Università
Ca'Foscari
Venezia

Master's Degree Programme
in **Computer Science - Data Management and Analytics**

Master's Thesis

Ca' Foscari
Dorsoduro 3246
30123 Venezia

Kernelized Convolutional Operator for Graph Neural Networks

Relatore
Chiar.mo prof. Andrea Torsello

Laureando
Daniele Crosariol
Matricola 847056

Anno Accademico
2021/2022

Contents

Abstract

In recent years the very application on machine learning involve Neural Network model based on tensor, but there exists many fields of research that are suited with data structure based on graph, like proteins and chemistry compounds. In this context, the Graph Neural Network (GNN) models have been developed, they are tipically based on message passing which lose information about the structure since the convolution is defined in a non-structural way. The GNNs are still not fully explored in the machine learning field, for this reason in this thesis we present a generalized Graph Neaural Network model that use Kernel operator and Convolution on graphs in order to approach a graph classification problem. We want to apply the convolution using a graph as mask to be compared with the subgraphs of the main graph using a graph Kernel, in this way we maintain the data structure and we don't lose information about it. The Goal of this project is to understand if with this approach the model can be trained and learn the patterns of the graph and produce relevant result.

Chapter 1

Introduction

Nowadays neural network models based on graphs, also known as Graph Neural Network or GNN, find application in many fields such as medicine, physics, chemistry, computer vision and economy. These neural network are used to make a prediction, either to build a regressor or to approach a classification problem.
Generally, when making predictions through neural networks, the input of the network is in the form of a matrix, but now the neural networks that use graphs as input data structure are growing more and more.
An interesting property of the graph-based neural network model is that they can be used for fast and scalable semi-supervised classification.
The graph data structure can have different level of complexity: the simplest form is a sequence of nodes, while the more complex forms can be trees, cyclic or acyclic graphs, which can be oriented or non-oriented, completely connected or sparse.
The use of these data structures are useful when the problem we are trying to solve is focused on relationships between entities. In fact, we can model the problem as a graph where the nodes are the entities and the edges represent the relationships between such entities. An example of such problem can be a set of atoms in the field of chemistry. Another application example is the search for similar documents, where the nodes are documents and the edges represent the similarity or the distance between documents.
The graph domain applications can be divided into two classes: the first class takes into consideration the whole graph, while the second focuses on singles nodes.
In the first class of applications, the objective function is independent from the single nodes and normally we have a unique output which predict the graph class; on the other hand, in the second class every node computes its output depending on itself and on its neighbourhood. In this thesis we face

a problem belonging to the first class of application.
In this thesis we want to use the convolution on graphs in order to extract information and predict the ground truth of the graph. Convolution is done using a graph kernel which compute a similarity function between the input graph and a mask graph. The goal is to check if the model is able to learn the convolution mask that it will be the feature to extract.
Common GNNs have limit of descriptive capacity since they lose information during the process of computation, moreover in case of embedding masks we would be in a continuous space. For this reason we redefine the convolution in structure space so we can avoid information drop out.
In the first chapter of this thesis we explain the state of art, how GNNs models work in general and what they are used for; in the second chapter we describe the most used GNNs; in the third chapter we introduce our model, explaining how it works. The final chapters will describe the analysis of our model and the comparison with the most used models.

Chapter 2

How GNNs work

GNNs belongs to machine learning models based on neural networks and they can be used to solve classification or regression problems.
The goal of GNNs is to extract information from a graph, exploiting the relationships that may occur between its nodes, edges or both of them, marked with a label.
Once the information has been extracted, GNNs generally try to predict the label of a node or, as in our case, the label of the whole graph.
Like classic neural networks, GNNs also use a learning algorithm in order to train the weights of the network.
There are many categories of GNN that use different approaches to model information, in this section we will explain some of the main approaches.
At the beginning, traditional machine learning applications handles graph structured data using a pre-processing phase which transforms the graph structured information into a simpler representation like vectors of reals.
In the following paragraph we describe the following type of GNNs:

- Message passing: this kind of GNN use nodes to gather information from their neighbor using a local transition function, after few iteration of this function nodes reach a stable state and fire the output.

- Graphs Auto Encoders: GAE encode graph information into a new feature space and they also train the encoding function in order to predict the final label.

- Convolutional GNN: Convolutional GNN use the convolution to extract the information from the graph.

2.1 Message Passing

The first approach to GNN we want to show evolved as an extension of random walk models and recurrent neural networks [6].
This method extended the theory of random walks introducing a learning algorithm in the pre-processing phase using an information diffusion mechanism called *message passing*: a graph is processed by a set of units, each one corresponding to a node of the graph, which are linked according to the graph connectivity. The units update their states and exchange information until they reach a stable equilibrium. The output of a GNN is then computed locally at each node on the base of the unit state [6].
Given a graph $G = (V, E)$, where $x \in V$ are the nodes and $(x, y) \in E$ are the edges between nodes. Moreover, the graph may have a set of labels associated to each node, edge or the graph itself. We can see the nodes as the objects and the arcs as the relationship between them. Then we can set a state x_n to each node, where n refers to its neighborhood. The state x_n represents the concept of the neighborhood and it can be described with different level of distance in terms of links between nodes and it can use or not any labels. Each node is associated with a local transition function f_w that compute the state x_n, and a local output function g_w which describe the decision.

The two functions can be defined as follows:

$$\begin{aligned} x_n &= f_w(l_n, l_{co[n]}, x_{ne[n]}, l_{ne[n]}) \\ o_n &= g_w(x_n, l_n) \end{aligned} \tag{2.1}$$

where l_n is the label of node n, $l_{co[n]}$ are the labels of the edges with vertex n, $x_{ne[n]}$ is the state of the neighbours of n, $l_{ne[n]}$ are the labels of the neighbours of n.
The transition and the output function may depend on the node and it is possible to implement a different nodes function for each node. For simplicity we will explain the case where all the nodes have the same transition and output functions.
Let's define x as the vector containing the states of all the nodes; o as the vector containing the the outputs of all the nodes, l as the vector containing all the labels and l_N as the vector containing all the node labels. We can then rewrite the transitional and output function in a global form instead of local form.

$$\begin{aligned} x &= F_w(x, l) \\ o &= G_w(x, l_N) \end{aligned} \tag{2.2}$$

where F_w is the global transition function and G_w is the global output function. We define a map $\varphi_w : \mathcal{D} \to \mathbb{R}^m$, which takes a graphs input and returns an output $\emptyset_n$ for each node.
The Banach fixed point theorem provides a sufficient condition for the existence and uniqueness of the solution of a system of equations. [6]
In order to build the GNN model, three steps must be followed:

1. Solve the equation of the transition and output functions;

2. Provide a learning algorithm to adapt the function;

3. Implement the function Fw and Gw;

As we can see in figure 2.1, starting from a simple graph we can create the encoding network replacing the nodes label with the transition function which forward the message to the neighbors and after some steps they fire the output with the output function.
The network at the bottom of image 2.1 shows how it behave in a feed forward manner: starting from a T_0 to T_m, each layer correspond to a time instant and at the end it firing the output. At each layer we have all the corresponding unit of the encoding network and the arcs are the connection with the neighbors which corresponds to the message passed at time T_i.

In order to solve the first step we already mentioned that the Banach's fixed point theorem ensure the existence and the uniqueness of the solution for the equation, moreover has been seen that the transition function can follow an iterative scheme computing the state at each time t_i.

$$x(t+1) = F_w(x(t), l) \tag{2.3}$$

where $x(t)$ denotes the th-iteration of x. Since $x(t)$ is the state updated by the transition F_w at time t-1, the Jacobi iterative method is used for solving the nonlinear equations. Both the outputs and the states are computed by iterating as follow:

$$\begin{aligned} xn(t+1) &= fw(l_n, lcon, x_{ne[n]}(t), l_{ne[n]}) \\ on(t) &= gw(x_n(t), l_n), n \in N \end{aligned} \tag{2.4}$$

When we talk about learning algorithm, we consider the parameters to estimate. In this case the parameter w needs to be estimated so that φ_w approximates the data in the learning data set.

