Big Data Project

Graph Neural Networks

Genes and pathways embedding

Locicero Giorgio, 1000024196

Contents

1 Objectives of the project **4**

2 The whole model **4**

2.1 Graph Neural Network 5

2.1.1 General framework 5

2.1.2 Graph convolutional network 9

2.1.3 Graphsage 11

2.1.4 Genes embeddings 13

2.1.5 Further optimization and ideas 18

2.2 Pathway embedding 20

2.2.1 Gradient descent 20

3 Conclusions and further research on the topic **25**

Introduction

Biological pathways are a way to represent sequences of interactions among molecules in a cell that leads to a certain product or a change in a cell or system in an organism. Pathways could be of many types, for this project we are interested in genetic pathways (**Gene Regulatory Network**). A gene (or genetic) regulatory network (GRN) is a collection of molecular regulators that interact with each other and with other substances in the cell to govern the gene expression levels of mRNA and proteins which, in turn, determine the function of the cell. GRN also play a central role in morphogenesis, the creation of body structures, which in turn is central to evolutionary developmental biology (evo-devo) [1]. In this project, the main focus is to find features for entering nodes in the graph(obtained from the pathways) that have no features based upon the features of already embedded genes(the features of the genes are the co-expression of the single gene to a single patient-person) and to find pathway embeddings based upon the features predicted(At the end of the project, the features associated with every gene, that is real numbers linked to patients, will be used to predict pathway features with a model that minimizes a loss function that will be defined in 2.2). During this project, the library and API used are StellarGraph(for the implementation of the GNN, that uses Tensorflow)[2] and Tensorflow. Another alternative to stellargraph is **spektral**[3] but the core algorithms implemented are more or less the same (because the code for GNN and GraphSage are taken from the original code[4][5]) At the end, the pathway embedding will be confronted for two types of patients(some sane patients, and some with some disease)

1 Objectives of the project

As already introduced before, the main objectives of this project are to define a model to predict gene labels based on a graph(induced from the pathways) based on a user defined loss that maps close or related nodes closer(seen in 4), and to find the pathways embeddings from the genes embeddings. With these embeddings the main focus is to find some spatial relation between different data(different patients with no diseases, patients with different diseases), like clusters or confronting the density in the spatial domain of the embeddings.

2 The whole model

To understand what is the aim and the strategies used in this research, the model is introduced in its most important features which will, in turn, be dissected in the next chapters.

The whole model firstly predict genes embeddings with a GNN model that is trained with a user defined loss function that will be seen and commented[4], secondly, It predicts **pathways** embeddings based on another loss function[6] and with a custom model[5](implemented in tensorflow) that could be seen as an alteration of the GCN model (2.1.2).

The theoretical model to compute the embeddings for the genes with a GNN is the following(1,2,3):

$$h_{g_i}^0 = g_i \tag{1}$$

Where g_i is the tensor of features of the gene i, that is the expressions of a gene to a group of patients. This is the base case of the embedding for the genes and could be seen as the final layer of the graph neural network of the paths from gene i to its neighborhood genes.

$$h_{g_i}^k = \sigma(W_k \sum_{u \in N(g_i)} \frac{h_u^{k-1}}{|N(g_i)|} + B_k h_{g_i}^{k-1}) \tag{2}$$

Where the σ is a non-linear function(RELU or Sigmoid), W_k and B_k are matrices(rank-2 tensors, even though the relation between tensors and matrices is not bijective) and are the parameters learned by the model to minimize the loss function(the methods used are Gradient Descent or variants[6], in the implementation of the model, **Adam** is used for genes embedding), this formula is a recurrent formula but could be unrolled for every layer that is part of the GNN(especially because Tensorflow does not handle really well recursive models and usually unrolls them).

The final embedding for every node will be:

$$z_{g_i} = h_{g_i}^k \tag{3}$$

The **Loss** function used to create the model for genes embeddings with a GNN model(trained with this function and the genes graph) will be the following:

$$\mathcal{L}(E) = \sum_{(u,v)\in E} ||(z_u - z_v)||_2 \tag{4}$$

With this function, genes linked by an edge with similar embeddings will be mapped closer in the embedding space while genes with different local structure will be more distant, more details will be seen in 2.1.4.

The embedding for **pathways** is defined by the following equation:

$$z_{P_i} = \sigma(W_P \sum_{u\in P_i} \frac{z_u}{|P_i|}) \tag{5}$$

The loss function used to create the model that predicts the pathways embeddings will be the following:

$$\mathcal{L}(P) = \sum_{P_i\in P} \sum_{P_j\in P} ||(z_{P_i} - z_{P_j})||_2 \tag{6}$$

This loss function needs to be minimized so the pathways embedding distances/similarity is minimized(Pathways with similar genes embeddings will be closer than pathways with little to none genes in common). The distance used will be a norm-2 but another norm should suffice and work the same way, in the implementation a stable version of the norm-2 is used to avoid unexpected behavior.

