# Social Media Course Project

## Application of Transformer-Like Deep Neural Network Architecture For Graph Edge Prediction

Bruno Guzzo
242504

Winter 2025

**Abstract**

This project explores the application of transformer-like Graph Neural Networks (GNNs) for edge prediction in semantic graph. The project aims to demonstrate the effectiveness of GNNs with architectural features inspired by transformer networks in capturing complex relationships and predicting potential connections within a graph dataset.

## 1 Introduction

The main given guideline assignment for the present project is *"Application of GNN on semantic graph generated by LLMs"*. To address such abstract assignment we chose a *"homemade"* approach in which we build from scratch a dataset and a neural network to solve the assignment.
Our baseline idea is to build a dataset of graphs form Wikipedia article links where each node is a page title, then design a custom neural network to operate on such data type.
To design such custom neural network we get inspired by the latest development in natural language processing and graph neural network.
Thus, we adopt a deep neural network architecture that extend the encoder stack of the transformer Vaswani et al. [2023] with the application of graph attention network Veličković et al. [2018].

## 2 Literature Review: Graph Attention

Graph Attention Networks (GATs) are a class of neural network architectures designed to operate on graph-structured data. They leverage a self-attention mechanism Vaswani

et al. [2023] to dynamically learn the importance of neighboring nodes, enabling effective feature aggregation while being independent of the underlying graph structure and allowing inductive learning.

### 2.0.1 Graph Attention Layer

The fundamental building block of a GAT Veličković et al. [2018] is the *graph attentional layer*. Given a graph with $N$ nodes, each represented by a feature vector $\mathbf{h}_i \in \mathbb{R}^F$ ($i \in \{1, \ldots, N\}$), the objective is to produce a new set of node features, $\mathbf{h}'_i \in \mathbb{R}^{F'}$.

### 2.0.2 Feature Transformation

Initially, each node's feature vector is linearly transformed:

$$\hat{\mathbf{h}}_i = \mathbf{W}\mathbf{h}_i, \tag{1}$$

where $\mathbf{W} \in \mathbb{R}^{F' \times F}$ is a learnable weight matrix.

### 2.0.3 Attention Mechanism

Next, an attention mechanism is employed to compute the importance of node $j$'s features to node $i$. This is achieved through a shared attentional mechanism $a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$:

$$e_{ij} = a(\hat{\mathbf{h}}_i, \hat{\mathbf{h}}_j). \tag{2}$$

In practice, $a$ is often implemented as a single-layer feed-forward neural network with parameters $\mathbf{a} \in \mathbb{R}^{2F'}$:

$$e_{ij} = \text{LeakyReLU}\left(\mathbf{a}^T[\hat{\mathbf{h}}_i || \hat{\mathbf{h}}_j]\right), \tag{3}$$

where $||$ denotes concatenation and LeakyReLU is a nonlinear activation function.

### 2.0.4 Masked Attention

To incorporate graph structure, the attention mechanism is masked. The coefficients $e_{ij}$ are only computed for nodes $j \in \mathcal{N}_i$, where $\mathcal{N}_i$ is the set of neighbors of node $i$ (including $i$ itself).

### 2.0.5 Normalization

The coefficients are then normalized using a softmax function over the neighborhood of each node:

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in \mathcal{N}_i} \exp(e_{ik})}. \tag{4}$$

#### 2.0.6 Feature Aggregation

Finally, the new feature vector for node $i$ is calculated by aggregating the transformed features of its neighbors, weighted by the normalized attention coefficients:

$$\mathbf{h}'_i = \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}\hat{\mathbf{h}}_j\right), \tag{5}$$

where $\sigma$ is a nonlinear activation function.

#### 2.0.7 Multi-Head Attention

To stabilize the learning process and capture different aspects of node relationships, multi-head attention is utilized. The operation of the graph attention layer is performed $K$ times in parallel, each with a separate transformation and attention parameters $\mathbf{W}^k$ and $\mathbf{a}^k$ (respectively). The output of each head is a new node feature vector, $\mathbf{h}_i^{'k}$.

#### 2.0.8 Concatenation

The output from the different heads can be concatenated together:

$$\mathbf{h}'_i = \Big\|_{k=1}^{K} \sigma\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k\hat{\mathbf{h}}_j^k\right). \tag{6}$$

In this case, the final output feature vector will have the dimension $KF'$.

#### 2.0.9 Averaging

Alternatively, for the final layers, the outputs can be averaged followed by the final activation:

$$\mathbf{h}'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\left(\sum_{j\in\mathcal{N}_i} \alpha_{ij}^k\hat{\mathbf{h}}_j^k\right)\right). \tag{7}$$

Here, the final output feature vector has the dimension $F'$.

## 3 Wikipedia Articles Link Dataset

Given the necessity of a consistent graph dataset to train our GNN we chose to build a suited dataset to address the project needs. Each data point of our dataset is a graph where each node is a Wikipedia article title and the edges represent links between them. The presence of an edge $(u, v)$ means that the article $u$ cite article $v$ in its body or vice-versa. Following this principle we obtain a non-directed graph where the edges indicate that two articles are somehow related.

