to achieve one of the original goals of this test, which was to determine if SNNs are able to detect structures within a graph, a task that the GNNs fail at.

	Corruption function	
Model	Node corruption	Edge corruption with $p = 0.005$
GCN	32.10 ± 1.46	25.54 ± 3.53
GAT	32.90 ± 1.18	33.48 ± 0.90
SCN	33.24 ± 1.74	47.28 ± 0.10
SCConv	34.56 ± 1.14	46.55 ± 5.04
SAT	32.57 ± 1.93	23.95 ± 2.39
SAN	33.82 ± 1.65	**50.45 ± 5.70**

Table 4.11: Classification accuracy of Cora classes

The performance of SAT being worse than GCN and GAT is also a result that is unexpected but not unbelievable. This is due to the nature of the attention mechanism, which focuses more on the individual connection between simplices rather than the topology of the graph. As such the task of substructure detection is something it is not suited for.

4.6 Summary of tests

To summarise this chapter, I performed all the tests promised in the implementation chapter and used them to distinguish the performance of the different models. The tests produced results that were expected, as well as results that were not expected and a discussion about why these unexpected results occurred was done.

A table of the best performing models on each test in shown in Table 4.12.

Test	Best model
MNIST	SAT
CIFAR10	SCConv
Trajectory	SAT
Adversarial	SCN
Cora	GCN
Cora-struct	SAN

Table 4.12: Table of datasets as well as the model with the best performance on each test. Coincidentally, every single SNN appears at least once.

Chapter 5

Conclusion

5.1 Contributions

My project far exceeded the success criteria set out in the project proposal. I have managed to produce a benchmarking software that addresses the current shortage of universal benchmarks when it comes to evaluating SNN performance.

The software comes equipped with four different test that asks different questions about the performance of the model and are rigorously designed to statistically separate model performance. Once again, the four tests involve graph classification, trajectory classification, adversarial resistance, and unsupervised representational learning. Before this, adversarial resistance and unsupervised representational learning have never been considered in evaluating the performance of SNNs, but as we have seen, these tests also provide meaningful insight into the performance of the models.

Secondly, as part of the software I have implemented 6 models (2 GNN and 4 SNN), 2 more than originally set out in the proposal. This includes the novel model SAT which is the first SNN to incorporate the attention mechanism, while still fulfilling the properties of an admissible architecture.

Unfortunately, one of the potential extensions set out in the proposal was to release the software as an open-source library, something that I have been unable to do. However, this was a very optimistic extension in the first place and I believe I more than exceeded expectations in the other two extensions.

5.2 Further work

Despite my success in developing the benchmarking software, some of the results produced are definitely interesting and warrant further research. These include questions such as:

1. What are the properties of the harmonic filter why is it so resistant to adversarial attacks?
2. Given that SNNs pay closer attention to higher order structures, are SNNs more adversely affected by the heterophily problem, a problem that is inherent in GNNs?

3. In the same vein, are SNNs more adversely affected by the over-smoothing problem, a problem I avoided by using a hierarchical architecture?

There are also potential avenues of improvement when designing new SNNs. For example, the attention mechanism in SAT and SAN operate on a single simplicial complex dimension, while the convolutioal propagation mechanism in SCConv interacts with adjacent simplicial complex dimensions. A possible extension of SAT/SAN would be to create a model that incorporates the attention mechanism but also provides interactions with adjacent dimensions.

Lastly, there is also room for further work when it comes to designing datasets for SNNs. There are currently only 2 original datasets designed specifically for SNNs that are derived from real-life data, those being missing data imputation based on research papers and ocean drifter trajectories around the island of Madagascar. Creating a larger real-world dataset such as one based on Reddit on Wikipedia would definitely advance the field.

5.3 Lessons learnt

This was certainly a challenging project to undertake. Starting the project was daunting as I had never interacted with GNNs or SNNs and definitely lacked the knowledge regarding them. The field of SNNs was also a new and rapidly developing field at the time of writing the proposal, meaning that the resources available were limited. However, with constant effort, it became easier to understand the papers as well as the underlying concepts and motivations. By the end, I can confidently say I have a decent understanding of GNNs, SNNs, and good testing practices. The process was also remarkably enjoyable and I had a lot of fun thinking about and writing the different tests and models. For example, at one point I was considering trying to treat a neural network as a simplicial complex. Motivated by the adage 'neurons that wire together fire together', the task would have been to predict which neurons would activate.