2.1 Graph Neural Network

All Graph Neural Network algorithms are conceptually related to node embedding approaches, general supervised approaches to learning over graphs, classification of nodes or graphs. In this research, there will not be an implementation of matrix factorization methods(related to spectral clustering, multi-dimensional scaling or approaches related to PageRank with random walks) because these methods were tested on the data and extremely unsatisfactory results were obtained from the models obtained, especially for the task at hand, because these type of models do not take into account any(or very little) graph structural information during training.

All models tested in this paper are the best(among the available ones) for the data given, that is **Graph Convolutional** models and inductive approaches like **GraphSage**.

2.1.1 General framework

Graphs are a kind of data structure which models a set of objects(nodes) and their relationships (edges). As a unique non-Euclidean data structure for machine learning, graph analysis

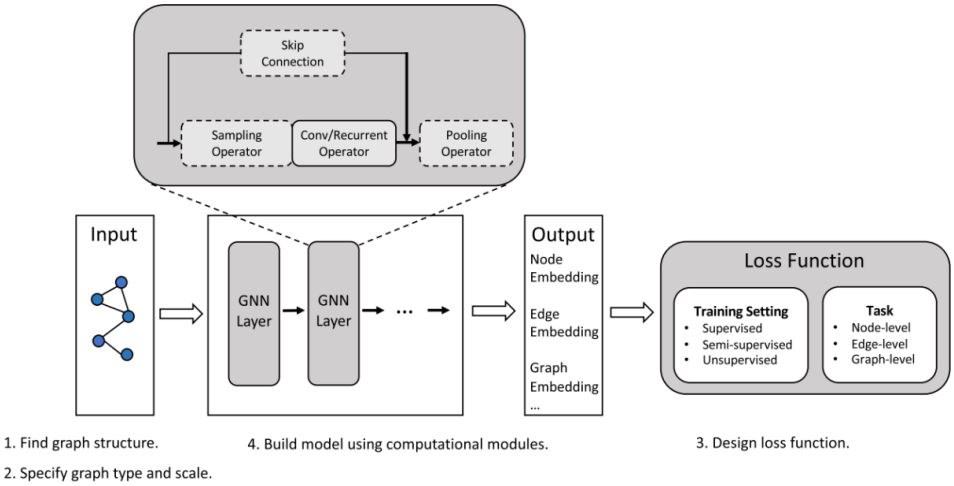


Figure 1: Pipeline for a GNN model

focuses on tasks such as node classification, link prediction, and clustering. Graph neural networks (GNNs) are deep learning based methods that operate on graph domain. Due to its convincing performance, GNN has become a widely applied graph analysis method recently. Almost all of the work on GNN is born from the transposition of the CNN models to graph data, but the passage from Euclidean data to non-Euclidean data rises a lot of concerns and problems on the definitions from CNN and the theory behind. Extending deep neural models to non-Euclidean domains, which is generally referred to as geometric deep learning, has been an emerging research area.

Another useful matter linked to Graph learning is **graph representation learning** which learns to represent graph nodes, edges or subgraphs by low-dimensional vectors. In the field of graph analysis, traditional machine learning approaches usually rely on hand engineered features and are limited by its inflexibility and high cost. Following the idea of representation learning and the success of word embedding, a lot of ways to do Graph Embeddings were born (Node2Vec is one of these embedding algorithms that uses random-walks and anonymous walks to create embeddings for nodes and graphs [7], along with other algorithms which will not be seen in this research).

Based on CNNs and graph embedding, variants of graph neural networks (GNNs) are proposed to collectively aggregate information from graph structure. Thus they can model input and/or output consisting of elements and their dependency (usually local dependency in the graph, especially for inductive models that build models based upon local neighborhoods of a defined depth, sampling is also really useful in these models, as seen in 2.1.3).

The graphs used in this research will be static(no dynamic edges during runtime or entering nodes) because the goal of this project is to find genes embeddings and pathways embeddings given the genes features and graph. The size of the graph used is not really large (around

5000 genes and 28500 edges).

The first step should be to design the loss function based on the task type and the training setting. For graph learning tasks, there are usually three kinds of tasks:

1. **Node-level** tasks focus on nodes, which include node classification, node regression, node clustering, etc. Node classification tries to categorize nodes into several classes, and node regression predicts a continuous value for each node. Node clustering aims to partition the nodes into several disjoint groups, where similar nodes should be in the same group.

2. **Edge-level** tasks are edge classification and link prediction, which require the model to classify edge types or predict whether there is an edge existing between two given nodes.

