Generative AI: opdracht 2: Answers to peer questions

Wilfred Van Casteren
nr:837377516
Open Universiteit
Masteropleiding AI

January 16, 2026

Introduction

Initially I wanted to make GraphRAGs the main subject of this report. As I wanted to use LLMs to return Cypher queries from prompts, I decided it would be too complex as I have no LLM running on a separate machine. Even though I really already had started setting up a Neo4j database and did some other readings, especially in Graph Querying. I even started ingesting some basic data into a Graph database.

Eventually I opted for the idea of starting to work with Graph Neural Networks. As this subject is actually very new to me, I was only able to do some initial work. The accompanying Jupyter notebook, contains some code not covered in this report. And it also is a bit more extensive when it comes to getting an idea of the world of Graphs and Graph Neural Networks.

But before I continue, it might be a good thing to introduce the terms used until now.

Retrieval augmented generation (RAG)

RAG is a technique where the system, at query time, first retrieves relevant external documents or data and then passes them to the LLM as additional context. The model uses this context as a source of truth when generating its answer, which typically makes the response more accurate, up to date, and on topic. This technique also allows LLMs to provide answers to questions that they wouldn't have been able to answer otherwise. For example, questions regarding internal company information, email history, and similar private data.

To get an idea of how RAG works and how LLMs in general can be used, I experimented implementing a local RAG, see this link.

GraphRAG

GraphRAGS use knowledge graphs to represent and connect information to capture not only more data points but also their relationships. Thus, graph-based retrievers can provide more accurate and relevant results by uncovering hidden connections that aren't often obvious but are crucial for correlating information

A knowledge graph model is especially suitable for representing structured and unstructured data with connected elements. Unlike traditional databases, they don't require a rigid schema but are more flexible in the data model. The graph model allows efficient storage, management, querying, and processing of the richness of real-world information. In a RAG system, the knowledge graph serves as the flexible memory companion to the language skills of LLMs, such as summarization, translation, and extraction.

Ideas behind graphs

The idea behind Graphs is the immensely powerful "triple": Subject - Predicate - Object. As soon as you start playing the triple game you realize most of the world can be explained in terms of these simple relationships.

- I (subject) 'm reading (predicate) a web page (object). The picture (subject) is on (predicate) the wall (object).
- The picture (subject) is on (predicate) the wall (object).

In AI, triples are the building blocks used to construct knowledge graphs, which provide Large Language Models (LLMs) with a structured and factually reliable "brain". By connecting millions of these triples, AI creates a knowledge graph—a web of relationships that allows it to reason and cross-reference information rather than just predicting the next word in a sentence.

Why to use a graph

- Simplifies Complex Data: Makes large datasets understandable at a glance, saving time compared to reading text or spreadsheets.
- Shows Relationships: Illustrates how different variables connect or influence each other.

- Aids Decision-Making: Helps spot issues (like declining sales) or opportunities (like growth areas) for quicker action.
- Enhances Communication: Offers a compelling visual story for presentations, reports, and general understanding, making data more memorable.
- Provides Context: Adds a visual layer to abstract concepts, making them more concrete and relatable.

Graph Machine Learning

So why graphs

As Graphs were interesting enough as a subject, I did some investigations on the subject of GNNs (Graph Neural Networks).

Use cases for GNNs

Common use cases for Machine Learning are the study of Social Networks, Drug Discovery, Particle Physics, Traffic and Route Optimization, Knowledge Graphs, Financial transaction networks, Chemistry (atoms & molecules), and systems like the universe itself.

- Uber Eats uses GNNs to recommend food to you.
- Pinterest uses PinSage GNN to curate your feed.
- X uses GNNs to detect fake news.
- Icecube labs use GNNs to detect neutrino particles on the South Pole.
- GNNs are used in drug discovery to beat Cancer!

Properties of graphs

Graphs are made up of nodes and edges. There are several different types of graphs:

- Directed/Undirected Graphs.
- Heterogeneous/Homogeneous Graphs have ‘typed’ nodes/edges. Typed means that you can have different types of nodes/edges in a graph. Think types of bonds and atoms in molecules.
- Graph with/without self-edges.
- Graphs with/without loops.
- Multi/Hyper graphs: multigraphs allows multiple edges between the same node, hypergraphs can connect any number of vertices instead of just two
- Weighted/unweighted graphs.

Comparison with text, images

Compared to text or images, graphs are something different. In text there is an explicit order, in images there is a euclidean distance, every pixel is one pixel away from its neighbor. In graphs there is no explicit order, there is no node that comes before another node, every node can be a starting point. It is however possible to express a text or an image as a graph.

Tasks performed using graphs

There are three general types of prediction tasks to be performed on graphs: graph-level, node-level, and edge-level tasks. In this report I will only do some neighboring node-level convolutions.

In a graph-level task, single property for a whole graph is predicted. For a node-level task, some property for each node (of a specific type) in a graph is predicted. For an edge-level task, the property or presence of an edge is predicted in a graph.

These three levels of prediction problems can be solved with a single model class, the GNN.

Graph-level task

In a graph-level task, the goal is to predict the property of an entire graph. For example, for a molecule represented as a graph, one would want to predict what the molecule smells like, or whether it will bind to a receptor implicated in a disease.

Node-level task

A node-level task in graph machine learning aims to predict properties or labels for individual nodes (entities) within a graph, using the node's features and its connections (edges) to other nodes, common examples include classifying users in a social network.

Approach

An approach to classify a node will most of the time include some or more of the following node properties:

- Use the degree of the node: using the degree (number of edges) to classify a node
- Node centrality: Node centrality measures a node's importance or influence in a network, quantified by different metrics like Degree (direct connections), Betweenness (bridge role), and Closeness (average distance to others), helping identify key hubs, brokers, or bottlenecks in systems like social networks or infrastructure.
- Clustering coefficient: A node's clustering coefficient measures how interconnected its neighbors are, indicating local network density, calculated as the ratio of actual edges between a node's neighbors to the total possible edges (forming a clique), ranging from 0 (no neighbors connected) to 1 (neighbors form a clique).
- Graphlets : to generalize the clustering coefficient by counting pre-specified subgraphs (graphlets).

Edge-level task

One example of edge-level inference is in image scene understanding. Beyond identifying objects in an image, deep learning models can be used to predict the relationship between them.