Beyond the research aspect of the project, I have also gained experience in the software engineering aspect. The importance of having structured code and using design principles from the start quickly became apparent as it made adding new models and tests easier. For example, the model SAN and the trajectory flow test were only added in early April, after the paper on SAN was released, but integrating them into the codebase was not difficult thanks to the way I has structured my code.

In the end, I am glad I undertook this project. It not only helped me gain a better understanding of graph representational learning, But I also managed to create something that could be useful to the research community and is something I can be proud of.

Bibliography

[1] J. Zhou, G. Cui, S. Hu, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, "Graph neural networks: A review of methods and applications," *AI Open*, vol. 1, pp. 57–81, 2020.

[2] B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko, "A gentle introduction to graph neural networks," *Distill*, 2021. https://distill.pub/2021/gnn-intro.

[3] A. Sperduti, "Encoding labeled graphs by labeling raam," in *Advances in Neural Information Processing Systems* (J. Cowan, G. Tesauro, and J. Alspector, eds.), vol. 6, Morgan-Kaufmann, 1993.

[4] C. Goller and A. Kuchler, "Learning task-dependent distributed representations by backpropagation through structure," in *Proceedings of International Conference on Neural Networks (ICNN'96)*, vol. 1, pp. 347–352 vol.1, 1996.

[5] M. Gori, G. Monfardini, and F. Scarselli, "A new model for learning in graph domains," in *IJCNN*, 2005.

[6] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Trans. Neural Networks*, vol. 20, no. 1, pp. 61–80, 2008.

[7] J. Bruna, W. Zaremba, A. Szlam, and Y. LeCun, "Spectral networks and locally connected networks on graphs," in *ICLR*, 2014.

[8] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," in *NIPS*, 2016.

[9] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *ICLR*, 2017.

[10] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *ICML*, 2017.

[11] S. Ebli, M. Defferrard, and G. Spreemann, "Simplicial neural networks," 2020.

[12] E. Bunch, Q. You, G. Fung, and V. Singh, "Simplicial 2-complex convolutional neural nets," 2020.

[13] Y. Chen, Y. R. Gel, and H. V. Poor, "Bscnets: Block simplicial complex neural networks," 2021.

[14] C. Bodnar, F. Frasca, Y. G. Wang, N. Otter, G. Montúfar, P. Liò, and M. Bronstein, "Weisfeiler and lehman go topological: Message passing simplicial networks," 2021.

[15] N. Glaze, T. M. Roddenberry, and S. Segarra, "Principled simplicial neural networks for trajectory prediction," in *ICML*, 2021.

[16] A. D. Keros, V. Nanda, and K. Subr, "Dist2cycle: A simplicial neural network for homology localization," 2021.

[17] V. P. Dwivedi, C. K. Joshi, T. Laurent, Y. Bengio, and X. Bresson, "Benchmarking graph neural networks," 2020.

[18] W. D. Heaven, "Ai is wrestling with a replication crisis," Nov 2020.

[19] W. Hu, M. Fey, M. Zitnik, Y. Dong, H. Ren, B. Liu, M. Catasta, and J. Leskovec, "Open graph benchmark: Datasets for machine learning on graphs," 2020.

[20] "Artificial intelligence - google scholar metrics," 2022.

[21] L. Giusti, C. Battiloro, P. Di Lorenzo, S. Sardellitti, and S. Barbarossa, "Simplicial attention neural networks," 2022.

[22] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," 2018.

[23] K. Ghiasi-Shirazi, "Generalizing the convolution operator in convolutional neural networks," *Neural Processing Letters*, vol. 50, pp. 2627–2646, apr 2019.

[24] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[25] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, L. Wang, G. Wang, J. Cai, and T. Chen, "Recent advances in convolutional neural networks," 2017.

[26] L. Chua and L. Yang, "Cellular neural networks: Theory," *Circuits and Systems, IEEE Transactions on*, vol. 35, pp. 1257 – 1272, 11 1988.