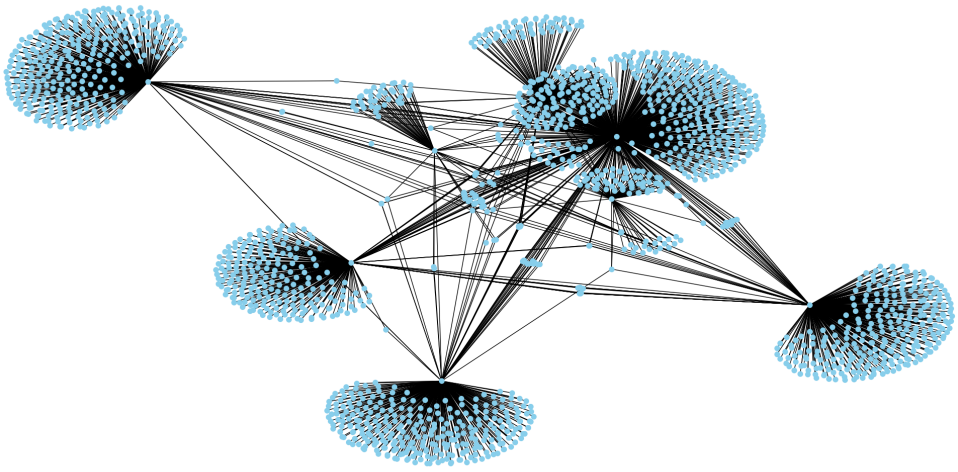

Figure 1: A graph of 2000 nodes built form the root article title 'Sustainability'.

### 3.1 Graph Extraction Algorithm

To obtain the graphs of our dataset we adopt a **breadth first strategy** to explore Wikipedia article links starting for a **root article**. This prioritize the inclusion of the immediate neighborhood of the root article. We adopt this methodology against a **depth first strategy** so we can have all the immediate neighbors of the root node, thus leading to a **robust representation** of the root article. This is particular helpful when using GNN where the network learn a node representation using the information present in it's first neighbors.

The final graph produced by our algorithm is limited by a maximum size of **20000 nodes**. This allow us to obtain a graph that capture the deep structure of Wikipedia.
Below we present a simplified **pseudo-code** of the BFS algorithm used, the final version is **recursive** and have several constraint to enforce graph consistency.

### 3.2 Node Embedding

To efficiently use our dataset with GNN it is needed to **embed each node** in way that it can be processed in a numerical way. To do so, we relay on state-of-the-art sentence embedding model to transform node labels in numeral representation. We chose to use a freely and open-source available **sentence-transformers** model: **all-MiniLM-L6-v2** Reimers and Gurevych [2019].

**Algorithm 1** Wikipedia Graph Extraction using BFS

**Require:** $rootArticleTitle$, $maxNodes$

$graph \leftarrow$ new Graph
$visited \leftarrow$ new Set, initialized with $rootArticleTitle$
$queue \leftarrow$ new Queue, enqueue $rootArticleTitle$
$graph$.addNode($rootArticleTitle$)
**while** $queue$ is not empty and $|graph.nodes| < maxNodes$ **do**
    $pageTitle \leftarrow queue$.dequeue()
    $page \leftarrow$ getWikipediaPage($pageTitle$) ▷ Handles Page/Disambiguation Errors
    **if** $page$ is not valid **then**
        **continue** ▷ Skip to the next page in queue
    **end if**
    **for** $linkTitle$ in $page.links$ **do**
        **if** $|graph.nodes| \geq maxNodes$ **then**
            **break** ▷ Exit loop
        **end if**
        **if** $linkTitle \in visited$ **then**
            $graph$.addEdge($pageTitle$, $linkTitle$)
        **else**
            $visited$.add($linkTitle$)
            $queue$.enqueue($linkTitle$)
            $graph$.addNode($linkTitle$)
            $graph$.addEdge($pageTitle$, $linkTitle$)
        **end if**
    **end for**
**end while**
**if** not isConnected($graph$) **then**
    **Error:** Graph is not connected
**end if**
**return** $graph$

It maps sentences and paragraphs to a **384 dimensional dense vector space** and can be used for tasks like **clustering** or **semantic search**. **all-MiniLM-L6-v2** is derived from the MiniLM Wang et al. [2020] architecture and it leverages a distilled version of the **BERT** (Bidirectional Encoder Representations from Transformers) Devlin et al. [2019] model.
The description of this model is outside the scope of this project and it is already been exposed in our previous natural language processing project.,

### 3.3 Final Dataset

The final dataset has been generated by executing the **algorithm 1** on a list of Wikipedia article titles and saving the graph information (nodes and edges) in JSON files. We used a list of Wikipedia article titles related to **sustainability and development** (around 300), but we have also included the **top 50 most visited** article in 2024. This would ensure that the final GNN model will preserve a small but broad **generalization capability**.

However, to ensure fast data loading, we have also created a **tensor** file version of the dataset which include node embedding and labels as well. Thus, the final dataset is composed of **389 graphs** with **20000 nodes each**, this amount to roughly **7 million of nodes**. It worth it to note that, of course, some graphs could share common node labels.

## 4 Graph Neural Network Architecture

Before to get to the final network architecture it is important to remark decisions and ideas that lead us to develop such architecture.

- **Problem complexity**: The link prediction task is not easy, especially when dealing with thousands of nodes and edges. It is needed to capture relation between distant nodes. So, the propagation of the information form nodes to nodes is crucial to ensure optimal result. For this reason, we could hypnotize the need for a **deep neural network** to address the long information transfer needed.

- **Dealing with graphs**: Each data point of our dataset ia a huge graph, so it is needed to use state-of-the-art graph neural network methodology to deal with such objects. We chose to adopt the aforementioned **graph attention** Veličković et al. [2018] 2 since it has shown better results than convolutional graph neural network Kipf and Welling [2017].