Approach

Some features that could be used when classifying edges are:

- Simple Scalars & Weights: Represent the strength or capacity of a connection, such as trade volume or traffic flow.
- Categorical Labels: Discrete tags identifying relationship types, especially in heterogeneous networks (e.g., "follow," "friend," "child of").
- Multidimensional Vectors: Complex, learnable feature vectors that encode multiple attributes simultaneously
- Structural Signatures: Topologically derived features like the shortest path distance between nodes or neighborhood overlap.
- Directional Encoding: Modern frameworks now often represent edge directions as multidimensional features to handle asymmetric relationships in directed graphs.

An overview of the tasks in Graph ML

An overview of the tasks in Graph ML is to be seen in Figure 1

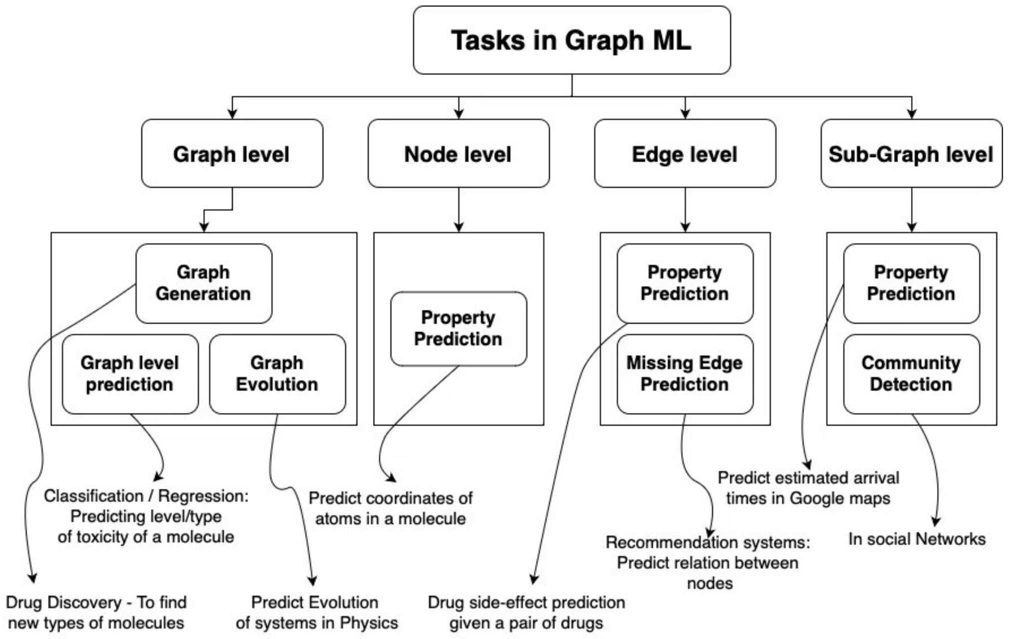

Tasks in Graph ML
Graph level
Node level
Edge level
Sub-Graph level
Graph Generation
Graph level prediction
Graph Evolution
Property Prediction
Property Prediction
Missing Edge Prediction
Property Prediction
Community Detection
Classification / Regression: Predicting level/type of toxicity of a molecule
Predict coordinates of atoms in a molecule
Predict estimated arrival times in Google maps
Recommendation systems: Predict relation between nodes
In social Networks
Drug Discovery - To find new types of molecules
Predict Evolution of systems in Physics
Drug side-effect prediction given a pair of drugs

Figure 1: General overview of tasks to be performed by a GML.

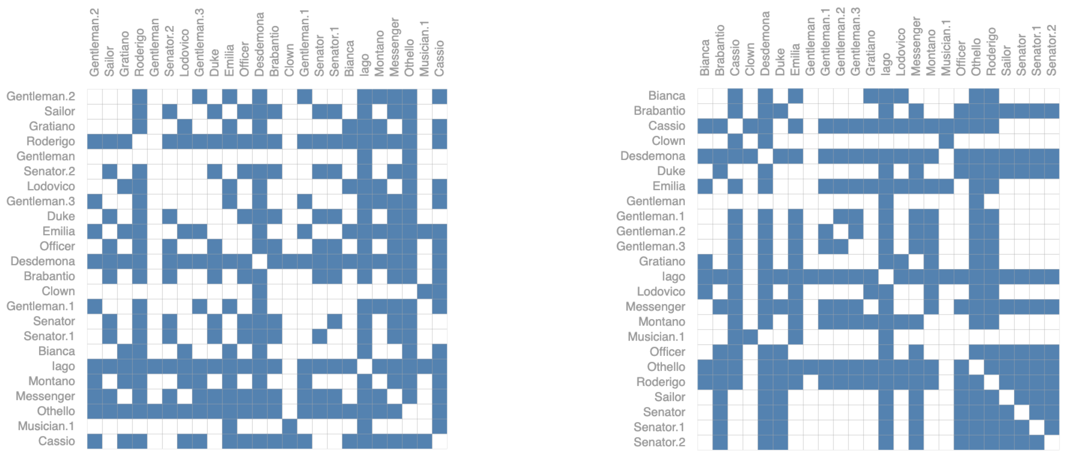
How to do

So, how would one try to solve these different graph tasks with neural networks? The first step is to think about how to represent graphs to be compatible with neural networks.

Representing a graph in a tensor

However, representing a graph's connectivity is complicated. Perhaps the most obvious choice would be to use an adjacency matrix, since this is easily tensorisable. However, this representation has a few drawbacks. the number of nodes in a graph can be on the order of millions, and the number of edges per node can be highly variable. Often, this leads to very sparse adjacency matrices, which are space-inefficient.

Another problem is that there are many adjacency matrices that can encode the same connectivity, and there is no guarantee that these different matrices would produce the same result in a deep neural network (that is to say, they are not permutation invariant).



(a) One visualization of the adjacency matrix representing the othello dataset.

(b) Another visualization of the adjacency matrix representing the othello dataset.

Figure 2: Two visualizations of the same adjacency matrix representing the othello dataset.

Adjacency list One elegant and memory-efficient way of representing sparse matrices is as adjacency lists. These describe the connectivity of edges between nodes as a tuple. And this is how my experiments will be set up.

Word2Vec and Node2Vec

Before continuing into the world of GNNs, I need to say something about Word2Vec and Node2Vec. These are ways of expressing words and nodes as vectors.