[27] B. Nica, "A brief introduction to spectral graph theory," May 2018.

[28] X. Chen, "Understanding spectral graph neural network," September 2021.

[29] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," 2017.

[30] M. R. Dale, *Spatial Graphs*, p. 191–221. Cambridge University Press, 2017.

[31] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and P. S. Yu, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 32, p. 4–24, Jan 2021.

[32] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," 2018.

[33] S. Barbarossa and S. Sardellitti, "Topological signal processing over simplicial complexes," *IEEE Transactions on Signal Processing*, vol. 68, pp. 2992–3007, 2020.

[34] L.-H. Lim, "Hodge laplacians on graphs," 2019.

[35] N. Glaze, T. M. Roddenberry, and S. Segarra, "Principled simplicial neural networks for trajectory prediction," *CoRR*, vol. abs/2102.10058, 2021.

[36] A. Muhammad and M. Egerstedt, "Control using higher order laplacians in network topologies," in *Proc. of 17th International Symposium on Mathematical Theory of Networks and Systems, Kyoto*, pp. 1024–1038, 2006.

[37] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," *CoRR*, vol. abs/1912.01703, 2019.

[38] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *CoRR*, vol. abs/1903.02428, 2019.

[39] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, I. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy, "Scipy 1.0-fundamental algorithms for scientific computing in python," *CoRR*, vol. abs/1907.10121, 2019.

[40] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, F. Boulogne, J. D. Warner, N. Yager, E. Gouillart, and T. Yu, "scikit-image: image processing in python," *PeerJ*, vol. 2, p. e453, Jun 2014.

[41] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," 1 2008.

[42] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, sep 2020.

[43] C. da Costa-Luis, "tqdm: A fast, extensible progress meter for python and cli," *Journal of Open Source Software*, vol. 4, p. 1277, 05 2019.

[44] R. Olfati-Saber and R. Murray, "Consensus problems in networks of agents with switching topology and time-delays," *IEEE Transactions on Automatic Control*, vol. 49, no. 9, pp. 1520–1533, 2004.

[45] M. T. Schaub, A. R. Benson, P. Horn, G. Lippner, and A. Jadbabaie, "Random walks on simplicial complexes and the normalized hodge 1-laplacian," *SIAM Review*, vol. 62, p. 353–391, Jan 2020.

[46] M. T. Schaub, A. R. Benson, P. Horn, G. Lippner, and A. Jadbabaie, "Random walks on simplicial complexes and the normalized Hodge 1-Laplacian," *SIAM Review*, vol. 62, no. 2, pp. 353–391, 2020.

[47] M. T. Schaub, Y. Zhu, J. Seby, T. M. Roddenberry, and S. Segarra, "Signal processing on higher-order networks: Livin' on the edge ... and beyond," *CoRR*, vol. abs/2101.05510, 2021.

[48] R. Achanta, A. Shaji, K. Smith, A. Lucchi, P. Fua, and S. Süsstrunk, "Slic superpixels compared to state-of-the-art superpixel methods," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 34, no. 11, pp. 2274–2282, 2012.

[49] F. Monti, D. Boscaini, J. Masci, E. Rodolà, J. Svoboda, and M. M. Bronstein, "Geometric deep learning on graphs and manifolds using mixture model cnns," *CoRR*, vol. abs/1611.08402, 2016.

[50] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010.

[51] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research),"

[52] M. V. den Bergh, X. Boix, G. Roig, and L. V. Gool, "SEEDS: superpixels extracted via energy-driven sampling," *CoRR*, vol. abs/1309.3848, 2013.

[53] J. Long, Z. yan, and H. chen, "A graph neural network for superpixel image classification," *Journal of Physics: Conference Series*, vol. 1871, p. 012071, apr 2021.

[54] K. Oono and T. Suzuki, "Graph neural networks exponentially lose expressive power for node classification," *arXiv:1905.10947*, 2019.

[55] C. Cai and Y. Wang, "A note on over-smoothing for graph neural networks," *arXiv:2006.13318*, 2020.