- **Dealing with words**: Despite he graphic nature of the problem we are fundamentally trying to learn relation from words and small phrases. This is a common problem of **NLP** (Natural Language Processing). So, it is natural to look to the

solutions applied in such filed to take inspiration from. This intuition lead us to chose a sub-layer structure similar to the one used in the well-known **transformer architecture** Vaswani et al. [2023].

## 4.1 Network Architecture In Detail

Our network architecture for link prediction on graph structured data is composed by the `GatModel` class and it is built around a custom `DeepGATBlock` module. The complete architecture is designed to process graph data, and performs node embeddings, multi-layered graph attention, and link prediction through a dot product decoder.

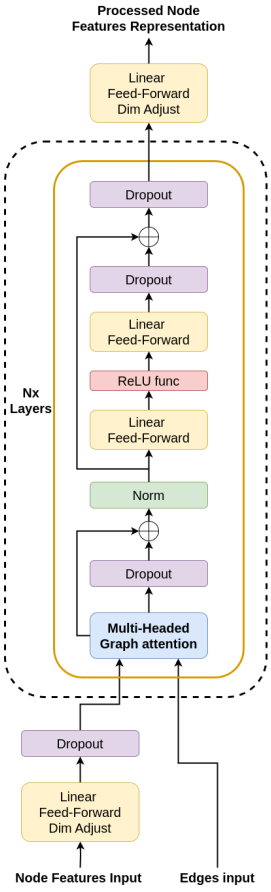


Figure 2: Graphical representation of the our final **graph neural network**.

The `GatModel` model consists of the following components:

1. An input dropout Srivastava et al. [2014] layer to regularize input features.

2. An input linear layer, $L_{in}$, that projects the input features into the hidden feature space.

3. A `DeepGATBlock`, which applies multiple levels of GAT layers, dropout, normalization, and Feed-Forward layers, as we will discuss in next section.

4. An output linear layer, $L_{out}$, which transforms the processed hidden features into the final embedding space.

#### 4.1.1 Mathematical Representation Of `GatModel`

The **encoding** process can be mathematically expressed as follows:

1. Given an input node feature matrix $X \in \mathbb{R}^{N \times F_{in}}$, where $N$ is the number of nodes and $F_{in}$ is the number of input features.

2. The input features undergo a linear transformation and dropout:
   $X' = Dropout(L_{in}(X))$, where $L_{in} : \mathbb{R}^{F_{in}} \to \mathbb{R}^{H}$, and $H$ is the number of hidden channels.

3. The transformed features, $X'$, are processed by the `DeepGATBlock`:
   $Z = \text{DeepGATBlock}(X', E)$, where $E$ represents the edge index of the graph.

4. Finally, a linear projection is applied to the GAT block output: $Z_{out} = L_{out}(Z)$, where $L_{out} : \mathbb{R}^{H} \to \mathbb{R}^{F_{out}}$ and $F_{out}$ is the number of output channels.

The **decoder** then computes the logits of edge existence based on the **dot product of node embeddings** extracted by the encoder: $\text{logits}_{ij} = (z_i \cdot z_j)$, where $z_i$ and $z_j$ are the embeddings of node $i$ and $j$, respectively.

#### 4.1.2 `DeepGATBlock` Architecture

The `DeepGATBlock` is a modular block composed of a series of *levels*, each of them processing node embeddings using a combination of graph attention mechanism, feed-forward network and normalization technique. Each level consists of the following operations:

1. A **multi-head graph attention layer**, $GAT$, which attends to the features of neighboring nodes. The output of the multi-head attention is not concatenated but rather averaged to match the input dimension. Given a node feature matrix $X \in \mathbb{R}^{N \times H}$ and an edge index $E$, the GAT layer calculates node representation according to the Graph Attention mechanism: $X_{att} = GAT(X, E)$, where the result $X_{att}$ is in $\mathbb{R}^{N \times H}$.

2. An attention **dropout layer** to regularize attention output.

3. A **residual connection** between input features and the output of the graph attention layer, followed by a layer normalization Ba et al. [2016]. The norm operation can be expressed as: $X_1 = LayerNorm(X + Dropout(X_{att}))$.

4. A **two-layer feed-forward network**, $FFN$, each of which apply a linear transformation, non-linear ReLU activation (in between linear transformation), and dropout layer on the output. Given an input features $X_1$, the network can be formalized as: $X_{ff1} = ReLU(L_1(X_1))$, and $X_{ff2} = Dropout(L_2(X_{ff1}))$. Where $L_1 : \mathbb{R}^H \to \mathbb{R}^H$ and $L_2 : \mathbb{R}^H \to \mathbb{R}^H$ represent the linear transformation in the two layer FFN block.

5. A **residual connection** between the input features of feed-forward networks, $X_1$ and it's output, $X_{ff2}$, followed by a layer normalization. The norm operation can be expressed as: $X_{out} = LayerNorm(X_1 + X_{ff2})$.

The final output of the `DeepGATBlock` is further processed by a **dropout** to regularize it.

