Text Classification using Graph Convolutional Neural Network

Ayoub Imad
285322@studenti.unimore.it

Abstract

Text classification is a fundamental task in natural language processing (NLP). It involves categorizing text into predefined labels based on its content. Traditional methods, such as bag-of-words, TF-IDF, and word embeddings coupled with machine learning algorithms, have shown significant promise but often struggle with capturing complex semantic relationships and contextual dependencies inherent in text. In recent years, Graph Neural Networks have emerged as a powerful paradigm for handling data with graph structures, offering a novel approach to text classification.

1 Introduction

Text classification is a key task in NLP with wide-ranging applications, including spam detection, and sentiment analysis. An important aspect of text classification is text representation. Historically, this involved using hand-crafted features like bag-of-words and n-grams, which are sparse lexical features. More recently, deep learning techniques such as convolutional neural networks (CNNs) have been adopted to learn text representations. These deep learning models excel at capturing semantic and syntactic information in local consecutive word sequences. However, they often overlook the global word co-occurrences that convey non-consecutive and long-distance semantics within the text. In the following sections, we will delve into the fundamentals of GCNs, discuss the methodology of applying GCNs to text classification, and present experimental results based on the RCV, $manuelDICE$ and $modenaToday$ datasets, comprising crime news articles extracted from Italian newspapers.

2 Graph Neural Network

Graph Neural Networks (GNNs) are a specialized class of neural networks designed to perform inference on data structured as graphs, they leverage the inherent structure of graphs to capture dependencies between nodes and propagate information throughout the network. The propagation relies on a message-passing mechanism. In this process, each node updates its state (or embedding) by aggregating information from its neighboring nodes. This updating process typically involves combining the current state of the node with the aggregated information from its neighbors. Common aggregation functions used in this process include mean, sum, or max pooling.

The architecture is composed of multiple layers, with each layer performing one round of message passing. More formally, message passing layers are permutation-equivariant transformations that map a graph to an updated representation of itself.

Consider a graph $G = (V, E)$, where V represents the set of nodes and E represents the set of edges. Let N_u denote the neighborhood of a node $u \in V$. Additionally, let $\mathbf{x}_u$ be the feature vector of node $u \in V$, and let $\mathbf{e}_{uv}$ be the feature vector of edge $(u, v) \in E$, each layer can be expressed as follows:

$$\mathbf{h}_u = \phi\left(\mathbf{x}_u, \bigoplus_{v \in N_u} \psi\left(\mathbf{x}_u, \mathbf{x}_v, \mathbf{e}_{uv}\right)\right)$$

Here, ϕ and ψ are differentiable functions and $\bigoplus$ is a permutation-invariant aggregation operator that can handle an arbitrary number of inputs, such as an element-wise sum, mean, or max. Specifically, ϕ is known as the update function, and ψ is known as the message function. Essentially, within a computational block, graph nodes update their representations by aggregating messages received from their neighbors. Thus, the outputs of

one or more layers are the node representations $\mathbf{h}_u$ for each node $u \in V$ in the graph.

The specific implementation of the message passing mechanism determines the characteristics and performance of the network. Different definitions of the update function (ϕ), message function (ψ), and aggregation operator ($\bigoplus$) result in different variants of GNNs, each with unique properties and capabilities. Thus, the way message passing is implemented can lead to different types of networks.

3 Applications of Graph Neural Network

Graph Neural Networks (GNN) can be used to solve a variety of graph-related machine learning problems:

- **Node Classification:** Predicting the classes or labels of nodes.
- **Link Prediction:** Predicting if there are potential linkages (edges) between nodes.
- **Graph Classification:** Classifying a graph itself into different categories.
- **Community Detection:** Partitioning nodes into clusters.
- **Anomaly Detection:** Finding outlier nodes in a graph in an unsupervised manner.

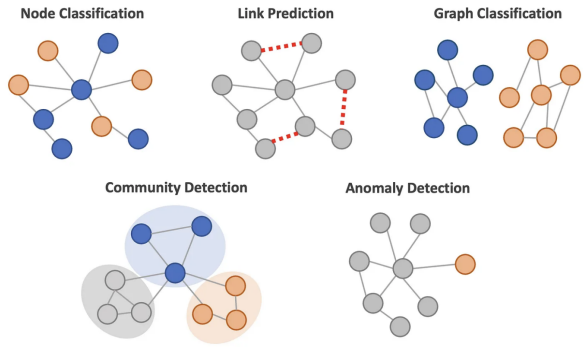

Figure 1: Applications of GNN

4 Graph Convolutional Neural Network

Among the various types of Graph Neural Networks (GNNs), Graph Convolutional Networks (GCNs) are perhaps the most popular and fundamental.