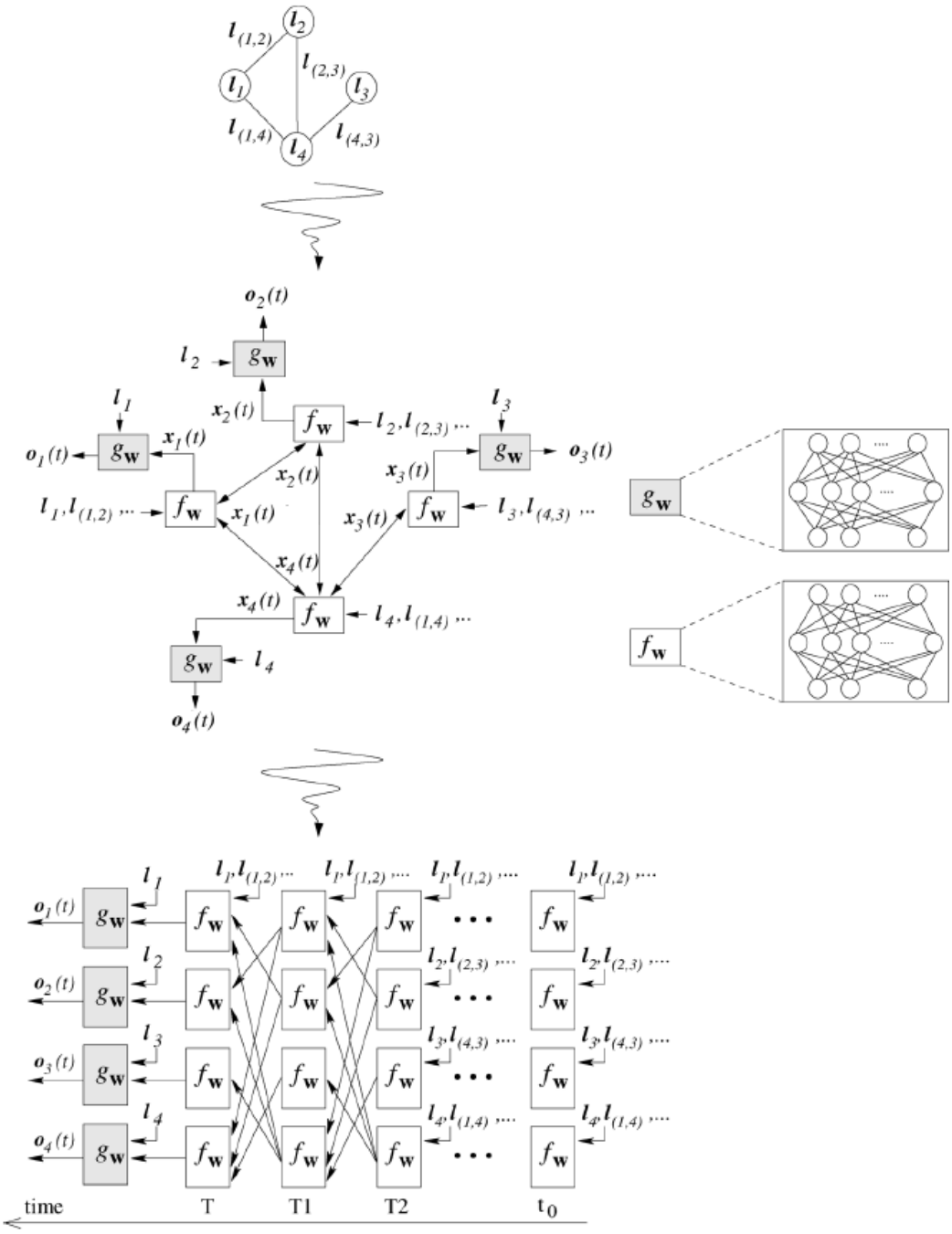

Figure 2.1: Starting from a simple graph we obtain the encoding network and the final model.

$$\mathcal{L} = (G_i, n_{i,j}, t_{i,j}) | G_i = (N_i, E_i) \in \mathcal{G}; n_{i,j} \in N_i; t_{i,j} \in \Re^m, 1 <= i <= p, 1 <= j <= q_i$$

where qi is the number of supervised nodes in G_i. For graph based problem qi=1 and a single node is chosen with the most relevance.
The learning task is the minimization of the following quadratic cost function

$$e_w = \sum_{i=1}^{p} \sum_{j=1}^{q_i} (t_{i,j} - \varphi_w(G_i, n_{i,j}))^2$$

additional terms can be added at this cost function in order to control other properties of the model. This algorithm is based on the very known gradient descent strategy and is composed of the following steps

1. The states are iteratively updated by Fw(5) until at time T they approach the fixed point of solution (2) such that x(t) = x
2. The gradient is computed
3. The weights are updated according to the gradient computed in step

2.2 Graph Auto Encoders

The next type of GNNs that we are going to describe are the graph auto-encoders (GAEs). These models use deep neural architectures to build an encoding and a decoding function: these two functions have the role of mapping nodes into a latent feature space and decode graph information from latent representations.
GAEs are useful when we miss information about the graph and we want to extract or recreate them. The common objectives of these GNN are the reconstruction of PPMI or adjacency matrix and to learn the generative distribution of data. These objectives can be seen from two perspectives, network embedding and graph generation.

The information graph reconstruction belongs to the network embedding, which generally is a vector representation of a node. GAEs learn Network embedding using an encoder which extract the network embeddings and the decoder to enforce it in order to preserve the graph topological information. The encoder and decoder can be implemented with different technique: at the beginning multi-layer perceptrons was used to build GAEs for network embedding learning, then it evolved using more complex methods.

The following example is an extract of the work Variational Graph Auto-Encoders which uses graph embeddings in order to reconstruct the adjacency matrix. [4]
The encoder consists of two layers GCN, which takes the form:

$$Z = enc(X; A) = Gconv(RELU(Gconv(A, X; \theta_1)); \theta_2) \tag{2.5}$$

where Z denotes the network embedding matrix of a graph.
The decoder reconstruct graph adjacency matrix decoding node relational information from their embeddings. It is defined as it follows:

$$\hat{A}_{v;u} = dec(z_v; z_u) = \sigma(z_v^T z_u);$$

Then the training is done by minimizing the negative cross entropy between the real adjacency matrix A and the reconstructed adjacency matrix $\hat{A}$ optimizing the variationa lower bound

$$L = E_{q(Z|X;A)}[logp(A|Z)]KL[q(Z|X; A)||p(Z)]$$

where KL is the Kullback-Leibler divergence function which measures the distance between two distributions, $p(Z)$ is a Gaussian prior $p(Z) = \prod_{i=1}^{n} p(z_i) = \prod_{i=1}^{n} N(z_i|0, I)$, $p(A_{ij} = 1|z_i, z_j) = dec(z_i, z_j) = \sigma(z_i^T z_j)$, $q(Z|X, A) = \prod_{i=1}^{n} q(z_i|X, A)$ with $q(z_i|X, A) = N(z_i|\mu_i, diag(\sigma_i^2))$ The mean vector μ_i is the i^{th} row of an encoder's outputs defined by Equation 1.5.

As we said before GAEs can be used to learn the data distribution of the graphs. This can be made by using the encoder that maps the graph into an hidden representation while the decoder extracts the graph given an hidden representation. It requires multiple graphs in order to train and learn data distribution.
This approach can be easily implemented with the use of Generative Adversarial Network (GAN), which is based in a competitive game between a Generator and Discriminator. In this game the Generator tries to learn the data distribution and generates fake example that Discriminator has to predict as fake.

2.3 Convolutional GNN

In this section we want to show some of the most common ConvGNN: [3], GraphSage [2], DCNN [1]. These GNN can be divided in two classes: spec-

tral based convolution and spatial based convolution.

2.3.1 Spectral Based Convolution

The approach we want to show is the spectral based ConvGNN. Spectral-based approaches define graph convolutions by introducing filters from the perspective of graph signal processing. This method takes into account the assumption of undirected graph in order to use the normalized graph Laplacian matrix that is the undirected graph matrix representation. The matrix is positive symmetric defined as following:

$$L = I_n - D^{1/2}AD^{1/2}$$

The matrix L is factorized as

$$L = U\delta U^T$$

where

$$U = [u_0; u_n; ...; u_n] \in R$$

is the matrix of eigenvectors ordered by eigenvalues and δ is the diagonal matrix of eigenvalues i.e.

$$\delta_{ii} = \lambda_i$$

The $U^TU = I$ is the orthonormal space formed by the eigenvectors.
Then a function is defined in order to map the vector of feature nodes into the orthonormal space.
If we consider the function of graph Fourier transform to a signal x, it is defined as $F(x) = U^Tx$ and the inverse function as $F^{-1}(\hat{x}) = U\hat{x}$, where $x \in R$ is the feature vector of all nodes of a graph and $\hat{x}$ represents the resulted signal from the graph Fourier transform.
$F(x)$ projects the input graph signal to the orthonormal space where the basis is formed by eigenvectors of the normalized graph Laplacian.
The signal $\hat{x}$ are the coordinates of the graph signal in the new space so that the input can be represented as $x = \sum_{i=1} \hat{x}_i u_i$ which is $F^{-1}(\hat{x})$. The convolution is then defined applying a filter $g \in R$ as it follows:

$$X * Gg = F^{-1}(F(x) \odot F(g)) = U(U^Tx \odot U^Tg)$$

where $\odot$ denotes the element-wise product. Spectral-based ConvGNNs all follow this definition. The key difference lies in the choice of the filter g.