#### 4.1.3 Mathematical Representation Of `DeepGATBlock`

Given a number of levels $L$:

1. for each level $l = 1, \ldots, L$:
   (a) $X^l_{att} = GAT(X^{l-1}, E)$
   (b) $X^l_1 = LayerNorm(X^{l-1} + Dropout(X^l_{att}))$
   (c) $X^l_{ff1} = ReLU(L^l_1(X^l_1))$
   (d) $X^l_{ff2} = Dropout(L^l_2(X^l_{ff1}))$
   (e) $X^l = LayerNorm(X^l_1 + X^l_{ff2})$
2. Final output: $X_{out} = Dropout(X^L)$

### 4.2 Chosen Architecture Parameters

Our final network architecture is composed of $L = 4$ levels of `DeepGATBlock` to address the necessity of a deep network. Other architectural parameter choice include:

1. The number of **input features** $F_{in} = 384$ driven by the output size of the chosen embedding model 3.2.

2. The number of **hidden channels** $H = 128$ to avoid long training time while maintaining a dense space for embeddings representations.

3. A low number of **output channels** $F_{out} = 64$ to allow fast decoder computation when decoding edges logits using **dot product**.

4. A higher **dropout probability** $Dropout_p = 0.75$ to avoid over-fitting and facilitate the network training. We also choose such higher probability to deal with he choice of a $L = 4$ deep network.

5. $K = 2$ graph attention heads for each network layer. This allows each level to learn **two different aspects** of nodes relationships providing the network with an enhanced data adaptation capability.

# 5 Network Training

The training of the Graph Neural Network (GNN) is performed using a **supervised learning approach**, aiming to predict the existence of edges in a given graph. The process involves feeding the GNN with a set of training graphs (form our aforementioned dataset), and comparing the network's predictions with the actual edge structures.

## 5.1 Training Procedure

The training loop iterates over a fixed number of epochs. Within each epoch, the training dataset is traversed, with the following steps being performed for each graph in the dataset:

1. **Forward Pass:** The node features $\mathbf{X}$ and the edge indices $\mathbf{E}$ of the graph are passed through the encoder to generate node embeddings $\mathbf{Z}$. The encoder consist of a stack of attention based blocks defined by:

$$\mathbf{Z} = \text{Encoder}(\mathbf{X}, \mathbf{E}) \tag{8}$$

   The final node embedding is used by the decoder to predict edge logit existence.

2. **Negative Sampling:** To enrich training, we create a set of negative edges $\mathbf{E}_{neg}$ using an under-sampling approach by sampling $n_{neg}$ non-existing edges based on a given rate $r$. The $n_{neg}$ is computed by scaling of a constant rate $R_{neg}$ the number of positive edges $n_{pos}$ in the graph: $n_{neg} = n_{pos} \times R_{neg}$.

3. **Edge Logit Prediction:** The node embeddings are used to predict logits for both positive edges (existing edges) and negative edges (non-existing edges). This is done with the decoder based on a dot product:

$$\text{logits} = \text{Decoder}(\mathbf{Z}, \mathbf{E}, \mathbf{E}_{neg}) \tag{9}$$

   More specifically the decoder takes two vectors $z_i$, $z_j$ extracted from the nodes embedding matrix $\mathbf{Z}$ according to the corresponding edge and compute the logit by a dot product: $logit_{ij} = z_i \cdot z_j$. The same operation is performed also for the negative sampled edges $\mathbf{E}_{neg}$.
   We choose this **imbalanced approach** to reduce the number of operation needed to compute the loss.

4. **Loss Computation:** A **binary cross-entropy loss** function is used to evaluate the prediction against a binary label: 1 for positive edges (existing edges) and 0 for negative edges (non-existing edges). The loss is given by:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\sigma(\text{logit}_i)) + (1-y_i)\log(1-\sigma(\text{logit}_i))\right] \tag{10}$$

where $\sigma$ represents the sigmoid function, $y_i$ is the true label for the $i$-th edge, and $\text{logit}_i$ is the predicted logit.

5. **Backpropagation and Optimization:** The calculated loss is backpropagated through the network, and the model's parameters are updated using the **AdamW optimizer** Loshchilov and Hutter [2019] to minimize the loss.

### 5.2 Loss Function

The loss function used is the **Binary Cross-Entropy with Logits Loss**. This loss function is chosen because it naturally handles the classification between binary labels representing existing or non-existing edges, by combining the **sigmoid activation** and the **binary cross-entropy** into a single efficient computation.

### 5.3 Metrics

Several metrics are computed during training and evaluation to measure model performance:

- **Area Under the Receiver Operating Characteristic Curve (AUC):** The Area Under the Curve (AUC) is used to quantify how well the model discriminates between positive and negative edges. This metric is computed by:

    1. Predicting the existence of edges using the trained model for a given graph data point with edges $\mathbf{E}$ and the negative sampled edges $\mathbf{E}_{neg}$. The final prediction is obtained using the sigmoid function: $\hat{y} = \sigma(\text{logits})$
    2. Computing the AUC score using a combination of labels and predicted values (obtained by the application of the sigmoid function) where $y = 1$ if the edge is in $\mathbf{E}$ and $y = 0$ if the edge is in $\mathbf{E}_{neg}$.

### 5.4 Agnostic Area Under the Receiver Operating Characteristic Curve (A-AUC)

In addition to standard link prediction AUC, an additional metric is introduced to better evaluate the model's capability in capturing semantic similarity between nodes: the *agnostic* area under the receiver operating characteristic curve (A-AUC). This metric is calculated using the following steps:

1. **Node Sampling:** A subset of node features is sampled randomly from the whole set according to a sampling rate $R_{cos}$.

2. **Node Embedding Extraction:** The sampled node features are used by the model to get the corresponding node embeddings through the encoder.