One should understand Word2Vec before trying to understand Node2Vec. That's why I will start this section with Word2Vec.

Word2Vec

Embedding words is a technique where individual words are transformed into a numerical representation of the word (a vector). Where each word is mapped to one vector, this vector is learned in a way which resembles a neural network. The vectors try to capture various characteristics of that word with regard to the overall text. These characteristics can include the semantic relationship of the word, definitions, context, etc. With these numerical representations, you can do many things, like identify similarities or dissimilarities between words. The simplest embedding would be a one-hot encoding of text data where each vector would be mapped to a word.

Idea

The effectiveness of Word2Vec comes from its ability to group together vectors of similar words. Given a large enough dataset, Word2Vec can make strong estimates about a word's meaning based on its occurrences in the text. These estimates yield word associations with other words in the corpus. For example, words like King and Queen would be very similar to one another. When conducting algebraic operations on word embedding, you can find a close approximation of word similarities. For example, the 2-dimensional embedding vector of king – the 2-dimensional embedding vector of man + the 2-dimensional embedding vector of woman yielded a vector which is very close to the embedding vector of queen.

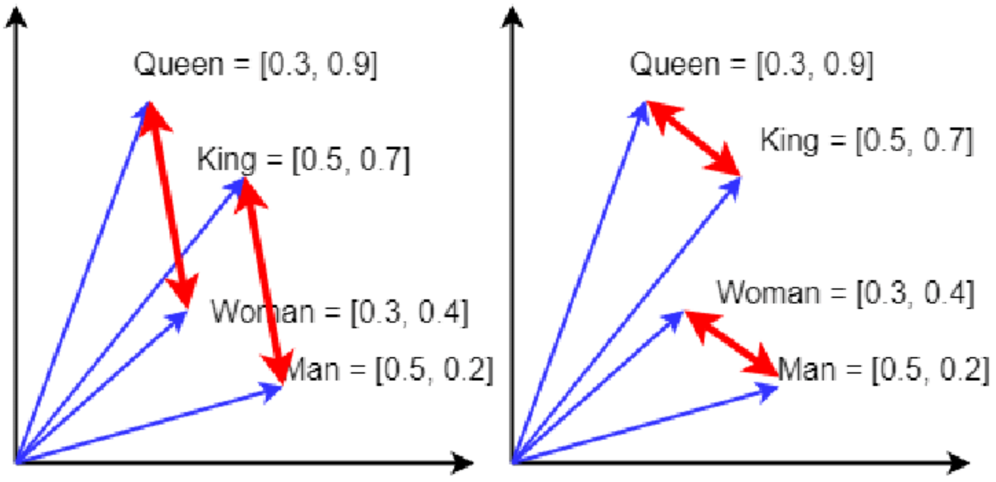


Figure 3: 'King' versus 'queen' word embedding, the numerical difference between man and woman is comparable to the difference between queen and king.

Skipgram model

Node2Vec is inspired by the skip-gram model, a way to learn a contextual embedding introduced when Word2Vec was introduced.. The Skip-gram model predicts the context for a given word. The context is defined as the adjacent words to the input term. The skip-gram model is a simple neural network with one hidden layer trained to predict the probability of a given word being present when an input word is present.

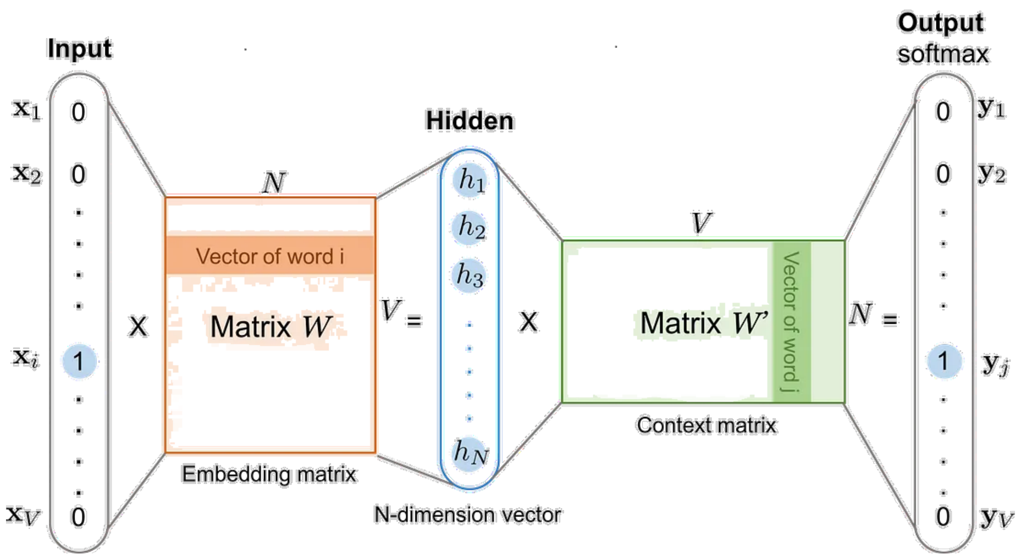


Figure 4: Skipgram model, the hidden layer presents the embedding vector for a chosen input word1.

The window size parameter,the number of context words considered per target word when training, has a significant influence on the results of the word embedding. Larger context window size tends to capture more topic/domain information. Smaller windows tend to capture more information about the word itself, e.g., what other words are functionally similar.

Node2Vec

The idea behind the Node2Vec model is just like in Word2Vec, using Node sequences (sentences) to train a deep learning (DL) model. The Node sequences are generated by creating random walks through a graph. These sentences are then used to train the model. Sometimes the walks are generated by giving each connected node an equal chance, other times chances are unequally distributed.