2.3.2 GCN

The GCN uses the spectral base approach described in this section in order to address a node classification problem.
The learning phase of this model consists on smoothing the label information over the graph by using a graph Laplacian regularization term in the loss function:

$$\mathcal{L} = \mathcal{L}_0 + \lambda \mathcal{L}_{reg}, \text{ with } \mathcal{L}_{reg} = \sum i, j A_{ij} ||f(X_i) - f(X_j)||^2 = f(X)^T \Delta f(X)$$

where $\mathcal{L}_0$ denotes the supervised loss with respect to the labeled part of the graph, $f(\dot{})$ can be a neural network or a differentiable function, λ is a weighing factor and X is a matrix of node feature vectors X_i. $\Delta = D - A$ denotes the unnormalized graph Laplacian.
The graph structure is mapped using a neural network model $f(X, A)$ and trains on a supervised target $\mathcal{L}_0$ for all nodes with labels.
The GCN model uses a multi-layer convolution with the following layer-wise propagation rule:

$$H^{(l+1)} = \sigma(\tilde{D}^{-1/2} \tilde{A} \tilde{D}^{-1/2} H^{(l)} W^{(l)})$$

$\tilde{A} = A + I_N$ is the adjacency matrix of the graph with added self-connections, I_N is the identity matrix, $\tilde{D}_{ij} = \sum_j \tilde{A}_{ij}$ and $W^{(l)}$ is a layer-specific trainable weight matrix. $\sigma(\dot{})$ denotes an activation function, such as the ReLU. $H^{(l)} \in R^{NxD}$ is the matrix of activations in the lth layer; H(0) = X.
Moreover the GCN model adopts a spectral convolution on graph defined as the multiplication of a signal $x \in R^n$ with a filter $g_\theta = diag(\theta)$ parameterized by $\theta \in R^N$ in the Fourier domain. Since the filter parameter takes into account only the diagonal, the convolution is simplified as

$$g_\theta * x \approx U g_\theta U^T x$$

where U is the matrix of eigenvectors that we have seen in the chapter before. This GCN model resolves some computation problem due to the decomposition e multiplication of eigenvector which have a time complexity of $O(n^3)$. The convolution is then speeded up with the following tricks: first one, the number of eigenvalues is approximated as $\lambda_{max} \approx 2$ so that it constrains the number of parameters further to address overfitting; second one, to minimize the number of operations per layer.
Once we apply the tricks, the filtering expression results as follows:

$$g_\theta * x \approx \theta(I_N + D^{-1/2} A D^{-1/2}) x$$

with a single parameter $\theta = \theta_0' = -\theta_1'$. The repeated application of this operator can lead to numerical instabilities when used in deep neural network model. For this reason a renormalization trick is adopted: $I_N + D^{-1/2}AD^{-1/2} \rightarrow \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$.

In order to build the GCN model it is necessary first to calculate $\hat{A} = \tilde{D}^{-1/2}\tilde{A}\tilde{D}^{-1/2}$ as a pre-processing step, then it is enough to build the forward model

$$Z = f(X, A) = softmax(\hat{A}ReLU(\hat{A}XW^{(0)})W^{(1)})$$

2.3.3 Spatial Based Convolution

The next two models belong to the spatial convolution category. The idea is similar to the message passing: every node propagates information along edges to the central node of the graph. It can be easily done through a convolution that convolve to the center. In this way the model can derive the updated representation for the central node.

DCNN

We start introducing the Diffusion Convolutional Neural Network [1] which make use of convolution as a diffusion process. The diffusion graph convolution consists in mapping node information to adjacent with a certain transition probability, after several iterations the information distribution reaches the stable state.
Considering a node classification problem, the propagation is defined as

$$H^{(k)} = f(W^{(k)} \odot P^k X)$$

where $f()$ is an activation function and P is the probability transition matrix. The model is completed by a dense layer that connects Z to the output layer. A classic stochastic gradient descent algorithm is used to backpropagate the error and to modify the weights of different layers.
DCNN concatenates $H^{(1)}$,$H(2)$,...,$H^{(K)}$ together as the final model outputs. This model has some limitation of scalability since DCNNs are realized as a series of operations on dense tensors, moreover there are limitation on locality due the fact that it is designed to capture local behavior.

Algorithm 1: GraphSAGE embedding generation (i.e., forward propagation) algorithm

Input : Graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$; input features $\{\mathbf{x}_v, \forall v \in \mathcal{V}\}$; depth K; weight matrices $\mathbf{W}^k, \forall k \in \{1, ..., K\}$; non-linearity σ; differentiable aggregator functions $\text{AGGREGATE}_k, \forall k \in \{1, ..., K\}$; neighborhood function $\mathcal{N} : v \rightarrow 2^{\mathcal{V}}$

Output : Vector representations $\mathbf{z}_v$ for all $v \in \mathcal{V}$

$\mathbf{h}_v^0 \leftarrow \mathbf{x}_v, \forall v \in \mathcal{V}$;
for $k = 1...K$ **do**
  for $v \in \mathcal{V}$ **do**
    $\mathbf{h}_{\mathcal{N}(v)}^k \leftarrow \text{AGGREGATE}_k(\{\mathbf{h}_u^{k-1}, \forall u \in \mathcal{N}(v)\})$;
    $\mathbf{h}_v^k \leftarrow \sigma\left(\mathbf{W}^k \cdot \text{CONCAT}(\mathbf{h}_v^{k-1}, \mathbf{h}_{\mathcal{N}(v)}^k)\right)$
  end
  $\mathbf{h}_v^k \leftarrow \mathbf{h}_v^k / \|\mathbf{h}_v^k\|_2, \forall v \in \mathcal{V}$
end
$\mathbf{z}_v \leftarrow \mathbf{h}_v^K, \forall v \in \mathcal{V}$

Figure 2.2: GraphSage model algorithm

Graph SaGe

Next model we want to talk about is again a spatial convolution and it is the GraphSaGe model [2], which take name from the two main steps of the model: Sample and Aggregate. Instead of using matrix factorization, this model affects node features in order to learn an embedding function that generalizes to unseen nodes.
An interesting characteristic of GraphSage is that it learns both the topological structure of each node neighborhood and its node feature distribution.
This model trains a set of aggregator functions that learn to aggregate feature information from a node's local neighborhood.

The first step of this model is to learn the parameters of the aggregation step. This learning step is made by an unsupervised method applying graph-based loss function to the output representations z_u. During this phase the parameters aggregator functions and the weight of matrix W^k are tuned using a stochastic gradient descent.
The loss function rewards try to make sure of a difference representation between a node and its neighborhood so that disparate nodes are highly distinct:

$$J_G(z_u) - log(\sigma(z_u^T z_v)) - Q \cdot \mathbb{E}_{v_n \sim P_n(v)} log(\sigma(-z_u^T z_u))$$

where u is a node in the neighborhood of v on fixed-length random walk, σ is the sigmoid function, P_n is a negative sampling distribution, and Q defines the number of negative samples.
As we said before GraphSage doesn't operate with matrix, in fact it applies

functions to neighbor node which aren't sorted nor have natural order. For this reason it would be ideal that the aggregation function would be symmetric.
We report the Mean Aggregator function proposed in literature which satisfy the symmetric property and is nearly equivalent to the convolutional propagation rule of the GCN. This function can be used in substitution to the aggregation and concatenation step of the algorithm reported above.
It takes the elementwise mean of the vectors in $\{h_u^{k-1}, \forall u \in \mathcal{N}(v)\}$ in the following way.

$$h_v^k \leftarrow \sigma(W \cdot MEAN(\{h_v^{k-1}\} \cup \{h_u^{k-1}, \forall u \in \mathcal{N}(v)\})$$

This aggregator differs from the one in the algorithm above since it skips the concatenation. This convolutional aggregator does concatenate the node's previous layer representation h_v^{k-1} with the aggregated neighborhood vector $h_{\mathcal{N}(v)}^k$.

We have seen how to train the parameters, so now we can describe how the algorithm works. At each iteration nodes aggregate information from their local neighbors and go forward with the iteration. In detail: at each step of the K iterations, h_v^k denotes the representation of node v and at first it aggregates the representation of its immediate neighborhood features into the vector $h_{\mathcal{N}(v)}^k$ then concatenates the node's current representation. The concatenated vector is fed through a fully connected layer with nonlinear activation function σ, which transforms the representations to be used at the next step of the algorithm.