[56] C. Bodnar, F. Frasca, Y. G. Wang, N. Otter, G. Montúfar, P. Lio, and M. Bronstein, "Weisfeiler and lehman go topological: Message passing simplicial networks," *ICML*, 2021.

[57] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," 2015.

[58] W. Wei, L. Liu, M. Loper, S. Truex, L. Yu, M. E. Gursoy, and Y. Wu, "Adversarial examples in deep learning: Characterization and divergence," 2018.

[59] Y. Dong, F. Liao, T. Pang, H. Su, J. Zhu, X. Hu, and J. Li, "Boosting adversarial attacks with momentum," 2018.

[60] P. Veličković, W. Fedus, W. L. Hamilton, P. Liò, Y. Bengio, and R. D. Hjelm, "Deep graph infomax," 2018.

[61] R. D. Hjelm, A. Fedorov, S. Lavoie-Marchildon, K. Grewal, P. Bachman, A. Trischler, and Y. Bengio, "Learning deep representations by mutual information estimation and maximization," 2018.

[62] P. Sen, G. M. Namata, M. Bilgic, L. Getoor, B. Gallagher, and T. Eliassi-Rad, "Collective classification in network data," *AI Magazine*, vol. 29, no. 3, pp. 93–106, 2008.

[63] Z. Chen, L. Chen, S. Villar, and J. Bruna, "Can graph neural networks count substructures?," *CoRR*, vol. abs/2002.04025, 2020.

[64] P. H. C. Avelar, A. R. Tavares, T. L. T. da Silveira, C. R. Jung, and L. C. Lamb, "Superpixel image classification with graph attention networks," 2020.

[65] J. Zhu, Y. Yan, L. Zhao, M. Heimann, L. Akoglu, and D. Koutra, "Beyond homophily in graph neural networks: Current limitations and effective designs," *Advances in Neural Information Processing Systems*, vol. 33, 2020.

Appendix A

Project Proposal

The proposal is available from the next page onwards.

Benchmarking simplicial neural networks

1 Introduction

Simplicial neural networks (SNN) are a generalization of graph neural networks (GNN) that work on simplicial complexes. A simplicial k-complex is a set of simplexes considered as a collective unit with the largest dimension of any simplex being k (a k-simplex) and a k-simplex is a geometric object with (k+1) vertices. So a point is a 0 simplex, an edge is a 1-simplex and a triangle a 2-simplex.

While GNNs are limited to pair-wise interactions between vertices, SNNs can encode higher order information between them [1]. A typical message-passing GNN works by aggregating information from neighbouring nodes through a graph convolution function and in doing so, each node can learn the structure and features of neighbouring nodes [2, 3]. A SNN works the same way but instead of nodes, each simplex is able to learn the structure and features of neighbouring simplexes. Other works on the application of SNNs have also been done on trajectory prediction, edge flow classification and isomorphism [4]. However, the field of SNNs is still relatively new and there are no standard benchmarks or benchmarking tests available.

The main benefit of a SNN as compared to a GNN is its ability to model higher order structures within a graph [5]. Since the label of a graph can be the result of its higher order interactions, I propose to creating a benchmarking test that will allow us to compare the SNN's ability to classify graphs. Once the message propagation and aggregation has been completed, we can combine the feature vectors of all the individual simplexes to get a combined feature vector that can be used to represent the entire complex. The same methodology of combining features can be done for GNNs except we combine the feature vector of the nodes. Since in both cases, we can use the combined feature vector to classify the graph, we will also be able to compare the SNN to a GNN baseline.

Thus, the focus of my project will be to create a benchmarking software with the main benchmarking test to compare the classification ability of the different SNN models against a GNN baseline. The main datasets used to test the classification ability will be the Modified National Institute of Standards and Technology database (MNIST) and Canadian Institute For Advanced Research-10 (CIFAR10) dataset. The MNIST dataset is a dataset consisting of handwritten digits from 0 to 9 while the CIFAR10 dataset consists of images from 10 classes including birds, planes and frogs. I will convert the datasets into simplicial 2-complexes and use this as the benchmarking test for the different models. Each model will act as a convolution layer and the resulting combined feature vector will be fed into a standard artificial neural network with 10 output nodes (1 for each class). I will then evaluate the models using different metrics such as classification accuracy.