3. **Graph-level** tasks include graph classification, graph regression, and graph matching, all of which need the model to learn graph representations.

In this research, we have defined a **Node-level** loss function that uses nodes generated embeddings as well as edges between the embeddings to compute the final value, as seen in 4. We work in a semi-supervised/unsupervised environment because we have the feature to use to create the embeddings but not the embedding itself that minimizes the loss function.

The next step is **building the model** using the **computational modules**. Some commonly used computational modules are:

- **Propagation Module**. The propagation module is used to propagate information between nodes so that the aggregated information could capture both feature and topological information. In propagation modules, the **convolution operator** and **recurrent operator** are usually used to aggregate information from neighbors while the **skip connection** operation is used to gather information from historical representations of nodes and mitigate the over-smoothing problem.

- **Sampling Module**. When graphs are large, sampling modules are usually needed to conduct propagation on graphs. The sampling module is usually combined with the propagation module.

- **Pooling Module**. When we need the representations of high-level subgraphs or graphs, pooling modules are needed to extract information from nodes.

Both the GCN model and GraphSage use **Propagation modules** to propagate information between nodes, even though they use different strategies to create the whole model(the GCN propagation is aimed toward spectral and trasductive approaches, while the GraphSage model is inductive and structural based upon the neighborhood of nodes).

Other Modules and techniques could be used to optimize and upgrade GNN models as seen in figure 2(like sampling, used in GraphSage and in other methods), but the literature is too

much and, in this research, we will see in depth two models in particular that use different techniques with the same goal.