Chapter 3

Our Model

In this section we describe our GNN model, which consists on a convolutional layer obtained by computing a graph kernel between subgraph of the graph to be classified, and severel structural masks. The features obtained by this convolutional layer are then pooled in a pooling layer and the results fed to a multi-layer perceptron (MLP).
The goal of our model is to learn structural feature from examples. In our approach the structural features are small graphs that are presents as subgraphs in the dataset and are useful for prediction. The idea is that it is not always possible to know which is the best features that contain the right information. We are interested in the structure of the information.
In detail we have a set of Graphs $G = (V, E)$ associated to a ground truth label; every node $n \in N$ is labeled and the edges $e \in E$ are oriented. The final goal is to build a GNN model that is able to predict the label of the graphs.

Figure 3.1 show the pipeline of the model. The colors show the flow for each mask.

The steps of our GNN model are the following:

1. Convolution Layer

2. Global Pooling Layer

3. Forward MLP

4. Learning (backpropagation)

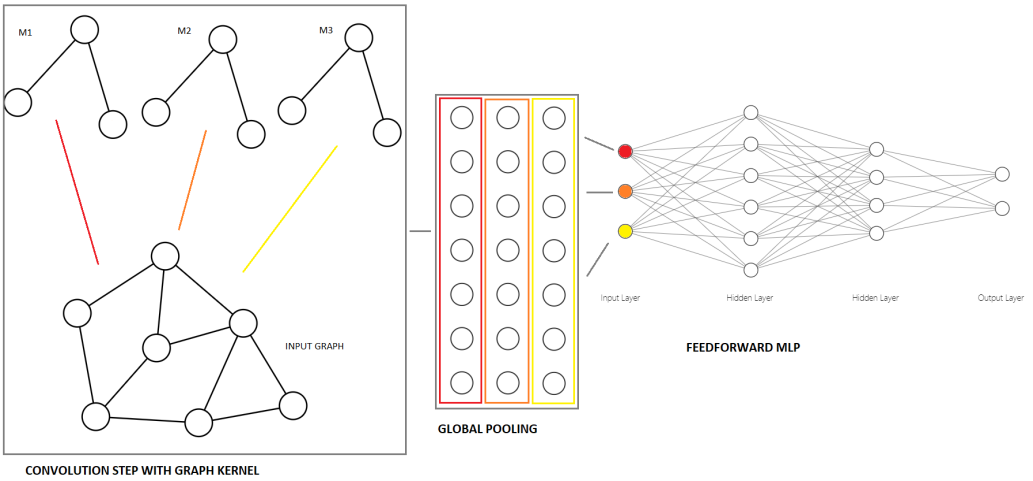

Figure 3.1: Pipeline of the model

Convolution The convolution step consist in computing a graph kernel between the egonet centered at each node and an arbitrary number of structural masks. The egonet around node n is the sub-graph of G restricted to node n and its neighbors. The neighborhood of a node can be computed online or offline and we decided do it offline in order to keep the model free of time consuming operation.
The neighborhood graphs are built in the following way:
for each graph the set of sub-graphs S is created from the nodes of it, for each node of the staring graph we create a sub-graph s made by the node, which it will be the central node, and its neighborhood deciding the number of hops between the central node and the farthest; all the nodes and edges in range of the number of hops belong to the neighborhood.

In the images 3.2 and 3.3 we have an example of the sub-graphs of the previous graph setting the number of hops equal to one.

The image 3.3 shows each sub-graph shape and the color suggests the belonging central node according to 3.2. For simplicity we show one hop of distance from the neighborhood, but the procedure doesn't change with more hops of distance except for the addition of edges among nodes of neighborhood.
Moreover it is important to keep the distance lower than the distance of the two nodes further away, otherwise we obtain that each sub-graph is equal to the starting graph.
Figure 3.1 show that convolution is made for each mask. For each mask we

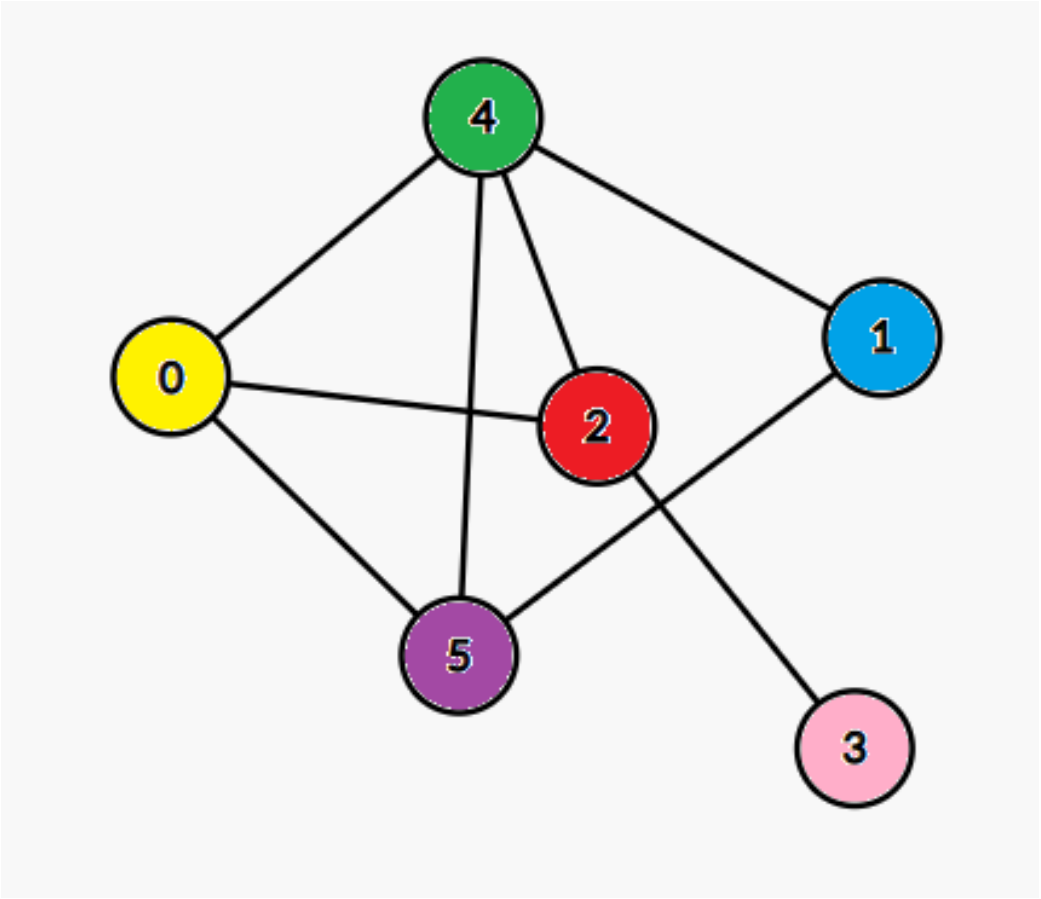

Figure 3.2: Image of the starting graph

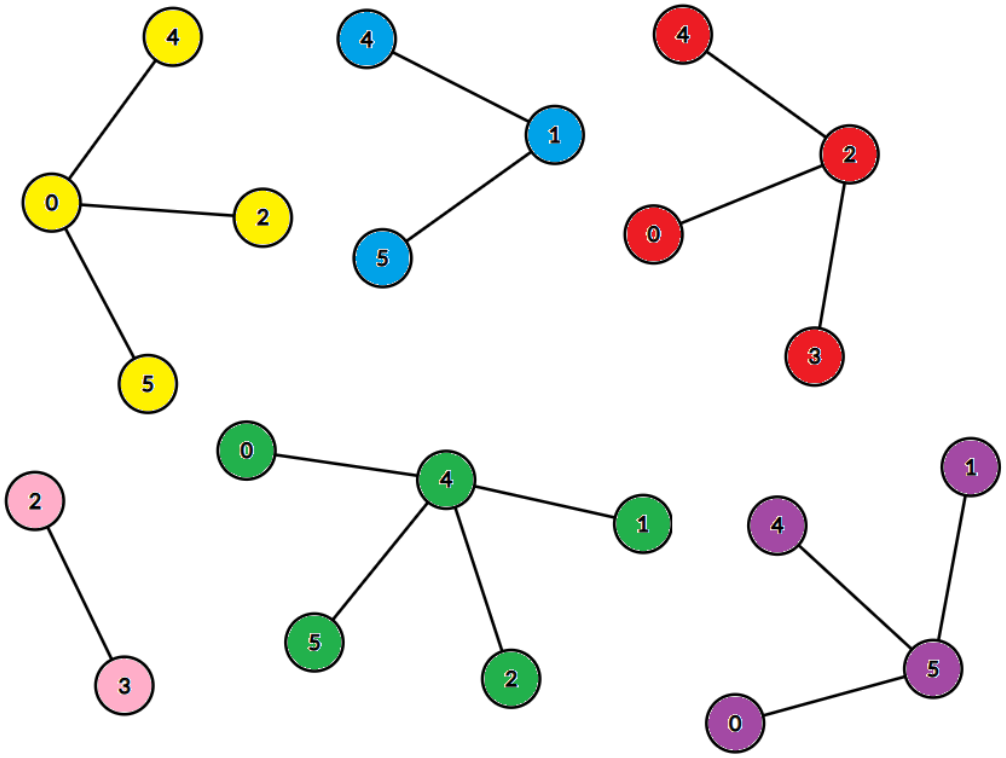

Figure 3.3: Image of the resulting sub-graphs

obtain a features vector which is the convolution result between the egonets and the mask, this vectors are the inputs for the pooling layer