3. **Adjacency Prediction Matrix Computation:** A prediction matrix is computed using the dot product between the sampled node embeddings with the application of the sigmoid function. Each cell of this matrix indicates the predicted probability of a connection between the two nodes involved in the dot product.

4. **Node Similarity Matrix Computation:** The cosine similarity between the sampled node features is computed.

5. **Ground-Truth Adjacency Matrix Generation:** A ground-truth matrix is generated from the cosine similarity matrix where cells with similarity value above a predefined threshold ($T_{cos}$) are assigned a value of 1, and all the remaining are set to zero.

6. **A-AUC Calculation:** A-AUC score is obtained by considering the ground-truth matrix as the true labels and the predicted adjacency matrix as the probability scores to be used in AUC computation.

The **A-AUC metric** provides a measure of how well the GNN captures the **semantic similarity between node**, independently from the explicit graph structure. It evaluate if the model is able to predict a potential link between node when such link is not explicitly in the training data but implied by feature similarity.

### 5.4.1 Training Parameters Used

The final network training have been executed using the following fix parameters:

- **Number of epochs**: $N_{epochs} = 2$
- **Learning rate**: $lr = 0.001$
- **Dataset splits and usage**: In all the training experiments the used dataset size has been limited to a fixed series of values, while the split percentage of **training set** and **test set** has been maintained constant to 90% and 10% respectively.
- **Negative edge sampling rate**: $R_{neg} = 0.3$
- **A-AUC cosine similarity threshold**: $T_{cos} = 0.7$
- **A-AUC nodes sampling rate**: $R_{cos} = 0.1$

# 6 Evaluation

We have run mainly **three training experiment** to evaluate how well the network adapt itself to the given graph. To do so we have used the already mentioned training parameters while limiting the dataset size in an incremental way.
Below, we show a **series of figure** to better understand the network behavior during training.

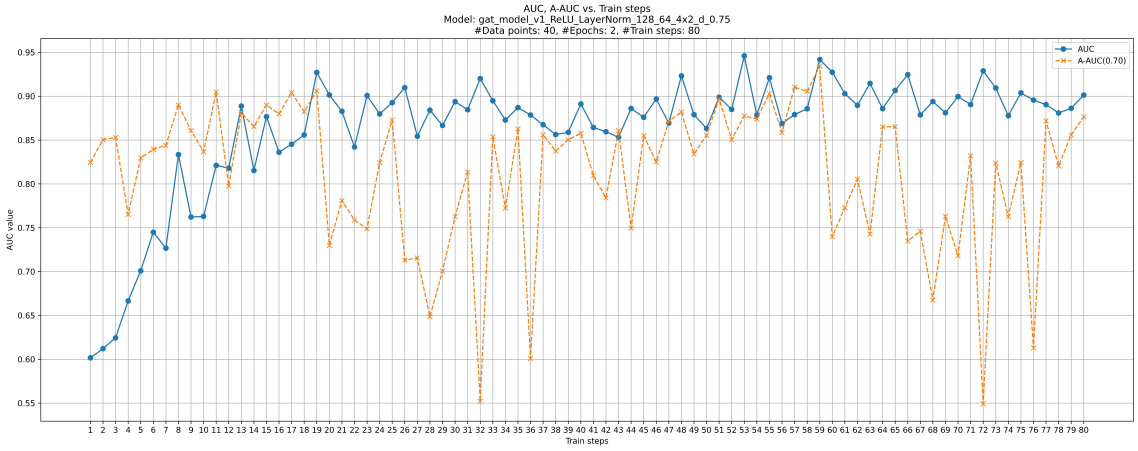


Figure 3: **AUC vs A-AUC** during training with 2 epochs and a train-set of 40 graphs.

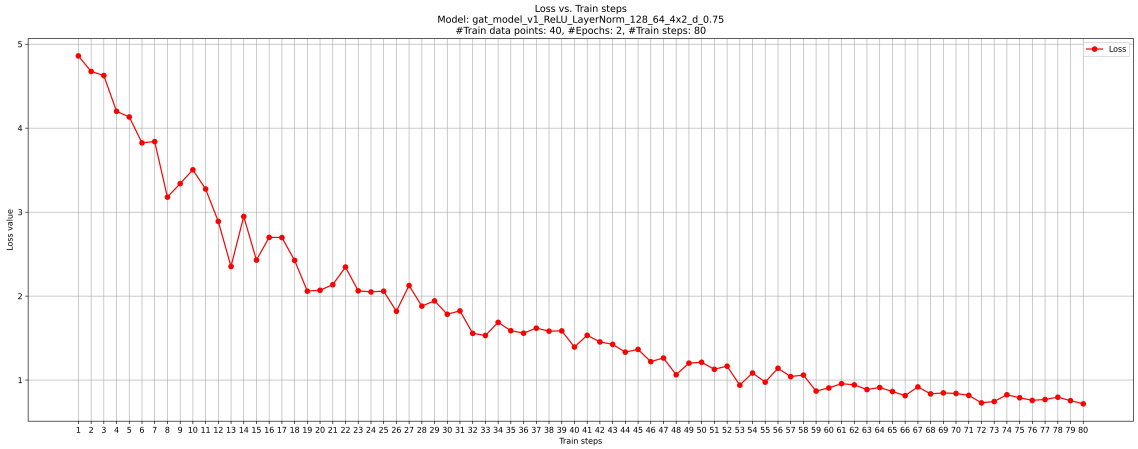


Figure 4: **Loss** during training with 2 epochs and a train-set of 40 graphs.