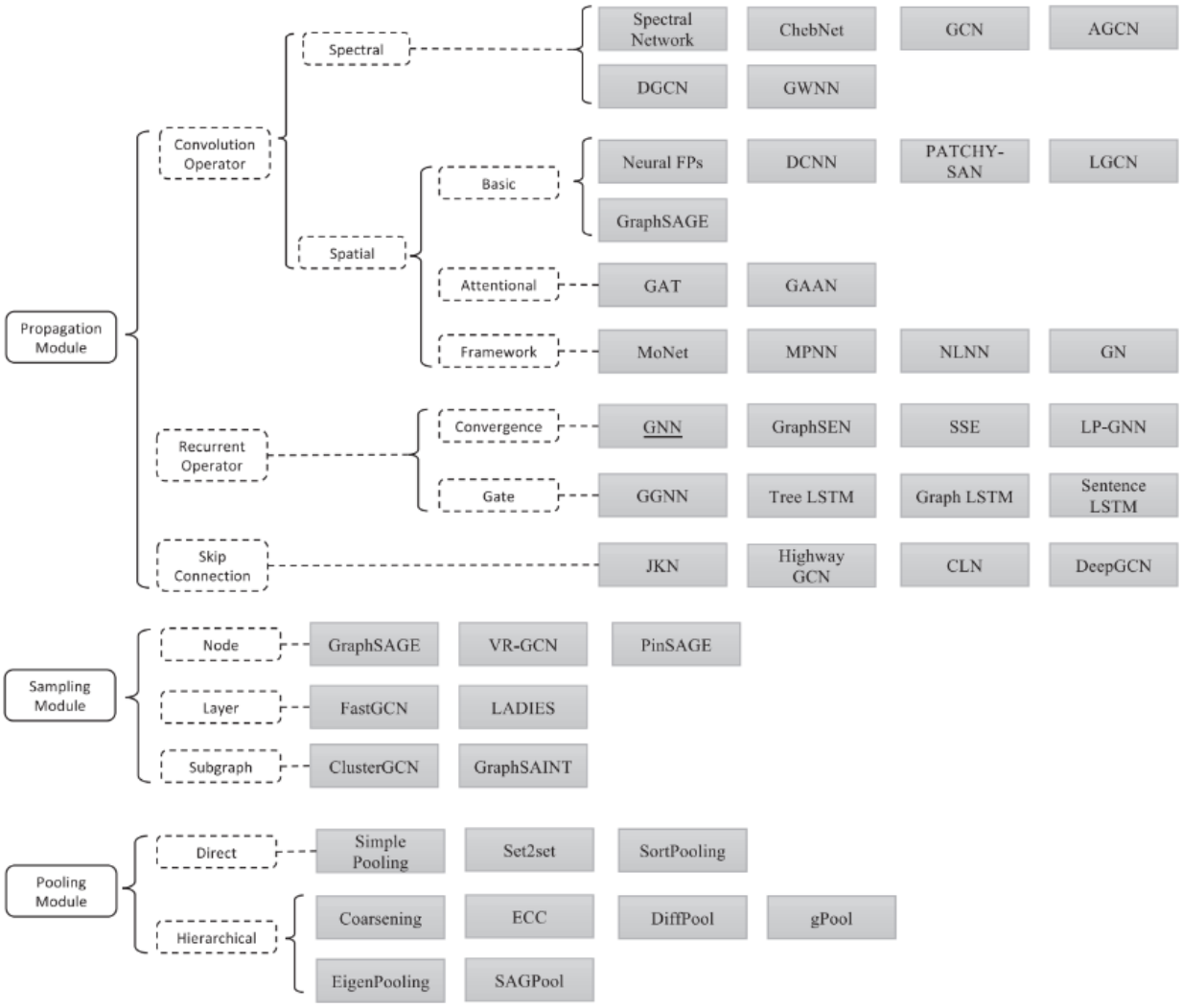


Figure 2: Different Modules and techniques used to create GNN models

For additional information about GNN and the general framework behind It, see the specifics models for GCN,GraphSage and so on, and give a look at [8] review about GNN.

2.1.2 Graph convolutional network

Graph Convolutional Networks are a transposition of Convolutional Neural Network on images and bear resemblance to spectral approaches for graph clustering and classification. GCN are called convolutional because filter parameters are typically shared over all locations in the graph(or a subset).

For these models, the goal is to learn a function of signals/features on a graph $G = (V, E)$ which takes as input:

- A feature description x_i for every node i summarized in a $N \times D$ feature matrix X (N: number of nodes, D: number of input features)
- A representative description of the graph structure in matrix form; typically in the form of an adjacency matrix A (or some function thereof)

The model produces a node-level output Z (an $N \times F$ feature matrix, where F is the number of output features per node). Every neural network layer can then be written as a non-linear function as defined in the following pattern formula:

$$H^{(l+1)} = f(H^{(l)}, A) \tag{7}$$

with $H^{(0)} = X$ and $H^{(L)} = Z$ (or z for graph-level outputs), L being the number of layers. The specific models then differ only in how $f(\_, \_)$ is chosen and parameterized.

An example of a very simple form of a layer-wise propagation rule is the following:

$$f(H^{(l)}, A) = \sigma(AH^{(l)}W^{(l)}) \tag{8}$$

where $W^{(l)}$ is a weight matrix for the l-th neural network layer and $\sigma(\_)$ is a non-linear activation function like the ReLU.

There are two limitations about the model previously described:

1. Multiplication with A means that, for every node, we sum up all the feature vectors of all neighboring nodes but not the node itself (unless there are self-loops in the graph). We can "fix" this by enforcing self-loops in the graph: we simply add the identity matrix to A.
2. A is typically not normalized and therefore the multiplication with A will completely change the scale of the feature vectors (understandable by looking at the eigenvalues of A). Normalizing A such that all rows sum to one, i.e. $D^{-1}A$, where D is the diagonal node degree matrix, gets rid of this problem. Multiplying with $D^{-1}A$ now corresponds to taking the average of neighboring node features. In practice, dynamics get more interesting when we use a symmetric normalization, i.e. $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ (as this no longer amounts to mere averaging of neighboring nodes).

The final formula after these considerations is :

$$f(H^{(l)}, A) = \sigma(D^{-\frac{1}{2}}(A + I)D^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{9}$$

This example is a way of using graph convolutional network to predict and compute features, this type of computation will be modified and used for the **path embedding** problem 2.2

The original GCN algorithm [9] is designed for semi-supervised learning in a transductive setting, and the exact algorithm requires that the full graph Laplacian is known during training. A variant of the GraphSage algorithm can be viewed as an extension of the GCN framework to the inductive setting.