We define the convolution step in the following way:

$$X = f(S, M)$$

where M is the set of the convolution masks, S the set of egonets and $f()$ is a graph kernel function that compute the kernel between each mask against each egonet. For our model we adopted the Weisfeiler-Lehman, which is a similarity measure for isomorphism.

Our model performs a convolution using graph kernel between S and M, the graph kernel is computed for each sub-graph $s \in S$ against each $m \in M$. X is the matrix containing all the kernel values with shape $|S|\text{x}|M|$ where the j^{th} row have the value of each kernel with the j^{th} mask m
Once graph kernels are computed the model performs a pooling operation in order to extract the most meaningful features, and those features will be used as input for a classic Multy Layer Perceptron for the final classification.

Global Pooling The second step consist in the pooling of the extracted features. The pooling is made globally for every convolution mask m and can be done in different ways.

- Max Pooling is a global pooling where for each mask we take only the greater values, in this way is like considering if the feature is present in the starting graph;
- Sum Pooling is a global pooling where for each row of X we sum the whole row, and in this case we have a feature that affects the whole graph;
- Average Pooling is a global pooling which consist on computing the average of each row of X. This case behave similar to the sum pooling, since it affects the whole starting graph but it doesn't affect the dimension of it.

In all these pooling step the result is a feature vector with the same cardinality of M. In our model we adopted the sum pooling because the Max Pooling does not fit our problem since we don't know which feature is present or not, and we want to keep the information of graphs dimension.

Figure 3.1 show the pooling step performed for each mask, each result of the pooling is then used as input for the MLP.

Feed the MLP Once the pooling step is done the resulting features obtained by the global pooling are used as input for a feed forward Multilayer Perceptron (MLP). This MLP is composed by two hidden layer with a Relu function which fires the neurons at every layer; the output layer is defined as one hot encoded. One hot encoding ensures that every neuron of the output represents one of the possible outcome class and has a value from 0 to 1; in this way the output result is the probability that the input belong to the class represented by that neuron.
The network computation of the MLP is defined as follow:

$$\begin{aligned} h_i^{(1)} &= ReLU(\sum_j w_{ij}^{(1)} x_j + b_j^{(1)}) \\ h_i^{(2)} &= ReLU(\sum_j w_{ij}^{(2)} h_j^{(1)} + b_j^{(2)}) \\ y_i &= \phi(\sum_j w_{ij}^{(3)} h_j^{(2)} + b_j^{(3)}) \end{aligned} \tag{3.1}$$

where x_j is the j_{th} neuron of the input layer, h_i^k is the i_{th} neuron of the k^{th} hidden layer, w_{ij}^k is the i_{th} weight associated to the j_{th} neuron of the k^{th} layer, y_i is the i_{th} of the output layer, b is the bias unit and ϕ is the sigmoid activation function. The learning of the MLP is made with a back propagation algorithm using a binary cross entropy function as loss function. The loss function of the MLP is defined as follow:

$$\begin{aligned} Loss(x, y) &= \{l_1, ..., lN\}, \\ l_n &= -(y_n \cdot log x_n + (1 - y_n) \cdot log(1 - x_n)) \end{aligned} \tag{3.2}$$

where x is the predicted label, y is the target label and N is size of the batch.

Learning step The last part of our model is the learning of the convolution masks.
The dominion of the convolution masks is discrete and it's hard to optimize the best masks which are the weights of the convolution layer of our model. For this reason we do not back-propagate the gradient fully to the structural masks, but rather we measure the likelihood of an edit operation increasing the objective function by taking the discrete gradient of the kernel computtd

as the difference between the kernel operation after and before the edit operation, and comparing it against the gradient back-propagated to the input of the MLP.
The computed likelihood is then used in a simulated annealing step to decide whether the edit operation is accepted or rejected.

We compute the direction of the new kernel as the difference between the kernel with a new mask and the kernel with the old masks, we consider the new kernel as correct direction if it has the same sign of the sum of the weights of the MLP. If the weights of the MLP are positive it means that the objective function is increasing, otherwise we need a new kernel that lowers the objective function.

Let $\Delta K^e = \sum_{i \in N}(X_i - X_i^e)$ be the discrete gradient of the sum-pooled kernels after edit operation e, while X_i is the value of the kernels computed against the egonet centered at i and X_i^e is the same computed after the masks where modified with edit operation e.

With this, we can compute the likelihood of objective increase by taking the dot product between the forward-computed discrete gradient ΔK^e and the gradient ∇W backward-propagated to the input of the MLP layer:

$$L^e = \Delta K^e \cdot \nabla W \ . \tag{3.3}$$

where ∇W is the gradient of the MLP. L_j^e determines if the j^{th} mask have to be modified with the new one or not: if the value of L_j^e is positive the new mask is chosen, otherwise the old one is kept.
Keeping the old mask can bring to a local minima and for this reason we adopted a simulated annealing algorithm in order to shake the loss function. A temperature $T = 0$ is set to the new mask adopted, then, when a new mask is rejected, it is rejected with probability $e^{-L_j^e/T}$. Each time the old mask is kept its temperature T is increased with an arbitrary measure. The faster the temperature rise, the less times an old mask is kept.

Chapter 4

Model results and analysis

In this section we explain how the training of the model is performed, how it is validated, and the datasets we used.
In order to evaluate the model results, we train a model for each dataset and we calculate the accuracy of each model with respect to the true response of the dataset. Since all datasets have 2 classes we check the rate between true positive and false positive and between true negative and false negative, in this way we check also if the model is biased in a single class or if it can effectively differentiate the two classes.

For this experiment we used node degree as structure information, either masks nodes and egonets nodes are labeled with outgoing degree.

4.1 Datasets

To evaluate our model, we chose to use three different datasets. Two of these datasets are well known in literature and are the very common used to evaluate many GNNs models.

The dataset we use are:

- Mutagenicity is a chemical compound dataset of drugs, which can be categorized into two classes: mutagen and non-mutagen. Mutagenicity is divided in 2401 graph belonging to class 1 and 1936 belonging to class 2;
- NCI1 is a dataset where each graph is chemical compound, which is relative to anti-cancer screens where the chemicals are assessed as pos-

itive or negative to cell lung cancer. NCI1 is divided in 2057 graph belonging to class 1 and 2053 belonging to class 2;

- PROTEINS is a dataset of proteins that are classified as enzymes or non-enzymes. Proteins is divided in 663 graph belonging to class 1 and 450 belonging to class 2.

4.2 Training Procedure and hyperparameters

In order to train the model, each dataset is divided in three parts:

- 75% of dataset form the training set;
- 12.5% of dataset form the validation set;
- 12.5% of dataset form the validation set.

Before starting the training we initialized the convolution masks and we have created the egonets for each input graph.
The masks are initialized setting an arbitrary number of nodes and random possible edges. In this experiment we report the result obtained using a mask with three nodes and two masks with four nodes. The nodes label of each mask is assigned with the outgoing degree of that node.
The egonets of each graph are initialized as described in the previous chapter. For each node of the input graphs a sub-graph is created with that node as centered node. At each nodes of the egonets is assigned a label initialized with the outgoing degree of that node.
The training step executes a number of epoch between 50 and 100. For each epoch, the model computes firstly the convolution layer, i.e. it computes the graph kernel and the gloobal pooling. Then, the MLP layer is trained with the input of the convolution layer: the MLP computes the gradient and update its weights and it backwards the gradients.
As last step of the training, the model tries to update the weight of the masks used in the convolution layer: it takes an edge for each mask, it adds or removes it (whether the edge already exist or not) and computes new graph kernels. Then, if the difference between the old kernel and the new one has the same sign of the gradients of the MLP, i.e. they have the same direction, it updates the mask with the new weights.
At the end of each training step, the model is evaluated using the validation set. In order to choose the model hyperparameters we adopted the early stopping technique.
To do this, we track the accuracy of the model evaluating the validation set

at each epoch, we keep the hyperparameters model of the epoch where the accuracy is maximized.
At the end of the training step the model is evaluated using the test set in order to understand if the accuracy measure is consistent with the validation.