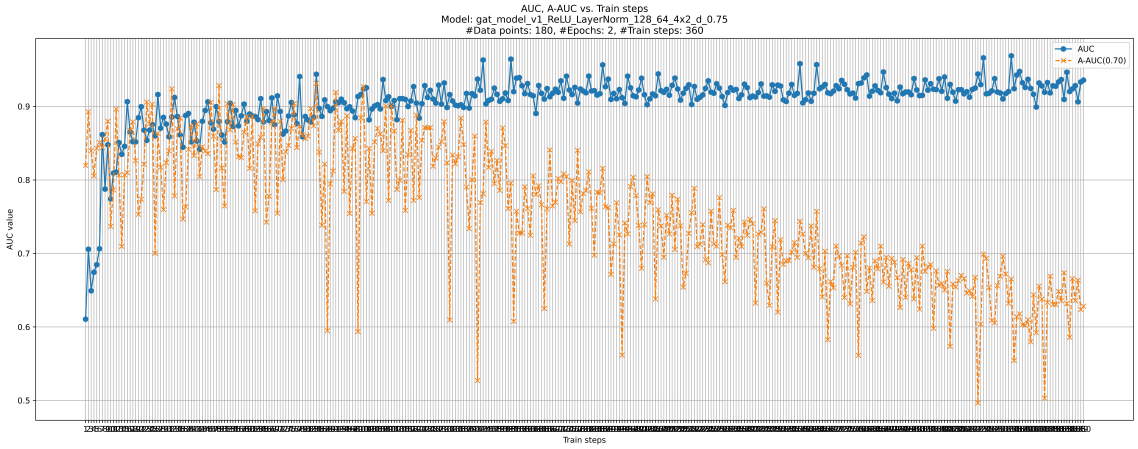


Figure 5: **AUC vs A-AUC** during training with 2 epochs and a train-set of 180 graphs.

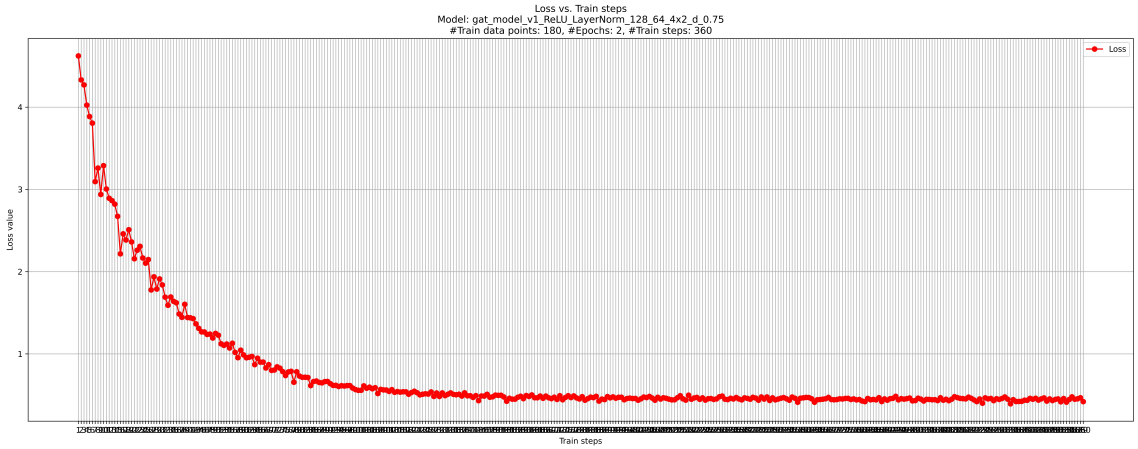


Figure 6: **Loss** during training with 2 epochs and a train-set of 180 graphs.

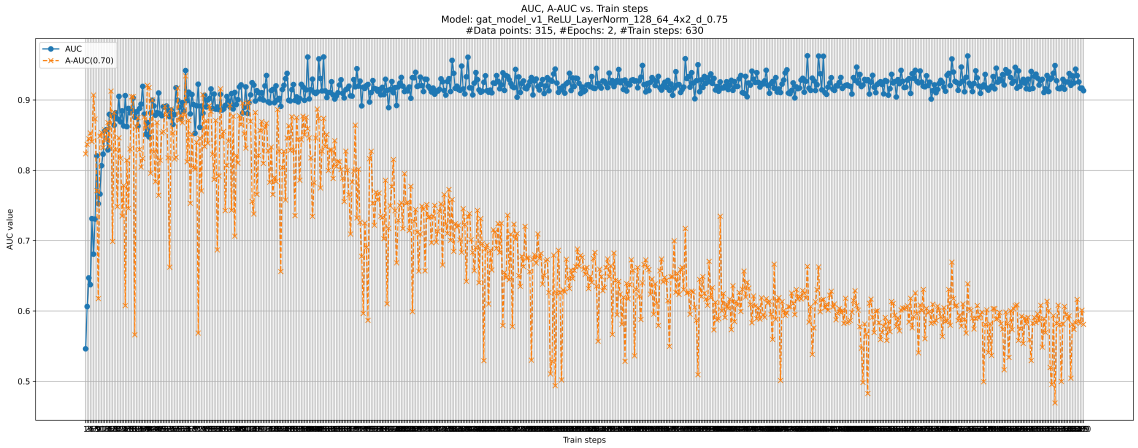


Figure 7: **AUC vs A-AUC** during training with 2 epochs and a train-set of 315 graphs.