2 Work to be done

2.1 Dataset

Although the CIFAR10 and MNIST datasets are readily available through public sources, work will have to be done to transform them into a form suitable for simplicial neural networks to work on.

The first step involves transforming the images into superpixel graphs using the superpixel algorithm simple linear iterative clustering (SLIC). Superpixel algorithms are used to group pixels into nodes representing perceptually meaningful regions, such as a region of similar intensity [6]. These superpixel graphs will then have to be converted into simplicial complexes. The current proposed method is as follows:

The superpixel graph will first be converted into region adjacency graphs in a form similar to the works done by Avelar et al [7]. The region adjacency graph will then be converted into a simplicial 2-complex by turning every node into a 0-simplex, edge into a 1-simplex and triangle into a 2-simplex. Should a better method of converting superpixel graphs into simplicial complexes be found during research, it will be used instead.

2.2 Models

There currently are only 2 papers regarding using SNN's for classifying simplicial complexes. These are the models created by Bunch et al [8] as well as Ebli et al [1]. The project will involve implementing these models myself and evaluating them using the proposed benchmark software.

While the implementation of a GNN is not in the scope of this project, it will also be interesting to see how my implementation of the 2 SNNs compare against state-of-the-art GNNs. For this purpose, I will be using existing an implementations from the PyTorch geometric[1] as a baseline to compare the SNNs against.

2.3 Evaluation of models

Along with the 2 SNN models stated above, I plan on including at minimum one GNN architecture as a baseline to compare the classification accuracy of the 2 SNN models against. Similar to the SNN models, this GNN will be used as a convolutional layer and the combined feature vector will be fed to the same artificial neural network with 10 output nodes.

The MNIST dataset benchmark would be used to confirm that the SNN models have been implemented correctly. Using a simplicial convolutional layer on the MNIST dataset yields a high accuracy result. In the test by Bunch et al using the MNIST dataset, the SNN outperformed the GNN at classifying the digits [8]. On the other hand, the CIFAR10 dataset has not been used as a benchmark test on a SNN yet so it will be interesting to see the result.

For the purpose of this dissertation and to evaluate the benchmarking software, the two SNN models will be benchmarked against the GNN model.

[1] https://github.com/pyg-team/PyTorch_geometric

3 Success criteria

The project will be considered a success if:

1. I Successfully create a test that can be used to benchmark simplicial neural networks from the MNIST and CIFAR10 dataset
2. I Successfully implement the 2 SNN models by Ebli et al and Bunch et al for use in the benchmarking software.
3. I Successfully create a benchmark software that is able to test the 2 SNN models along with one GNN model.

4 Possible extensions

Should there be additional time, possible extensions that could be considered include

1. Adding 1 or 2 new SNN models to benchmark, possibly be extending the current implementations or referencing a new model should such a paper be published.
2. Adding an additional benchmark test, such as edge flow classification.
3. Build an open-source library containing the implemented SNNs and benchmarking tests.

5 Starting point

I have read several papers on GNNs and SNNs but have neither worked with such architectures, nor studied anything related to graph theory before.

I have worked on using an artificial neural network to classify images in TensorFlow. However, I will also need to work with PyTorch in order to use the PyTorch geometric library as well as scikit in order to use SLIC superpixel segmentation. I have not used PyTorch, PyTorch geometric or scikit before.

6 Resources

I will be using my own laptop for development. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure. I will be committing my source code to GitHub and will write my dissertation on Overleaf. Backups will be made weekly on Google drive. Should there be a hardware failure, I will switch to working on MCS machines.

For training and evaluating the models, I will use GPUs provided by the Computational Biology Group. This means I will also need a CL account.

7 Timeline and milestones

1. **21 Oct - 3 Nov (Mich 3 - 4)**

   Research about simplicial complexes and how to create them. Read up about Hodge laplacian/ spectral graph theory. Write a function/file that can create simplicial complexes from the MNIST dataset.

   Deliverables:

   - Project codebase repository
   - Function/file that creates a simplicial 2-complex graph from an image and works on the MNIST dataset.

2. **4 Nov - 17 Nov (Mich 5 - 6)**

   Extend the above function/file to also work on the CIFAR10 datasaet. Begin testing with GNN baseline.