4.3 Evaluation

We try several different configuration of starting masks, with different number of masks, different number of nodes and/or different edges.
From our tests we can see that the model is sensitive to the initial masks: if the model reaches a local maxima it can stuck into them, this means that the training of the masks does not guarantee the reaching of a global maxima. Another side-effect of an unlucky initialization of the mask is that the model can require more epoch to reach the optimal solution. Thanks to the simulated annealing the model can get out from local maxima, but it still does not guarantee the reaching of the global maxima.

We report the results using one mask with three nodes and two masks with four nodes.
With the plots in the following figure we can see that model converges to a final division of the classes since it reaches a stable accuracy of the validation set.

From plots 4.1 4.2 4.3 is visible that the training of our model on the three datasets converges to a stable state.
We report some histograms with the number of true positives, true negatives, false positives and false negatives, each regarding a different dataset.

The histogram in figure 4.4 shows the number of correct prediction by class for the model trained with dataset Proteins. For class 1 the model predicts correctly 77 graphs and failed 21 times, while for class 2 the correct prediction are 47 against 17 fails.

The histogram in figure 4.5 shows the number of correct prediction by class for the model trained with dataset Mutagenic. For class 1 the model predict correctly 270 graphs and failed 111 times, instead for class 2 the correct prediction are 179 against 90 fails.

The histogram in figure 4.6 shows the number of correct prediction by class for the model trained with dataset Mutagenic. For class 1 the model predict correctly 198 graphs and failed 109 times, instead for class 2 the correct prediction are 193 against 116 fails.

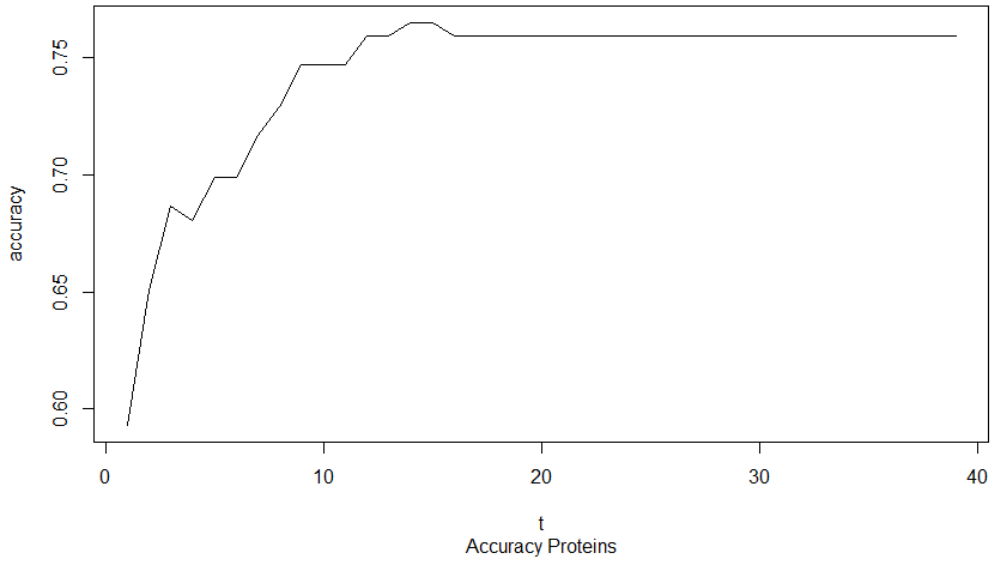

Figure 4.1: Accuracy of the model during the training of dataset PROTEINS.

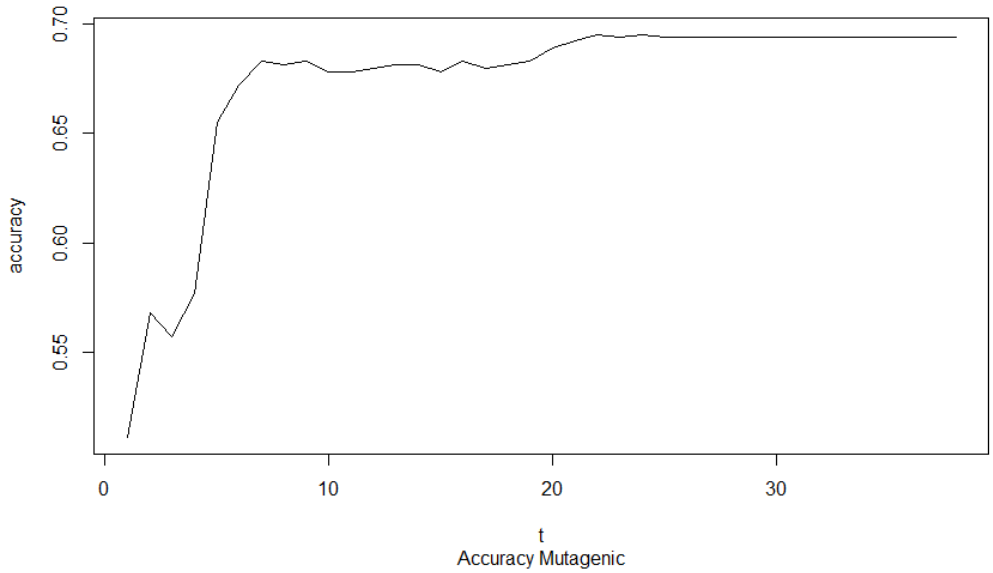

Figure 4.2: Accuracy of the model during the training of dataset Mutagenic.

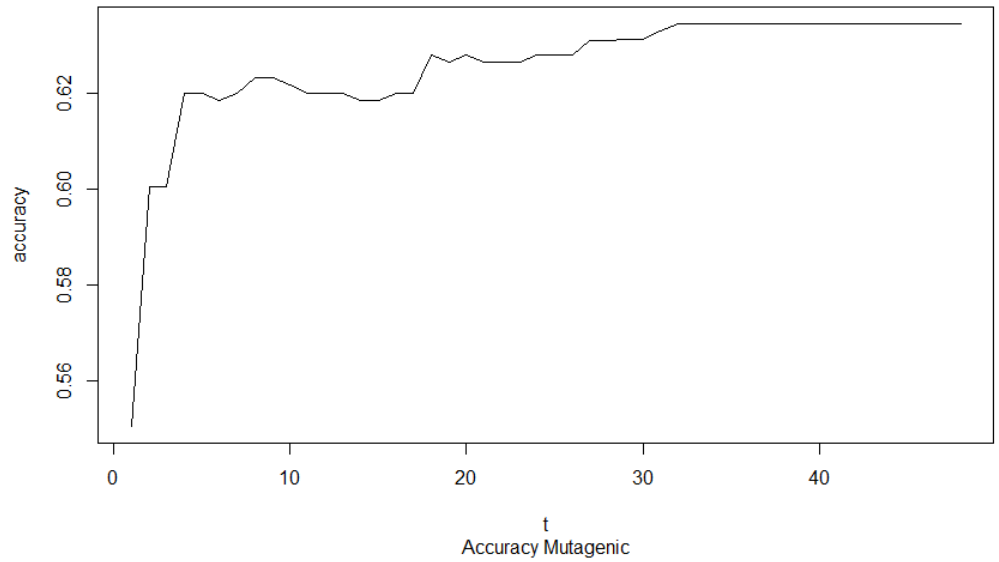

Figure 4.3: Accuracy of the model during the training of dataset Mutagenic.