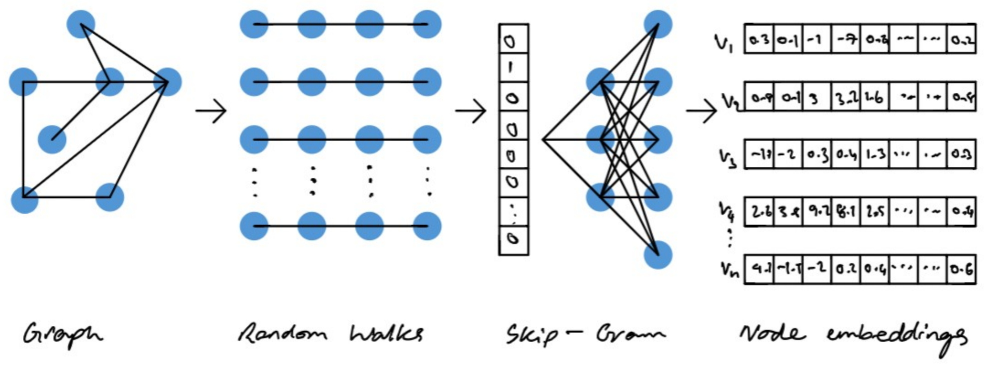


Figure 5: A very general overview of the Node2Vec methodology.

GNNs (Graph Neural Networks)

A GNN is an optimizable transformation on all attributes of the graph (nodes, edges, global-context) that preserves graph symmetries (permutation invariances). GNNs adopt a “graph-in, graph-out” architecture meaning that these model types accept a graph as input, with information loaded into its nodes, edges and global-context, and progressively transform these embeddings, without changing the connectivity of the input graph. There are multiple types of GNNs of which I will only mention the GCN (Graph Convolutional Network).

GCN

The majority of GNNs are Graph Convolutional Networks. The convolution in GCN is comparable to a convolution in neural networks. It multiplies neurons with weights (filters) to learn from data features. Just like a CNN learns about a pixel by convolving the pixel's neighbors, a GCN learns a node's features from neighboring nodes. The major difference between GCN and CNN is that a GCN is developed to work on non-euclidean data structures where the order of nodes and edges can vary.

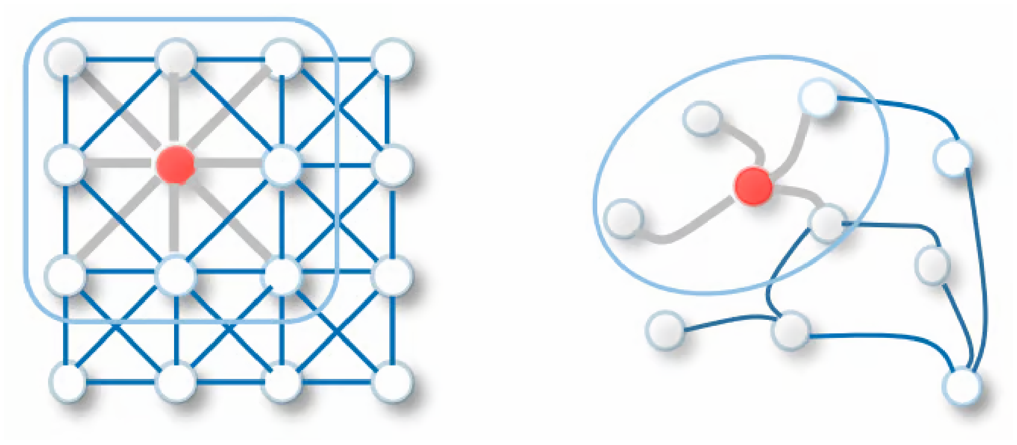


Figure 6: Convolving in a cnn (left) versus 'convolving' in a gcn (right).

Updating the embedding for a target node

GNNs work by aggregating information from a node's neighbors (and the neighbors' neighbors, and so on) to compute a representation (embedding) of each node or edge. The embedding captures both the features of the nodes and the structure of the graph.

Given a Graph G(V,E), the goal is to map each node v to its own d-dimensional embedding, a representation, that captures all the node's local graph structure and data (node features, edge features connecting to the node, features of nodes connecting to our node v proportional to importance of each neighborhood node) such that nodes similar are closer together in the embedding space.

In figure Figure 7 node A's vector embedding is learned by learning it's neighbors' embedding. As can be seen every neighbor's embedding gets updated by a different number of neighbors, this is why most of the time a node's embedding update is normalized by the number of neighbors. Otherwise, nodes with more neighbors get much higher values for their newly learned embedding.

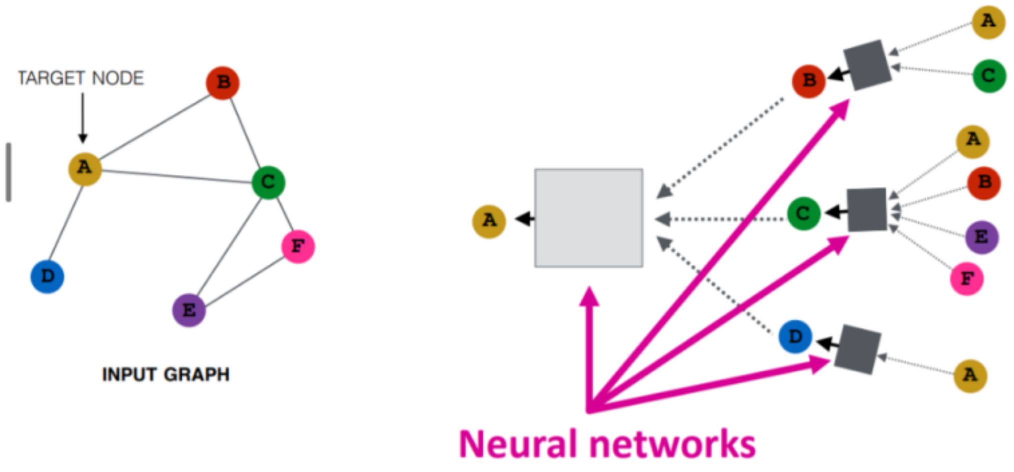


Figure 7: Overview of the way node A's embedding gets updated by its direct and further neighbors.

Implementing a Custom GNN Layer

To get an idea of how convolutions work I added two custom implementations that perform some convolutional work. I still did not get a complete insight into their workings, but getting closer.

By subclassing torch.nn.Module

Libraries like PyTorch Geometric provide powerful, pre-built layers for constructing Graph Neural Networks (GNNs). understanding how to build a GNN layer from scratch using core PyTorch operations offers a greater understanding and the flexibility to implement novel or customized message-passing schemes.