Graph convolution is an efficient method to extract and summarize node information based on graph structures, similar to how convolutional operations in Convolutional Neural Networks (CNNs) are used for image analysis. In images, pixels are organized in a grid, and convolutional filters (weight matrices) slide over the image with a specified stride size. The neighbors of a pixel are determined by the filter size; for example, a 3x3 filter considers the eight surrounding pixels as neighbors, and these weighted pixel values are aggregated into a single value. The output of this convolution process is smaller in size than the input image but provides a higher-level view, beneficial for tasks like image classification.

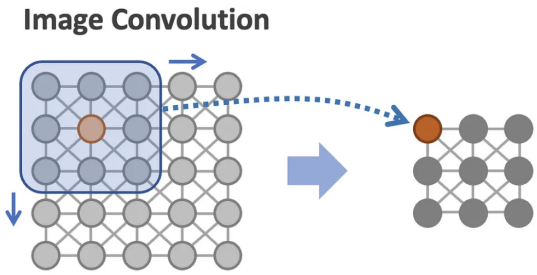

Figure 2: Image Convolution Operation

In contrast, nodes in graphs are arranged in a non-structural manner, and the neighborhood size varies for each node. Graph convolution involves averaging the features of a given node (e.g., the red node in the accompanying illustration) and its neighbors (gray nodes within the blue circle) to update the node's representation. This process enables the node representation to capture localized graph information.

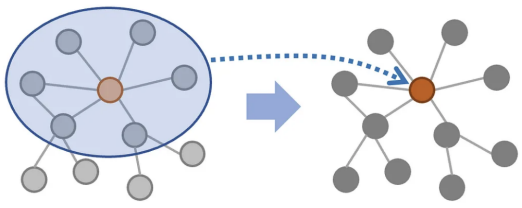

Figure 3: Graph Convolution Operation

From another point of view the operation of graph convolution is illustrated as follows: the central node (orange) collects features from its neighboring nodes (blue). The collected features from the neighboring nodes are averaged together with the feature vector of the central node, combining local information around the central node. Next, the averaged feature vector is then multi-

plied by a weight vector (W), applying a linear transformation to the aggregated features, which helps in learning a new representation for the central node based on its neighborhood. The resulting vector, also yellow, represents the updated node features after the weighted transformation. The final step is to update the central node's feature vector with this newly computed representation.

Figure 4: Graph Convolution Operation

One thing to note in this convolution operation is that the number of graph convolutions determines how many steps away node features are aggregated into each node. In the image below, the first convolution aggregates the blue nodes' features into the orange node and the second convolution can incorporate the green nodes' features into the orange node.

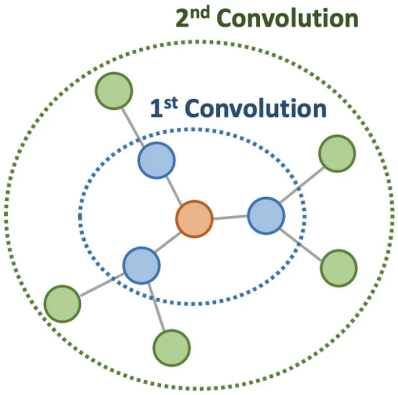

Figure 5: Graph Convolution Operation

Formally, consider a graph $G = (V, E)$ where V (with $|V| = n$) represents the set of nodes and E represents the set of edges. Each node is assumed to have a self-loop, meaning $(v, v) \in E$ for any node v. Let $X \in R^{n \times m}$ be a matrix that includes the features of all n nodes, where m is the dimension of the feature vectors, and each row $\mathbf{x}_v \in R^m$ is the feature vector for node v.

We introduce the adjacency matrix A of G and its degree matrix D, where $D_{ii} = \sum_j A_{ij}$. The diagonal elements of A are set to 1 due to the self-loops. For a one-layer, the new k-dimensional node feature matrix $L^{(1)} \in R^{n \times k}$ is computed as follows:

$$L^{(1)} = \rho(\tilde{A}XW_0)$$

where $\tilde{A} = D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ is the normalized symmetric adjacency matrix, and $W_0 \in R^{m \times k}$ is a weight matrix. Here, ρ is an activation function, such as ReLU, defined as $\rho(x) = \max(0, x)$.

To incorporate information from higher-order neighborhoods, multiple layers can be stacked. For the $(j+1)$-th layer, the node feature matrix $L^{(j+1)}$ is computed as:

$$L^{(j+1)} = \rho(\tilde{A}L^{(j)}W_j)$$

where j denotes the layer number and the initial node features are given by $L^{(0)} = X$.

5 Experiment

This experiment aims to classify documents related to crimes that occurred in the city of Modena using a Graph Convolutional Network and to evaluate the model's performance. To achieve this, three datasets are available: RCV, manualDICE (MD), and ModenaToday (MT), which are described in detail in paragraph 4.