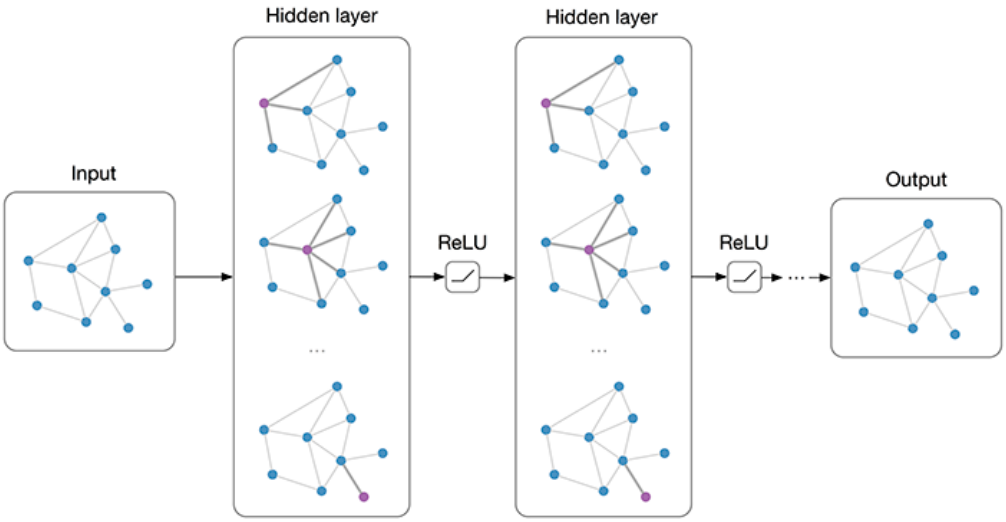


Figure 3: Multi-layer GCN with first order filters

2.1.3 Graphsage

The approach followed by GraphSage(SAmple and aggreGatE) is an **inductive** approach that takes into consideration unseen nodes during the training, in contrast of **trasductive** approaches that need all nodes to be present during the training of the model to predict embeddings. This is done by taking a subset of the neighborhood during the training of the model and aggregate the embeddings in this subset to find the node embedding. Unlike embedding approaches that are based on matrix factorization, by incorporating node features in the learning algorithm, the algorithm simultaneously learn the topological structure of each node's neighborhood as well as the distribution of node features in the neighborhood.

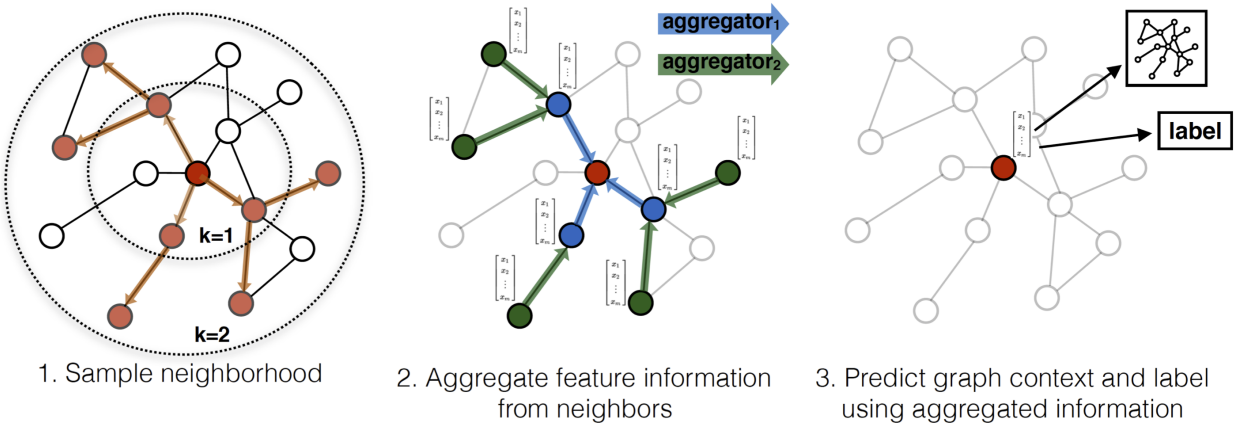


Figure 4: Sample and aggregation visualization of Graph Sage

However, in this research, a trasductive approach is considered where all nodes need to be present for the model to work, this is because the loss function needs all the edges to be present to find the minimum(or local minimum in a certain range).

Instead of training a distinct embedding vector for each node, the model firstly trains a set of aggregator functions that learn to aggregate feature information from a node's local neighborhood. Each aggregator function aggregates information from a different number of hops, or search depth, away from a given node.

The key idea behind this model is that It learns how to aggregate feature information from a node's local neighborhood. The first part of the GraphSAGE model is the training of the model with a loss function(known loss function or user defined) to find the parameters that will be used to generate the embeddings, while the second part is embedding generation (forward propagation) algorithm, which generates embeddings for nodes assuming that the GraphSAGE model parameters are learned by the first step.

The embedding generation, or forward propagation algorithm(algorithm 1), assumes that the model has already been trained and that the parameters are fixed. In particular, the model assumes that we have the parameters of K aggregator functions(denoted $\text{AGGREGATE}_k, \forall k \in$

$1, ..., K$, which aggregate information from node neighbors, as well as a set of weight matrices $W_k, \forall k \in 1, ..., K$) are learned, which are used to propagate information between different layers of the model or "search depths".

Algorithm 1: Embedding generation (forward propagation)

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

Output: Vector representations $z_v \forall v \in V$

$h_v^0 \longleftarrow x_v, \forall v \in V$;

for k = *1,...,K* **do**

  for $v \in V$ **do**

    $h_{N(v)}^k \longleftarrow \text{AGGREGATE}_k(h_u^{k-1}, \forall u \in N(v))$;

    $h_v^k \longleftarrow \sigma(W^k \cdot \text{CONCAT}(h_v^{k-1}, h_{N(v)}^k))$;

  end

  $h_v^k \longleftarrow \frac{h_v^k}{||h_v^k||_2}, \forall v \in V$

end

The intuition behind the algorithm is that at each iteration, or search depth, nodes aggregate information from their local neighbors, and as this process iterates, nodes incrementally gain more and more information from further reaches of the graph. Further considerations on the algorithm could be seen in [10]

The neighborhood is taken by a sample of k-distant nodes from the node considered.