The fundamental idea behind many GNN layers is message passing, where nodes iteratively aggregate information from their neighbors and update their own representations. We can break this down into two main steps for each node:

- Aggregation: Collect features or ”messages” from neighboring nodes
- Update: Combine the aggregated information with the node’s current feature vector to produce an updated feature vector

For this objective a basic layer that performs the Aggregate and Update steps. It will transform node features using a learnable weight matrix, aggregates the transformed features from neighbors using a simple sum, and then applies an activation function.

Mathematically, for a node i, this operation can be described as:

$$a_i = \sum_{j \in \mathcal{N}(\text{i}) \cup \{i\}} W h_j$$

$$h_i' = \sigma(a_i)$$

Here, h_j represents the feature vector of node j, W is a learnable weight matrix, $\mathcal{N}(i)$ is the set of neighbors of node i and σ is a non-linear activation function (like ReLU). Note that the node itself is included (i) in the aggregation, often referred to as adding a self-loop. This ensures the node’s original features are considered in the update.

The code

The code in Listing 1 is a way to implement a message passing algorithm using a basic Linear layer. For a node to not only aggregate over its neighbors feature vector, self_loops are added (line 35-38). In line 44 the complete graph’s nodes features matrix is transformed to better capture the types it represents. On line 55 the transformed matrix is being added to the empty feature vector, for those node indices present in the target edge list or col. Finally the aggregated features are passed through a ReLu.

Listing 1: Message Passing using nn.Module

```
1  import torch
2  import torch.nn as nn
3  import torch.nn.functional as F
4
5  class SimpleGNNLayer(nn.Module):
6      """
7      A basic Graph Neural Network layer implementing message passing.
8      Args:
9      in_features (int): Size of each input node feature vector.
10     out_features (int): Size of each output node feature vector.
11     """
12     def __init__(self, in_features, out_features):
13         super(SimpleGNNLayer, self).__init__()
14         self.in_features = in_features
15         self.out_features = out_features
16         # Define the learnable weight matrix
17         self.linear = nn.Linear(in_features, out_features, bias=False)
18         # Initialize weights (optional but often good practice)
19         #nn.init.xavier_uniform_(self.linear.weight)
20
21     def forward(self, x, edge_index):
22         """
23         Defines the computation performed at every call.
24         Args:
```

```
25   x (torch.Tensor): Node features tensor with shape [num_nodes, in_features].
26   edge_index (torch.Tensor): Graph connectivity in COO format with shape [2, num_edges].
27   edge_index[0] = source nodes, edge_index[1] = target nodes.
28   Returns:
29   torch.Tensor: Updated node features tensor with shape [num_nodes, out_features].
30   """
31   num_nodes = x.size(0)
32
33   # 1. Add self-loops to the adjacency matrix represented by edge_index
34   # Create tensor of node indices [0, 1, ..., num_nodes-1]
35   self_loops = torch.arange(0, num_nodes, device=x.device).unsqueeze(0)
36   self_loops = self_loops.repeat(2, 1) # Shape [2, num_nodes]
37   # Concatenate original edges with self-loops
38   edge_index_with_self_loops = torch.cat([edge_index, self_loops], dim=1)
39
40   # Extract source and target node indices
41   row, col = edge_index_with_self_loops
42
43   # 2. Linearly transform node features
44   x_transformed = self.linear(x) # Shape: [num_nodes, out_features]
45
46   # 3. Aggregate features from neighbors (including self)
47   # We want to sum features of source nodes (row) for each target node (col)
48   # Initialize output tensor with zeros
49   aggregated_features = torch.zeros(num_nodes, self.out_features, device=x.device)
50
51   # Use index_add_ for efficient aggregation (scatter sum)
52   # Adds elements from x_transformed[row] into aggregated_features
53   # at indices   specified by col
54   # index_add_(dimension, index_tensor, tensor_to_add)
55   aggregated_features.index_add_(0, col, x_transformed[row])
56
57   # 4. Apply final activation function (optional)
58   # For this example, let's use ReLU
59   output_features = F.relu(aggregated_features)
60
61   return output_features
62
63  def __repr__(self):
64   return f'{self.__class__.__name__}({self.in_features}, {self.out_features})'
```

Scatter add Line 55 probably needs some extra explanation. Figure 8 is a simplified depiction as the vector used here is a scalar. It shows how for instance a node having 3 neighbors (the node's index is 0) gets its embedding updated by adding it's 3 neighbors' node embedding. The target index value origins from the edge_index tuple list. In this case one of the values (5,1,2) is the current embedding of the target node itself. Normalization is not included in this depiction.

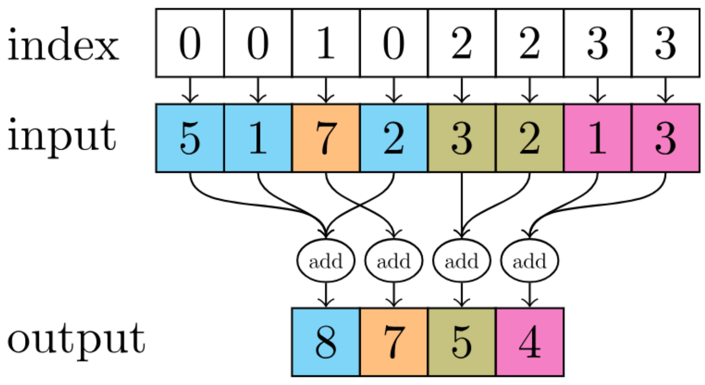


Figure 8: Scatter add or index add visualized.

More information Follow this link for a extra explanation.

By subclassing conv.MessagePassing

PyG provides the MessagePassing base class, which helps in creating such kinds of message passing graph neural networks by automatically taking care of message propagation. The user only has to define the functions, message(), and update(), as well as the aggregation scheme to use, aggr="add", aggr="mean" or aggr="max".

The code in Listing 2 is a way to implement a message passing algorithm using a basic Linear layer by subclassing conv.MessagePassing. For a node to not only aggregate over its neighbors feature vector, self_loops are added (line 23). Line 29-33 calculates the normalization factor. The messages are propagated on line 36 into a variable representing the recalculated feature vector.