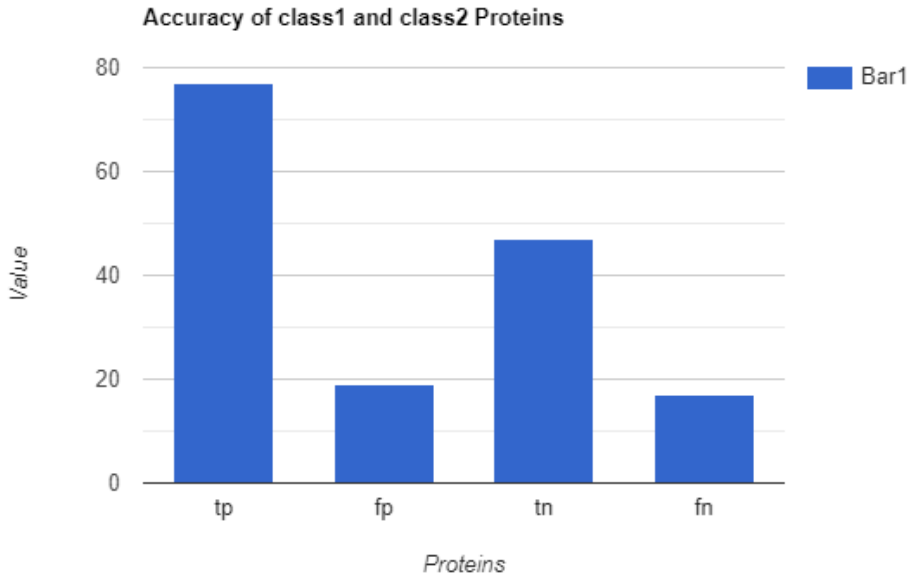

Figure 4.4: histogram of the number of true positive, false positive, true negative, false negative predicted by the model trained with dataset Proteins.

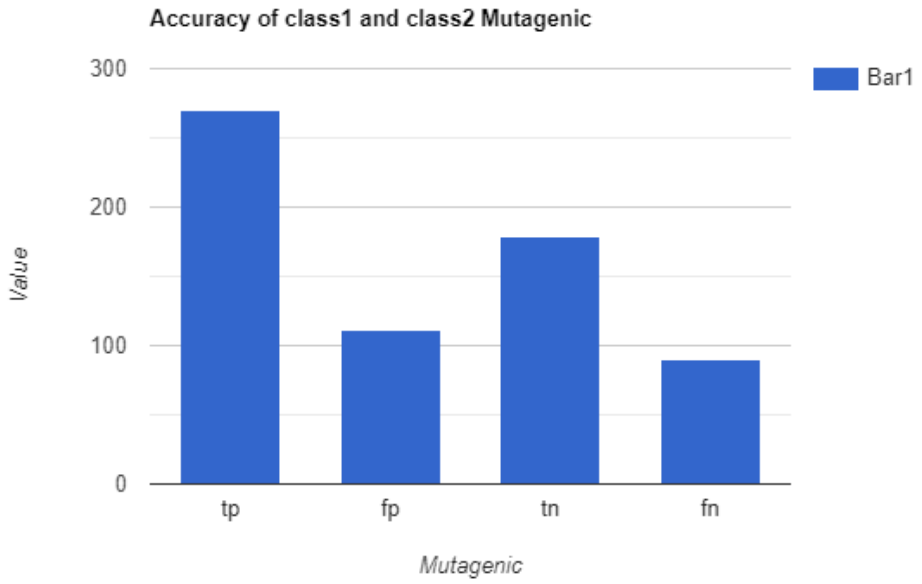

Figure 4.5: histogram of the number of true positive, false positive, true negative, false negative predicted by the model trained with dataset Mutagenic.

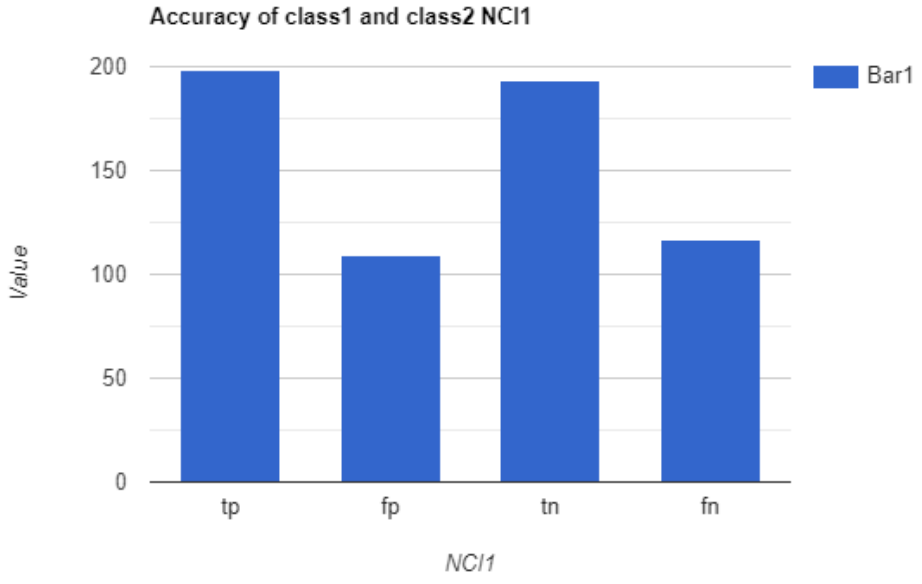

Figure 4.6: histogram of the number of true positive, false positive, true negative, false negative predicted by the model trained with dataset NCI1.

The histograms show that the model is not biased into a single class in all the datasets.
The percentages of accuracy of the single classes are the following:

Dataset	class 1	class 2
Proteins	78%	70%
Mutagenic	71%	67%
NCI1	64%	63%

Table 4.1: Sum up of classes accuracy.

Features structures The masks are the features learned by the model are structures information about the input graphs. We show the masks learned from the previous run for each dataset in tables 4.2, 4.3, 4.4. In each table the first column defines the mask number, the second column defines the set of nodes of the mask and the associated node label, the third columns defines the set of the edges.

Proteins masks		
Mask	**nodes**	**edges**
Mask1	{1: 0, 2: 0, 3: 2}	{(3, 2), (3, 1)}
Mask2	{1: 3, 2: 2, 3: 0, 4: 0}	{(1, 2), (1, 3), (2, 1), (1, 4), (2, 3)}
Mask3	{1: 3, 2: 0, 3: 2, 4: 1}	{(1, 2), (3, 2), (1, 3), (3, 1), (1, 4), (4, 2)}

Table 4.2: Masks structure learned from dataset Proteins.

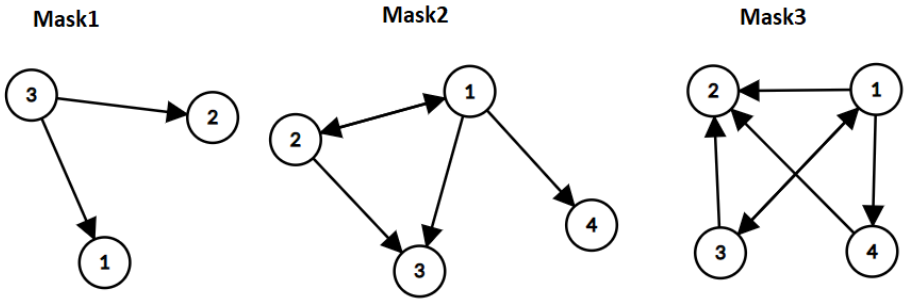

Figure 4.7: Shape of masks learned with Proteins dataset.

Mutagenicity masks		
Mask	**nodes**	**edges**
Mask1	{1: 1, 2: 1, 3: 2}	{(1, 2), (3, 2), (3, 1), (2, 3)}
Mask2	{1: 0, 2: 3, 3: 0, 4: 2}	{(4, 1), (2, 1), (2, 3), (4, 2), (2, 4)}
Mask3	{1: 1, 2: 1, 3: 1, 4: 1}	{(4, 1), (1, 4), (2, 1), (3, 4)}

Table 4.3: Masks structure learned from dataset Mutagenicity.

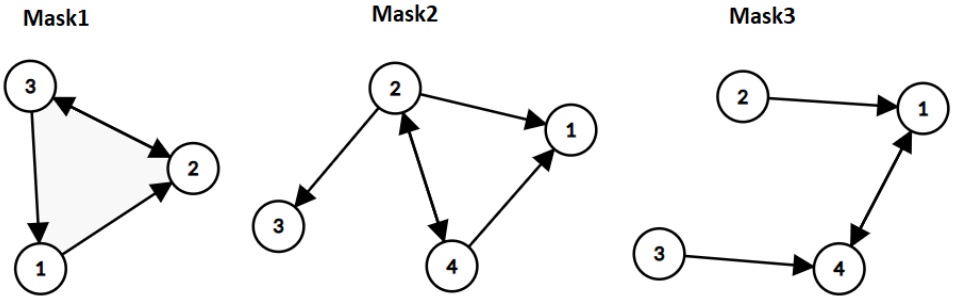

Figure 4.8: Shape of masks learned with Mutagenicity dataset.