Ideally, an aggregator function would be symmetric (i.e., invariant to permutations of its inputs) while still being trainable and maintaining high representational capacity. The symmetry property of the aggregation function ensures that the model neural network model can be trained and applied to arbitrarily ordered node neighborhood feature sets. The aggregation function used could be user-defined or taken from different aggregation architectures, the one that will be used for this research is the mean aggregator that take the element-wise mean of the vectors in the neighborhood(It is also called convolutional because It is the generalization of the filters seen in 2.1.2, with an inductive approach, so it is a linear approximation of a localized spectral convolution), that is:

$$h_v^k \longleftarrow \sigma(W \cdot \text{MEAN}(h_v^{k-1} \cup h_u^{k-1}, \forall u \in N(v))) \tag{10}$$

That is what is seen in 2(even though the old value of the node is directly multiplied and summed, but that is a detail, the formula leads to similar, almost equal, results).

An important distinction between this convolutional aggregator and the others aggregators(that will not be used in this research) is that it does not perform the concatenation operation in line 5 of Algorithm 1 i.e., the convolutional aggregator does concatenate the node's previous layer representation h_v^{k-1} with the aggregated neighborhood vector $h_{N(v)}^k$. This concatenation can be viewed as a simple form of a "skip connection" [8](this paper is

really good and a must read because It is a review of most of the strategies used for GNN and the differences/similarities between them) between the different "search depths", or "layers" of the GraphSAGE algorithm, and it leads to significant gains in performance.

The model learns a function that generates embeddings by sampling and aggregating features from a node's local neighborhood. In the main paper[10],[5] the algorithm could also be used to predict embeddings in dynamic graphs, but in the **stellargraph** implementation, the graphs used needs to be static and defined at runtime (no nodes or edges could be added to a graph at runtime to be embedded) or this is what the documentation seems to be implying(for GraphSage, the given demo takes a static graph and train the model on the graph).

2.1.4 Genes embeddings

The data at hand is highly structured, graph edges were generated from the meta-pathways of genes, edge weights are the interactions between genes in pathways(if a gene inhibits or stimulates another genes based on the value of the edge weight) and genes expressions for patients as features. Every one of these structured characteristics will be used to generate the genes embeddings, and the model used will be the one defined by the formulas seen in the beginning of this chapter 2.

	TCGA-BH-A0BC-11A-22R-A089-07	TCGA-BH-A0HA-11A-31R-A12P-07	TCGA-E2-A1LB-11A-22R-A144-07	TCGA-E2-A1LH-11A-22R-A14D-07	TCGA-E2-A1BC-11A-32R-A12P-07	TCGA-E9-A1NF-11A-73R-A14D-07	TCGA-E9-A1N9-11A-71R-A14D-07	TCGA-BH-A0H5-11A-62R-A115-07	TCGA-BH-A0DT-11A-12R-A12D-07	TCGA-A7-A0CH-11A-32R-A089-07	...
2	85990.0	63154.0	53226.0	44518.0	43854.0	83423.0	58630.0	73218.0	82395.0	93813.0	...
9	435.0	136.0	165.0	542.0	66.0	179.0	199.0	1207.0	1799.0	4584.0	...
10	13.0	1.0	2.0	6.0	0.0	1.0	343.0	19.0	68.0	104.0	...
15	10.0	3.0	1.0	1.0	2.0	1.0	4.0	12.0	5.0	12.0	...
16	6543.0	6577.0	7111.0	6129.0	4223.0	8044.0	7968.0	6046.0	6882.0	9475.0	...
...	...	...	...	...	...	...	...	...	...	...	...

Figure 5: genes features

An example of genes features used to compute embeddings could be seen in figure 5, the data is not normalized. Normalized data was also used to create the model and estimate the parameters but the results were similar to the one obtained with GraphSage(for the GCN, the normalized data reduces fluctuation of the loss function and the value is more reasonable).

The loss function used to compute the model is the one defined in previous sections 4, the loss function is modified a little in the code to account for instability(of the norm computation) and for the requirements of tensorflow and keras models(for method definitions and signatures).