Listing 2: MessagePassing using conv.MessagePassing

```python
 1 import torch
 2 from torch.nn import Linear, Parameter
 3 from torch_geometric.nn import MessagePassing
 4 from torch_geometric.utils import add_self_loops, degree
 5 
 6 class GCNConv_custom(MessagePassing):
 7  def __init__(self, in_channels, out_channels):
 8   super().__init__(aggr='add')
 9   self.lin = Linear(in_channels, out_channels, bias=False)
10   self.bias = Parameter(torch.empty(out_channels))
11 
12   self.reset_parameters()
13 
14  def reset_parameters(self):
15   self.lin.reset_parameters()
16   self.bias.data.zero_()
17 
18  def forward(self, x, edge_index):
19   # x has shape [N, in_channels]
20   # edge_index has shape [2, E]
21 
22   # Step 1: Add self-loops to the adjacency matrix.
23   edge_index, _ = add_self_loops(edge_index, num_nodes=x.size(0))
24 
25   # Step 2: Linearly transform node feature matrix.
26   x = self.lin(x)
27 
28   # Step 3: Compute normalization.
29   row, col = edge_index
30   deg = degree(col, x.size(0), dtype=x.dtype)
31   deg_inv_sqrt = deg.pow(-0.5)
32   deg_inv_sqrt[deg_inv_sqrt == float('inf')] = 0
33   norm = deg_inv_sqrt[row] * deg_inv_sqrt[col]
34 
35   # Step 4-5: Start propagating messages.
36   out = self.propagate(edge_index, x=x, norm=norm)
37 
38   # Step 6: Apply a final bias vector.
39   out = out + self.bias
40 
41   return out
42 
43  def message(self, x_j, norm):
44   # x_j has shape [E, out_channels]
45 
46   # Step 4: Normalize node features.
47   return norm.view(-1, 1) * x_j
```

More information Follow this link for a more complete explanation.

Using and testing the layers

The Two classes defined above and a the GCNConv layer will be tested on the Karateclub dataset. This is a small dataset with only 34 nodes and 156 edges. I will set up 3 models, each model will use a different of convolving mechanism. The initial feature vector for each node has 34 elements, every node has a unique vector from a identity matrix.

The graph

It is good practice to inspect the graph or part of the graph. In this case the dataset is limited enough for visualization. The colors represent the types of the nodes.

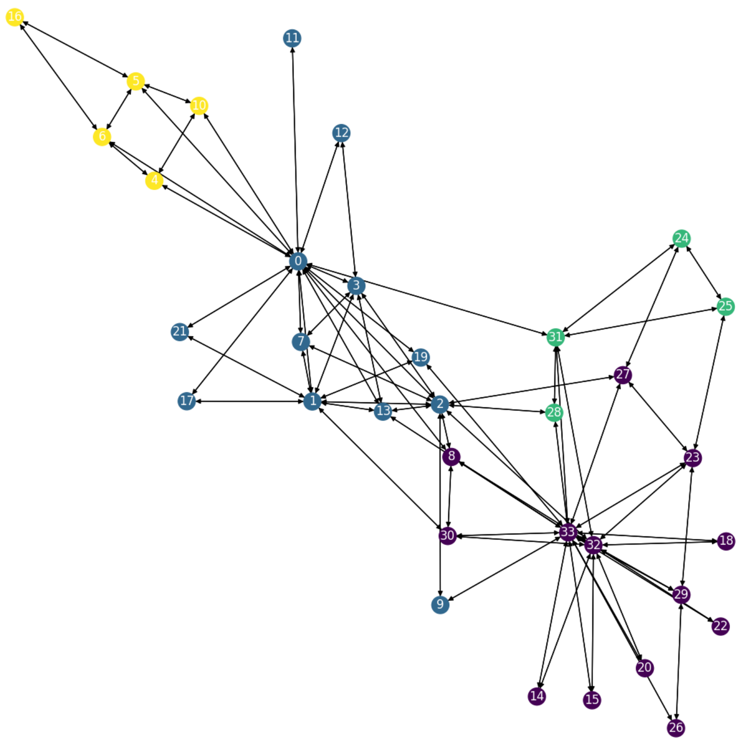


Figure 9: The karateclub dataset as a grah.

Purpose of the models

The purpose of the models will be to adapt the feature vectors. The loss will be the driving factor in this adaptation. There are 4 types of nodes. While training, there is one known representant for every node type. The loss used will be cross entropy loss.

The models

To get an idea of the models. They are really simple models set up in the same way.

The model using predefined PyG GCNConv as a convolution layer.

Listing 3: A model using predefined PyG convolution

```
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv

class GCN(torch.nn.Module):
 def __init__(self):
  super().__init__()
  torch.manual_seed(1234)
  self.conv1 = GCNConv(dataset.num_features, 4)
  self.conv2 = GCNConv(4, 4)
  self.conv3 = GCNConv(4, 2)
  self.classifier = Linear(2, dataset.num_classes)
 def forward(self, x, edge_index):
  h = self.conv1(x, edge_index)
  h = h.tanh()
  h = self.conv2(h, edge_index)
  h = h.tanh()
  h = self.conv3(h, edge_index)
  h = h.tanh()
  out = self.classifier(h)
  return out, h
```

GCNConv layers This message-passing and aggregation mechanism enables GCN to capture both local and global information within graph structures, making it effective for a wide range of graph-related tasks such as node classification, graph classification, and link prediction.

In practice, multiple layers of GCN are often stacked to capture information from nodes at further distances. The output of each layer serves as the input for the next, allowing the model to progressively extract higher-level feature representations. For instance, a two-layer GCN enables each node's representation to include information from its two-hop neighbors, while a three-layer GCN can incorporate information from three-hop neighbors, and so on.

The model using the custom SimpleGNNLayer as a convolution layer.

This model will behave differently, as there is no normalization.

Listing 4: A model using custom convolutions

```
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv

class GCN_custom(torch.nn.Module):
 def __init__(self):
  super().__init__()
  torch.manual_seed(1234)
  self.conv1 = SimpleGNNLayer(dataset.num_features, 4)
  self.conv2 = SimpleGNNLayer(4, 4)
  self.conv3 = SimpleGNNLayer(4, 2)
  self.classifier = Linear(2, dataset.num_classes)
 def forward(self, x, edge_index):
  h = self.conv1(x, edge_index)
  h = h.tanh()
  h = self.conv2(h, edge_index)
  h = h.tanh()
  h = self.conv3(h, edge_index)
  h = h.tanh()
  out = self.classifier(h)
  return out, h
```

The model using the custom GCNConv convolution layers.