   Deliverables:

   - **Milestone** - Function/file that creates a simplicial 2-complex graph from an image and works on both datasets achieved.
   - Complete the design of a pipeline that can perform a full test on the GNN baseline.

3. **18 Nov - 1 Dec (Mich 7 - 8)**

   Slack period, start working on first SNN

   Deliverables:

   - **Milestone** - Success criteria of implementing benchmark pipeline achieved.
   - Skeleton of first SNN

4. **2 Dec - 8 Dec (Christmas break week 1)**

   Begin implementation of first SNN.

   Deliverables:

   - Skeleton of first SNN committed

5. **16 Dec - 29 Dec (Christmas break week 2 - 3)**

   Complete implementation of first SNN. Begin implementation of second SNN.

   Deliverables:

   - First SNN completed and tested.
   - Skeleton of second SNN committed.

6. **30 Dec - 12 Dec (Christmas break week 4 - 5)**

   Complete implementation of second SNN. Prepare progress report and presentation.

   Deliverables:

   - Second SNN completed and tested.
   - Prepare progress report and presentation.

7. **13 Dec - 19 Dec (Christmas break week 6)**

   Slack period and completion of progress report and presentation

   Deliverables:

- Second SNN completed and tested.
- Completion of progress report and presentation.

8. **20 Jan - 2 Feb (Lent 1 - 2)**

   Full evaluation and data collection.

   Deliverables:

   - **Milestone** - Success criteria of implementing and evaluating 2 separate SNN models achieved.
   - Document evaluation techniques and results.

9. **3 Feb - 16 Feb (Lent 3 - 4)**

   Slack period and room for extensions.

   Deliverables:

   - Any outstanding evaluation deliverables that have not been completed before this.

10. **17 Feb - 2 Mar (Lent 5 - 6)**

    Completion of extensions and evaluation of extensions. Completion of Introductory chapter

    Deliverables:

    - Any extensions that were started should be completed at this point.
    - Introductory chapter of the dissertation

11. **3 Mar - 16 Mar (Lent 7 - 8)**

    Completion of preparation and implementation chapter of the dissertation

    Deliverables:

    - Preparation chapter of the dissertation.
    - Implementation chapter of the dissertation.

12. **17 Mar - 30 Mar (Easter break 1 - 2)**

    Completion of evaluation and conclusion chapter of the dissertation

    Deliverables:

    - Evaluation chapter of the dissertation.
    - Conclusion chapter of the dissertation.

13. **31 Mar - 13 Apr (Easter break 3 - 4)**

    Completion of appendix chapter of the dissertation

    Deliverables:

    - Appendix chapter of the dissertation.
    - Draft of completed dissertation.

14. **14 Apr - 27 Apr (Easter break 5 - 6)**

    Dissertation is to be finalized along with the codebase of the project.

    Deliverables:

    - Codebase of the project.

- Completed dissertation.

15. **28 Apr - 11 May (Easter 1 - 2)**

    Slack period

    Deliverables:

    - **Milestone** - Completed dissertation.

References

[1] Stefania Ebli, Michaël Defferrard, and Gard Spreemann. "Simplicial Neural Networks". In: (2020).

[2] Jie Zhou et al. "Graph neural networks: A review of methods and applications". In: *AI Open* 1 (2020), pp. 57–81. ISSN: 2666-6510.

[3] Vijay Prakash Dwivedi et al. "Benchmarking Graph Neural Networks". In: (2020).

[4] Cristian Bodnar et al. "Weisfeiler and Lehman Go Topological: Message Passing Simplicial Networks". In: (2021).

[5] Cristian Bodnar et al. "Weisfeiler and Lehman Go Cellular: CW Networks". In: (2021).

[6] Radhakrishna Achanta et al. "SLIC Superpixels Compared to State-of-the-Art Superpixel Methods". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 34.11 (2012), pp. 2274–2282.

[7] Pedro H. C. Avelar et al. "Superpixel Image Classification with Graph Attention Networks". In: (2020).

[8] Eric Bunch et al. "Simplicial 2-Complex Convolutional Neural Nets". In: (2020).