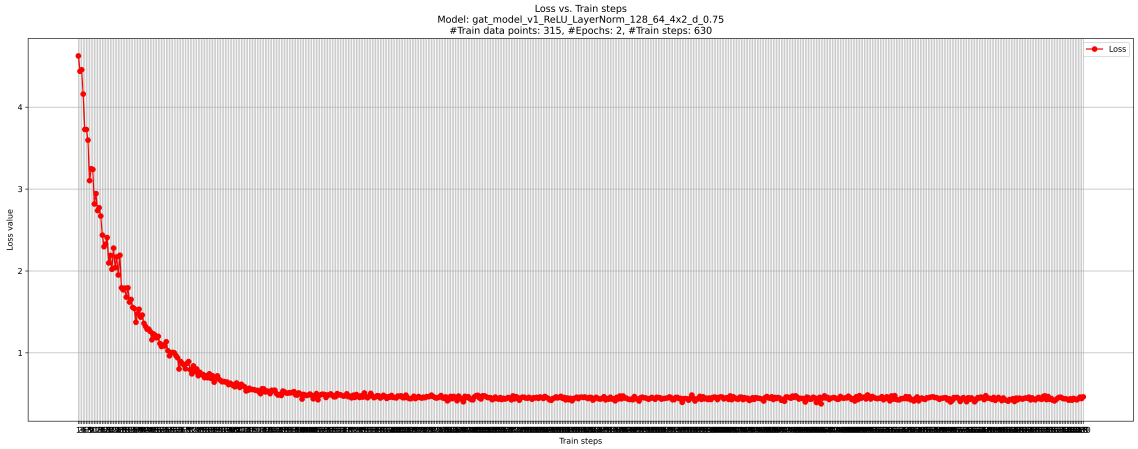


Figure 8: **Loss** during training with 2 epochs and a train-set of 315 graphs.

### 6.0.1 Training Process Evaluation

From the previous diagrams it is evident how the network is capable of a quick adaption to the provided graph data, although it is also clear that increasing the training set do not provide a sensible reduction in term of AUC an Loss after a certain threshold.
However, thanks to the defined **A-AUC metric 5.4** we can notice a curious phenomenon. If we consider the experiment with a consistent number of training steps (5 and 7) we can observe a decreasing thread in A-AUC while maintaining a higher AUC.

The reason of this behaviors lies in the dataset definition where a couple of referenced articles titles could **not be semantic close**. For example, the Wikipedia article *Sustainability* cite the page *Venn diagram*, of course these two are not semantically close. Thus, the embeddings do not capture such relationship making the **A-AUC lower** while training.

| Training set size | Epochs | Loss | **AUC** | **A-AUC** |
|---|---|---|---|---|
| 40 | 2 | 0.99754 | **0.89362** | **0.81470** |
| 180 | 2 | 0.45256 | **0.92331** | **0.68553** |
| 320 | 2 | 0.44383 | **0.92472** | **0.60340** |

Table 1: **Averaged metrics** of the second (and last) training epoch.

It's worth it to notice that the A-AUC has a broad variance given the random underselling used in it's calculation, however the averaged epoch metric allow us to confirm our understanding. Taking in account this behavior we can state that **Wikipedia articles links do not depend mainly on the semantic closeness**.

### 6.0.2 Testing Results

The three training experiment conducted includes, of course, a testing phase with the 10%/20% of the dataset used for such purpose. We present all the testing results in the following charts 9, 10 and 11.

As noted during the training phase, we have a similar behavior for the testing phase where we obtained **higher AUC** value in all three test experiments and **decreasing A-AUC** as well. It is also worth it to note that getting a small **AUC increment** ($< 0.3$) implies the need of a **8 time** training set size increment 2.

| Training set size | Epochs | **AUC** | **A-AUC** |
|---|---|---|---|
| 40 | 2 | **0.900** | **0.772** |
| 180 | 2 | **0.922** | **0.628** |
| 320 | 2 | **0.927** | **0.578** |

Table 2: Averaged **test results metric**

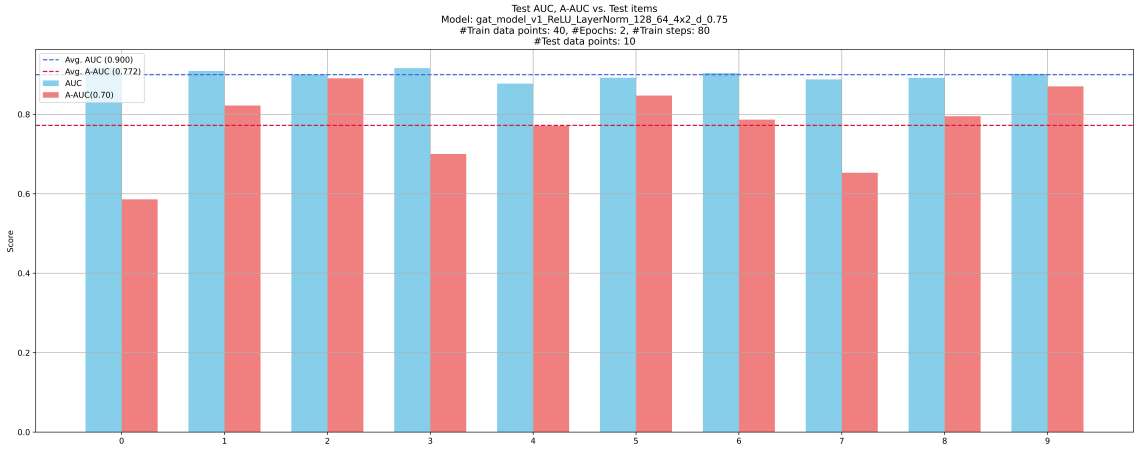


Figure 9: Evaluation of the model trained with **40 data points** for 2 epochs.