6 Datasets

The datasets are structured as follows:

Datasets		
	# documents	# categories
RCV	21,593	4
MT	5,510	6
MD	1,118	6

To evaluate the results, each dataset is partitioned into three subsets: the training set, the validation set, and the test set. The training set is used for the initial training phase, where the model learns to classify based on provided examples. The validation set is employed to prevent overfitting by providing feedback on model performance during the training process. Finally, the test set is used to assess the model's performance.

7 Methodology

The methodology comprises several stages, beginning with the preprocessing of each document's

text. In this initial phase, the text undergoes several transformations: all words are converted to lowercase, and numerical digits, punctuation, words containing fewer than three letters, and stopwords are removed. Next, for each document i, a co-occurrence graph G_i is constructed where each node represents a word. An edge is established between nodes v_j and v_k if the corresponding words co-occur within a fixed window of 4 elements in the considered document.

To generate the initial word embeddings, represented as the feature matrix X, the Word2Vec model is employed. This model is specifically a pre-trained version that has been trained using an Italian Wikipedia dump with embedding size set to 100. Word vectors effectively capture both syntactic and semantic regularities of natural language. An illustrative example of this capability is shown in the subsequent figure, where the vectorial distance between 'man' and 'woman' is equivalent to the distance between 'king' and 'queen'.

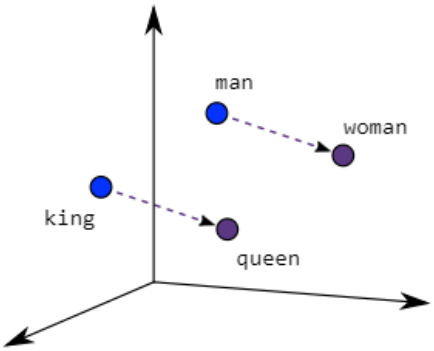

Figure 6: Word2Vec

Following the feature engineering phase, a model is trained using the graph representation to extract document embeddings. These embeddings are then fed into a softmax classifier to determine the document's category.

8 Network architecture

The Graph Convolutional Network model is implemented using the PyTorch framework. The model architecture is defined as follows:

- The `GCN` class extends the `torch.nn.Module` and is initialized with three parameters: `input_dim` (word embeddings dimensionality), `hidden_dim`, and `output_dim` (number of classes).
- The network comprises two graph convolutional layers (`GCNConv`) with the following configurations:
  - The first convolutional layer maps the input dimension to the hidden dimension.
  - The second convolutional layer maps the hidden dimension to itself.
- After each graph convolution, a ReLU activation function is applied to introduce nonlinearity.
- The features are then aggregated across nodes using a `global_max_pool` to form a graph-level representation.
- A dropout layer is applied for regularization.
- A fully connected layer (`Linear`) maps the pooled graph representation from the hidden dimension to the output dimension.

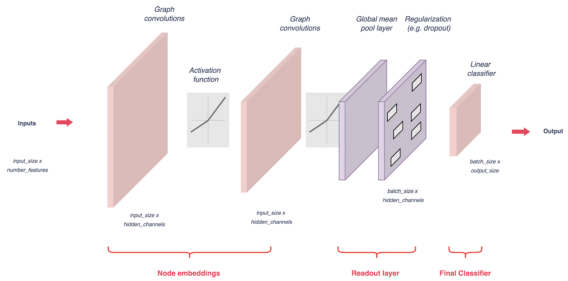

Figure 7: Architecture

9 Training phase

During the training phase, 10% of the training set was randomly selected to serve as the validation set. The network underwent training for a maximum of 200 epochs, utilizing the Adam optimizer and a cross entropy loss function to evaluate performance. Training was stopped if there was no improvement in accuracy on the validation set for 10 consecutive epochs. The learning rate was set at 0.02, and a dropout rate of 0.5 was implemented to mitigate overfitting. The `hidden_dim` was set to 64.

10 Results

The table below presents the accuracy results for the training, validation, and test sets. The algorithm was executed ten times for each dataset.

Results			
	train	validation	test
RCV	89.11%	85.25%	86.76%
MT	91.91%	85.49%	87.75%
MD	85.95%	64.44%	64.29%

The plot below displays the training and validation accuracies during the training phase for the MT dataset.

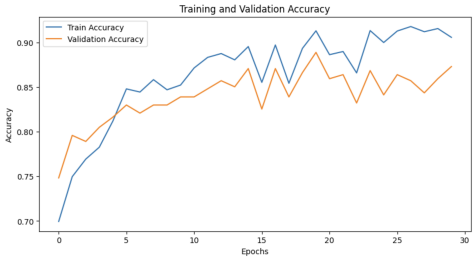

Figure 8: MT train and validation accuracy plot