NCI1 masks		
Mask	**nodes**	**edges**
Mask1	1: 0, 2: 2, 3: 0	(2, 1), (2, 3)
Mask2	1: 3, 2: 0, 3: 0, 4: 0	(1, 2), (1, 3), (1, 4)
Mask3	1: 0, 2: 0, 3: 1, 4: 2	(3, 2), (4, 3), (4, 2)

Table 4.4: Masks structure learned from dataset NCI1.

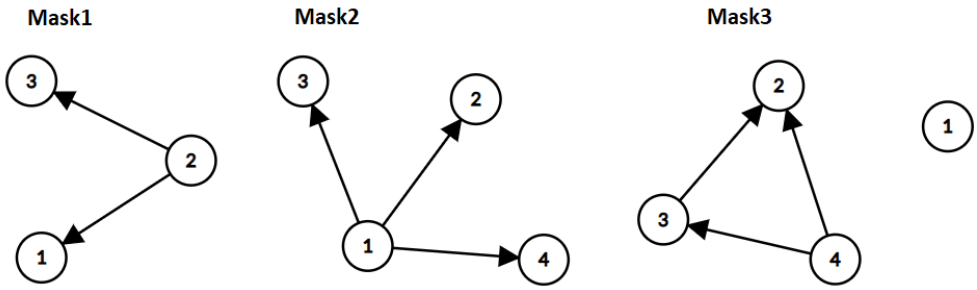

Figure 4.9: Shape of masks learned with NCI1 dataset.

Features Relevance In order to evaluate the relevance of the masks learned from the model we have trained only the MLP with the resulting masks as a further verification.
To do the evaluation of the single mask learned we have trained the MLP removing one mask at time from the convolution step, so we had compute

the accuracy of the model on the same validation and test sets.

We report the final accuracy for each configuration of masks used to feed the MLP.
Table 4.5 describes the accuracy of the MLP model trained using the config-

dataset	**configuration 1**	**configuration 2**	**configuration 3**
Proteins	76%	70%	57%
Mutagenic	61%	62%	64%
NCI1	63%	63%	54%

Table 4.5: Sum up of MLP accuracy trained with the configuration.

urations of two masks learned by our GNN model. Configuration 1 consists of the combination of mask1 and mask2; configuration 2 is the combination of mask1 and mask3 and configuration 3 is the combination of mask2 and mask3.

From tables 4.5 we can see which are the feature extracted that are relevant and which not.

Proteins For dataset Proteins it's clear that the first feature is the one which is more relevant, while the third is completely useless. The configuration mask1 and mask2 reflects the accuracy of the complete model which means that these are the two feature that the model uses to separate the two class.
As an additional evidence we can see that configuration mask1 and mask3 lowers the accuracy of the model to 70% and the configuration of mask2 and mask3 drop the accuracy completely to 57%. Moreover, looking at the distribution of the two class in the validation and test set, we can notice that is exactly the same percentage of the accuracy, 57% for class 1 and 43% for class 2. Looking at the true positive and true negative we can see that with this last configuration the model is biased into a single class.
For Proteins dataset the first feature learned is the most relevant, the second feature learned add some information. The third feature is completely irrelevant.

Mutagenic Looking at the feature learned from Mutagenic dataset we can see that for each configuration of mask the accuracy is lower respect to having all the features together. Configuration of mask1 and mask2 has an

accuracy of 61%, configuration of mask1 and mask3 has an accuracy of 62% and configuration of mask2 and mask3 has an accuracy of 64%. Having all the features masks the model reaches 69∼71% of accuracy. This means that all the masks are relevant.

NCI1 NCI1 dataset behave like Proteins, both configurations of mask1, mask2 and mask1 and mask3 reach 63% of accuracy like having all the three features together, while the configuration of mask2 and mask3 drop the accuracy to 54% accuracy making the model biased into one class. From this table we can say that only the first feature is relevant for this dataset, since is the only one present in all the configuration having the better accuracy.

From tables 4.2, 4.3 and 4.4 we see the structure of the masks. For dataset Proteins and NCI1 the first feature learned is the most relevant. For both dataset this feature has the shape of a three nodes egonet, moreover for Proteins dataset the MLP reach 70% only with that feature. For this reason we can say that the model succeed to learn a feature which is more present on a single class. For Mutagenicity dataset we are not able to interpret this masks structure, the only evidence we have is that these features contain some informations that MLP is able to take advantage of it.
As we said befor features learned from NCI1 have a behavior similar to features of Proteins, but the general information of the first feature is lower respect to Proteins. The second feature learned also has a shape of an egonet, we can say that probably that structure is present in both class equally.
In general we consider the masks that gives no information of the problem as not present or equally present in both classes.

The model has been tested also with more than three mask, but the performance of accuracy decreased as it can be seen in the following table.
In table 4.6 we report the result of a configuration of: one mask of three nodes, 2 masks of four nodes and a mask of five nodes.
In table 4.6 we can see how the performance of the model are diminished.

dataset	**accuracy**
Proteins	0.714
Mutagenic	0.657
NCI1	0.603

Table 4.6: Sum up of MLP accuracy trained with a configuration of four masks.

This is not surprising because is well known that increasing the number of variables in a function increase the number of locality point, in classification problems also means increase the probability of overfitting.
We tried additional experiments trying to increase the number of nodes on the three masks configuration but we wasn't able to gain more information from the features. Moreover increasing the number of nodes at the masks increase the time of computing the possible edge flips since the number of possible edges is quadratic respect to the number of nodes.

4.3.1 Comparison with common GNNs

In this section we compare the accuracy of our model and the best accuracy reached with some classic GNNs we described in chapter 2.
The result are taken from [5] which contains result on dataset PROTEINS and NCI1 evaluated for the following models: DCNN, GCN and GraphSaGe.
From 4.7 we can see that our model performs like GCN model with Proteins

Models	**PROTEINS**	**NCI1**
DCNN	0.732	0.729
GCN	0.766	0.770
GraphSage	0.746	0.732
Our Model	0.765	0.640

Table 4.7: Comparison of accuracy with common GNNs.

dataset, but performs the lower with NCI1 dataset.
For the dataset Mutagenicity we don't have result of common GNNs, so we can't compare the performance of our model with this dataset.

Chapter 5

Conclusion

Graph Neural Network models are used to approach different problems involving graphs and the field of possible GNNs models is very large. In the second chapter we have shown some of these possible models, describing the different techniques adopted by these models. Most of the very common models that we have shown use features embedding to map the features in a different space.
The model we propose belongs to the spatial based convolution. Adopting the convolution requires the knowledge of the problem we are dealing with and the features to extract in order to gain information of the problem.
In the third chapter we have proposed a model that exploits the convolution in order to learn features which we don't know or features that would require a complex analysis of the problem.
We have evaluated the model in chapter four showing the performances in terms of accuracy and the relevance of features learned. The limits of this model is the growing of the number of feature to learn since increasing the number of variables increase the probability to get stuck in a locality point. With less features our model extracts more information from the dataset rather than having a large number. Moreover increasing the number of features does not only decrease the amount of information extracted but also increase the time required in order to compute all the possible new kernels due to the edges combination of the convolution masks.
In the end we can say that the model is able to extract information from different dataset, but the amount of information depends on the structure of the dataset, on the starting mask to learn and on the structure information we want to learn.

5.1 Future Work

To improve our model we have two major solutions:

- The first solution take into account the learning of the label of the mask: in this thesis we use the outgoing degree of the nodes as labels, but it is possible to use the original labels of the dataset. In fact, at the moment we do not take into consideration this information while training our model.

- The second solution adds additional layer to the convolution step. The underlying idea is to make an hierarchical convolution which extract the information at each layer of convolution.

The reason we did not implemented these solutions is that the computation of the gradient both for the first and second solution is complex. Since the weights are discrete the back-propagation of the gradients would be hard to compute.

Bibliography

[1] James Atwood and Don Towsley. Search-convolutional neural networks. *CoRR*, abs/1511.02136, 2015. URL `http://arxiv.org/abs/1511.02136`.

[2] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. 30, 2017. URL `https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf`.

[3] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. URL `http://arxiv.org/abs/1609.02907`.

[4] Thomas N. Kipf and Max Welling. Variational graph auto-encoders, 2016. URL `https://arxiv.org/abs/1611.07308`.

[5] Yao Ma, Suhang Wang, Charu C. Aggarwal, and Jiliang Tang. Graph convolutional networks with eigenpooling. *CoRR*, abs/1904.13107, 2019. URL `http://arxiv.org/abs/1904.13107`.

[6] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009. doi: 10.1109/TNN.2008.2005605.