This layer should perform the same as the PyG GCNConv layer, it does the same as the original.

Listing 5: A model using custom convolutions

```
import torch
from torch.nn import Linear
from torch_geometric.nn import GCNConv

class GCN_custom1(torch.nn.Module):
```

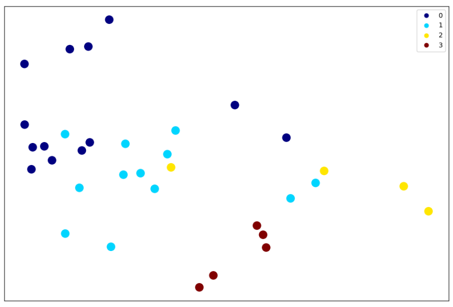
```
def __init__(self):
 super().__init__()
 torch.manual_seed(1234)
 self.conv1 = GCNConv_custom(dataset.num_features, 4)
 self.conv2 = GCNConv_custom(4, 4)
 self.conv3 = GCNConv_custom(4, 2)
 self.classifier = Linear(2, dataset.num_classes)

def forward(self, x, edge_index):
 h = self.conv1(x, edge_index)
 h = h.tanh()
 h = self.conv2(h, edge_index)
 h = h.tanh()
 h = self.conv3(h, edge_index)
 h = h.tanh()
 out = self.classifier(h)
 return out, h
```

Results of the experiment

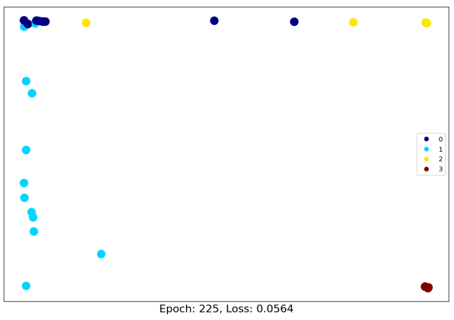
The experiment wasn't really set up as an experiment, the models didn't get initialized in the same way to mention only one flaw. I just wanted to use the different layers and see whether they did do something well. And I guess they did.

The model using predefined PyG GCNConv as a convolution layer.

The nodes are quiet separable after training.



(a) The 'untrained' embedding.

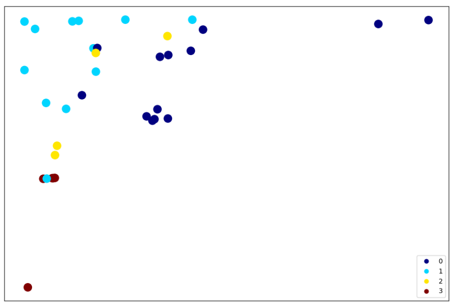


(b) The 'trained' embedding.

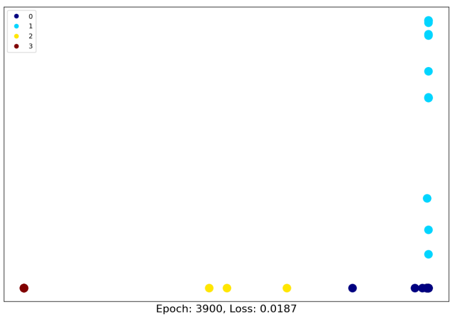
Figure 10: Visualization of the embedding after using the PyG GCNConv convolutions model.

The model using custom SimpleGNNLayer as a convolution layer.

It took a lot of epochs before the embedding became separable.



(a) The 'untrained' embedding.

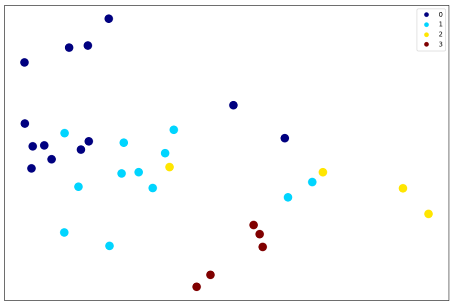


(b) The 'trained' embedding.

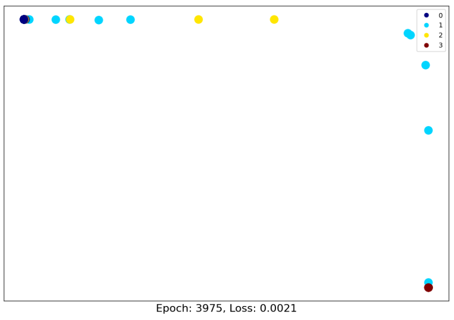
Figure 11: Visualization of the embedding after using the SimpleGNNLayer convolutions model.

The model using custom GCNLayer as a convolution layer.

It took a lot of epochs before and even then the embedding wasn't separable.



(a) The 'untrained' embedding.



(b) The 'trained' embedding.

Figure 12: Visualization of the embedding after using the custom GCNConv convolutions model.

Node prediction

In this section another dataset will be used. The dataset that will be used is the Cora dataset, a citation network. It contains 2708 publications, the nodes. The edges between nodes represent a citation relation, in one way or another. There are 10556 edges. A node has 1433 features, every feature element is a publication's word count for one of 1433 unique words like 'RNN', 'ReLu'. To get an idea of the composition of the nodes see Figure 13

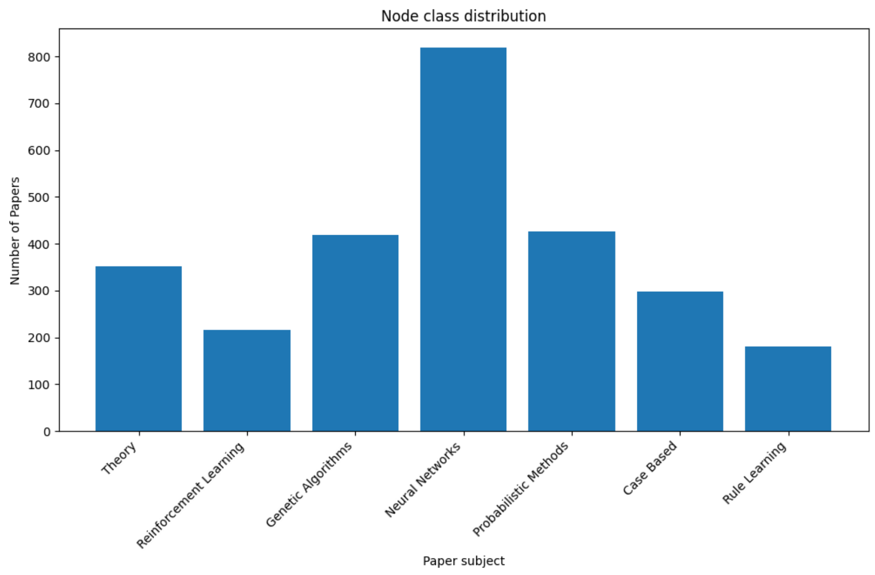


Figure 13: The node type counts.