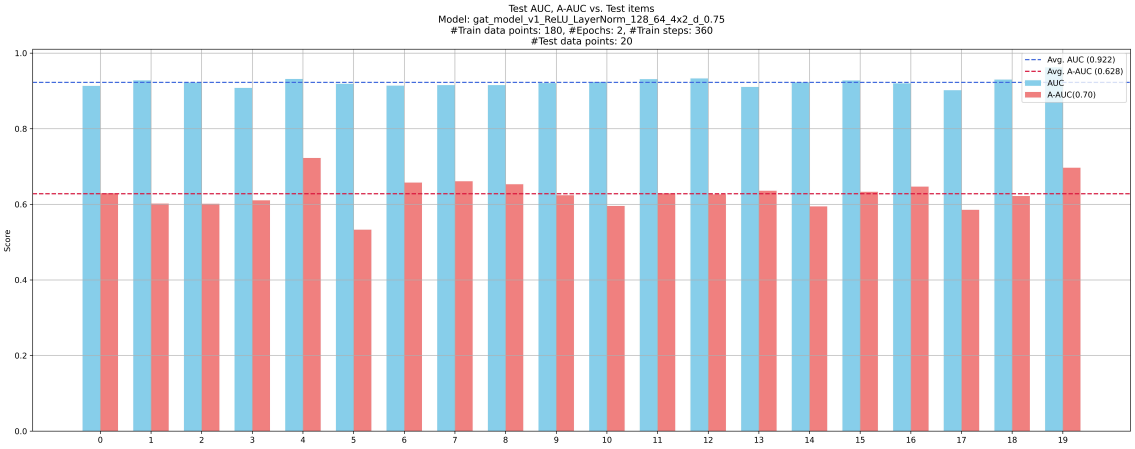


Figure 10: Evaluation of the model trained with **180 data points** for 2 epochs.

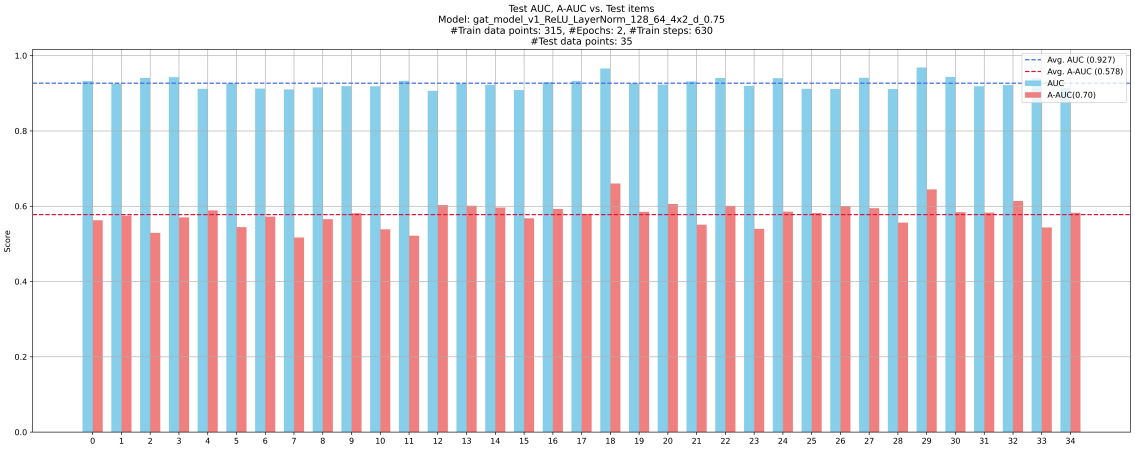


Figure 11: Evaluation of the model trained with **315 data points** for 2 epochs.

### 6.0.3 Conclusion & Improvements

Despite the *"home-made"* approach used, we have successfully shown how to build a **deep but very small** graph neural network wile still being able to get good performance.

Further work to improve the present architecture could explore different activation function and normalization layer or perhaps experiment with various network architecture such as MOE (Mixture Of Experts).

# 7 Real Case Application

To test the developed neural network on a real case of application we choose to build two graph without edges with a LLM and then evaluate how well the network predict the edges.

We used Llama 3.2B LLM with several prompt built by a template that asks for the generation of 10 arguments related to a provided topic. This process has been repeated multiple times in order to get two node sets, one related to the given domain (sustainable development) and one not related to the given domain - medicine.

# 8 Methodology

# 9 Real Case Application

## References


Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL https://arxiv.org/abs/1810.04805.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017. URL https://arxiv.org/abs/1609.02907.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019. URL https://arxiv.org/abs/1711.05101.

Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. URL https://arxiv.org/abs/1908.10084.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014. URL http://jmlr.org/papers/v15/srivastava14a.html.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018. URL https://arxiv.org/abs/1710.10903.

Wenhui Wang, Furu Wei, Li Dong, Hangbo Bao, Nan Yang, and Ming Zhou. Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers, 2020. URL https://arxiv.org/abs/2002.10957.