Algorithm 1: Loss function used for the generation of the GNN model for genes embeddings in GCN

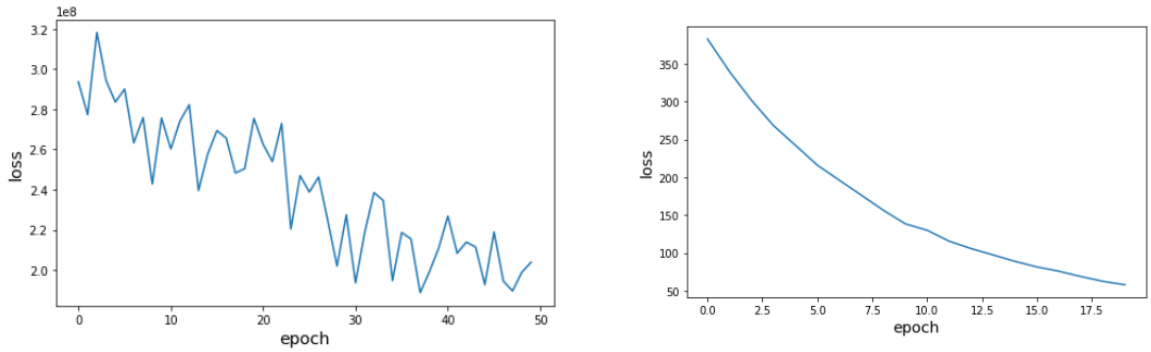
```
indexOfGenes = dict()
for i in range(0,len(controlsDK.index)):
    indexOfGenes[controlsDK.index[i]] = i

tensorEdges = tf.constant(geneEdges,tf.int32)
tensorEdgesTest = tf.map_fn(lambda row: tf.
            map_fn(lambda element:indexOfGenes[element.numpy()],tf.
                    gather(row, [0, 1]).numpy()) ,tensorEdges)

@tf.function
def difference_of_edge_genes_embedding(edges,y_pred):
    #gene1 = tf.gather(y_pred,edges[0])
    #gene2 = tf.gather(y_pred,edges[1])
    gene1 = y_pred[0][edges[0]]
    gene2 = y_pred[0][edges[1]]
    return tf.sqrt(tf.reduce_sum(tf.square(gene1 - gene2)) + 1.0e-12)


@tf.function
def genes_loss(y_true, y_pred, edges):
    loss = tf.constant(0, tf.float32)
    for row in edges:
        loss = loss + difference_of_edge_genes_embedding(row,y_pred)
    return loss

#@tf.function
def genes_loss_with_closure(edges):
    def loss_out(y_true, y_pred):
        loss = tf.constant(0, tf.float32)
        for row in edges:
            loss = loss + difference_of_edge_genes_embedding(row,y_pred)
        return loss
    return loss_out
```

This loss is specific to the GCN because the models use different representations for layers in the keras models. An important consideration about this loss function is that It implies that the underlying model is not divided in batches because different batches could contain different edge embeddings that needs to be in the same batch, this slows down the computation by a lot.

Algorithm 2: StellarGraph graph creation

```
graphWithFeatures = sg.StellarGraph(controlsDK ,geneEdges )
directedGraph = sg.StellarDiGraph(controlsDK ,geneEdges)
```

For the GCN construction, an undirected graph is used, while ,for GraphSage, a directed graph is used. Between these two strategies the directed graph seems to be the better choice because the original graph is directed and weighted to represents inhibition or stimulation of a gene to another in a pathway, this consideration will find additional proofs when the model



(a) GCN (b) GraphSage

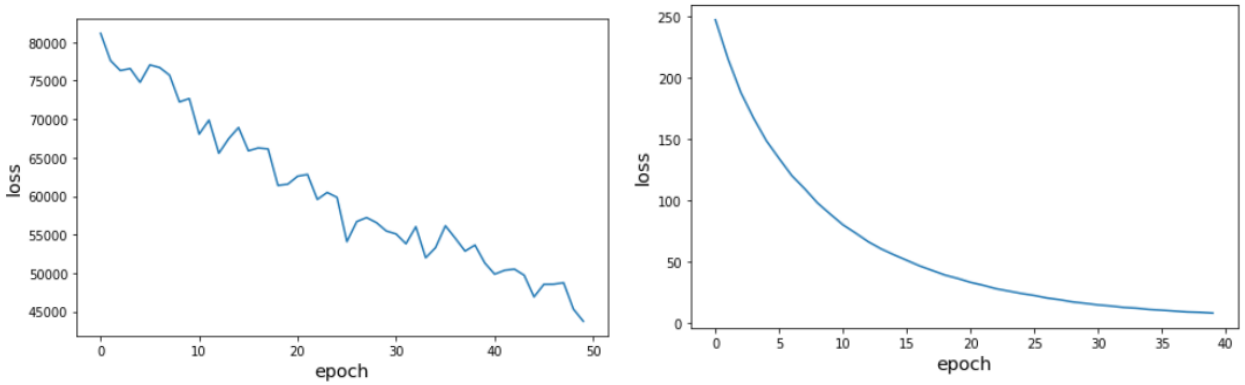
Figure 6: Loss progress for GCN and GraphSage during the epochs for non-normalized data

is built and confronted with the GCN results(how the loss function behave when creating the models through the epochs).