The graph is too big to show completely, so I will show only part (750 nodes) of it, see Figure 14. This implies some nodes will have no citations as, the counterparts of these relations will not be part of the plot.

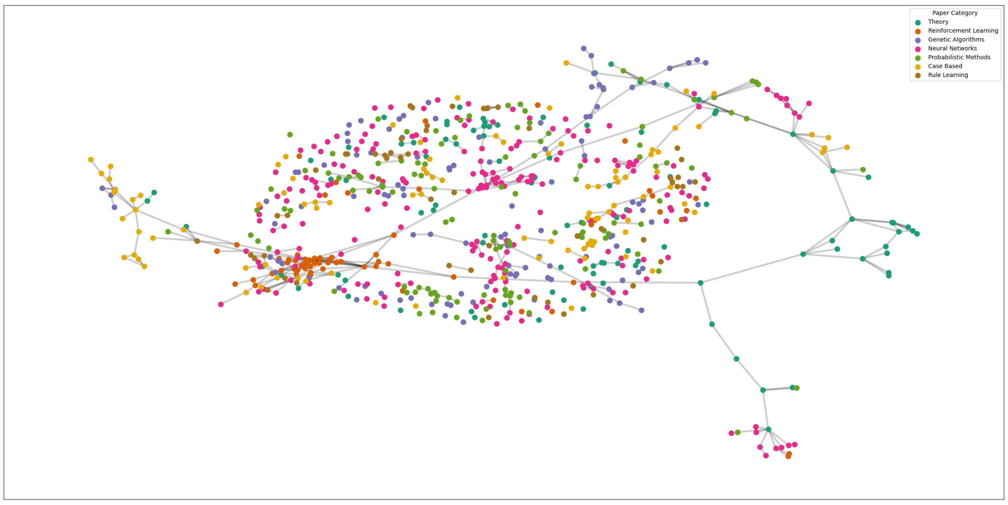


Figure 14: The node type counts.

Predicting the node type

I will try to train a model to predict the node type, the node type is defined by its main subject.

The model

The model contains the same PyG type of convolutions to find some kind of pattern in which node cite which kind of nodes. The result is a 75% prediction accuracy. T

Listing 6: A model for node type prediction

```
class GCN(torch.nn.Module):
 def __init__(self, in_channels, hidden_channels, out_channels, dropout=0.5):
  super(GCN, self).__init__()
  self.conv1 = GCNConv(in_channels, hidden_channels)
  self.conv2 = GCNConv(hidden_channels, out_channels)
  self.dropout = dropout
  self.bn = torch.nn.BatchNorm1d(hidden_channels)

 def forward(self, x, edge_index):
  x = self.conv1(x, edge_index)
  x = self.bn(x)
  x = F.relu(x)
  x = F.dropout(x, p=self.dropout, training=self.training)
  x = self.conv2(x, edge_index)
  return x

 def get_embeddings(self, x, edge_index):
  x = self.conv1(x, edge_index)
  x = self.bn(x)
  x = F.relu(x)
  return x
```

Results

The results of the model's training. It can be seen that loss reached a minimum and accuracy reaches a maximum in Figure 15

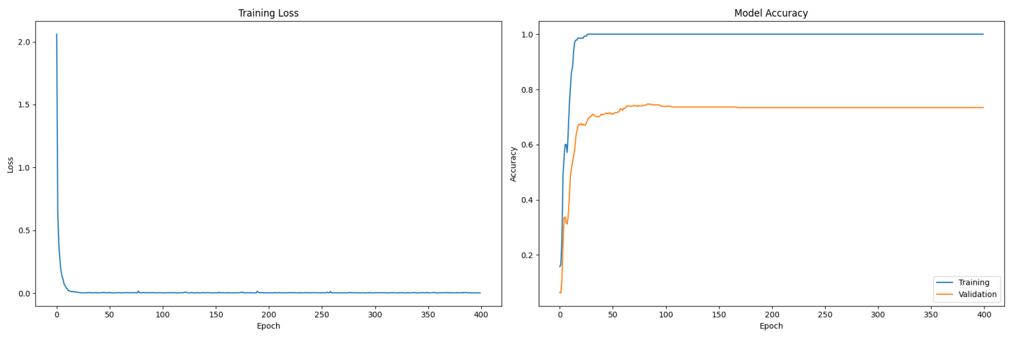
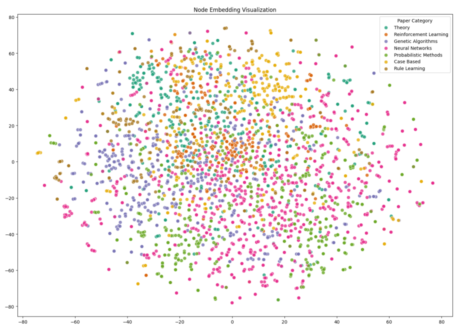


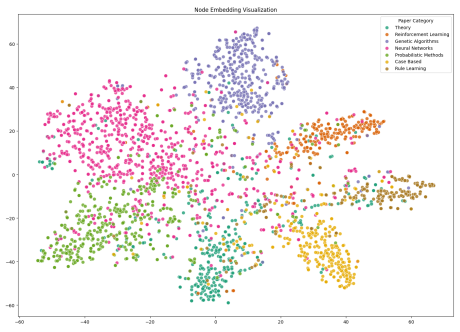
Figure 15: Accuracy and loss for the model.

Visualisation of the nodes' embedding

The trained embedding offers more perspectives when trying to predict node types from embedding.



(a) The 'untrained' embedding.



(b) The 'trained' embedding.

Figure 16: Visualization of the Cora node embedding before and after training the model.