Algorithm 3: GNN model

```
generator = FullBatchNodeGenerator(graphWithFeatures, method="gcn")
gcn = GCN(
    layer_sizes=[16, 16],
    activations=["relu", "relu"],
    generator=generator,
    dropout=0.5
)
x_inp, x_out = gcn.in_out_tensors()
predictions = layers.Dense(controlsDK.shape[1],
                           kernel_initializer='normal')(x_out)
model = Model(inputs=x_inp , outputs=predictions)

model.compile(
    optimizer=optimizers.Adam(learning_rate=0.0001),
    loss=genes_loss_with_closure(edges=tensorEdgesTest),
    metrics=["acc"])

history = model.fit(
    train_gen,
    epochs=50,
    batch_size=controlsDK.shape[0],
    verbose=2,
    shuffle=False
)
```

As could be seen in figure 6 and in 7, the loss function for GCN decreases overall but mantains high values and fluctuates too much, so this model is not very good for the task of genes embeddings with the structure defined in this research.



(a) GCN (b) GraphSage

Figure 7: Loss progress for GCN and GraphSage during the epochs for normalized data

After fitting the model to create the embeddings that minimize the loss(for the defined epochs), we have a model that could be used to predict the genes embeddings, but for optimization and best choice, the next chapter 2.2 will use the embeddings created by the **GraphSage** model that creates better embeddings.

The loss function for GraphSage is almost the same as the one defined for GCN with a little alteration in the sub-method for the difference of genes embeddings

Algorithm 4: Loss function for GraphSage

```
@tf.function
def difference_of_edge_genes_embedding(edges,y_pred):
    gene1 = y_pred[edges[0]]
    gene2 = y_pred[edges[1]]
    return tf.sqrt(tf.reduce_sum(tf.square(gene1 - gene2)) + 1.0e-12)

def genes_loss_with_closure(edges):
    def loss_out(y_true, y_pred):
        loss = tf.constant(0, tf.float32)
        #partials = tf.map_fn(lambda edge: difference_of_edge_genes_embedding(edge,y_pr
                                                    tensorEdgesTest)
        for row in edges:
            loss = loss + difference_of_edge_genes_embedding(row,y_pred)
        return loss
    return loss_out
```

The considerations done for the loss in the previous sections are the same for this implementation in tensorflow.

The GNN model that uses GraphSage takes 10 nodes at depth 1(5 in-edges, 5 out-edges) and 4 nodes at depth 2(2 in-edges,2 out-edges), results on the model could differ if the number of nodes or layers changes, right now this is not the main task of this research and the results do not seem to change for different layers for the GraphSage model.

Algorithm 5: GNN model for GraphSage and fitting with previous loss function

```
batch_size = controlsDK.shape[0]
in_samples = [5, 2]
out_samples = [5, 2]
generator = DirectedGraphSAGENodeGenerator(directedGraph,
                                           batch_size,
                                           in_samples,
                                           out_samples)
x_inp, x_out = graphsage_model.in_out_tensors()
prediction = layers.Dense(units=train_subjects.shape[1],
                          activation="softmax")(x_out)
model = Model(inputs=x_inp , outputs=predictions)

model = Model(inputs=x_inp, outputs=prediction)
model.compile(
    optimizer=optimizers.Adam(learning_rate=0.005),
    loss=genes_loss_with_closure(edges=tensorEdgesTest),
    #loss = testLoss([0,1]),
    metrics=["acc"],
)

history = model.fit(
    train_gen, epochs=30,
    #validation_data=test_gen,
    batch_size=controlsDK.shape[0],
    verbose=2, shuffle=False
)
```

The number of epochs used to train the model is 30 but could be augmented to 50 because the loss continues to decrease monotonically.

As could be seen in figure 7, the loss function for GraphSage decreases overall and in a better way than the loss for the GCN model.

After the creation of the model, the prediction of the genes embedding can be done with the following lines of code:

Algorithm 6: Genes embedding prediction with the GraphSage model

```
all_mapper = generator.flow(controlsDK.index)
all_predictions = model.predict(all_mapper)
all_predictions = all_predictions.astype("float64")

genesEmbeddings = pd.DataFrame(all_predictions,
                               index =controlsDK.index,
                               columns =controlsDK.columns)
```

Where controlsDK are the genes with their identifiers as indexes and model is the GraphSage model(or the GCN).

An example of genes embeddings computed after the model fitting and prediction for all

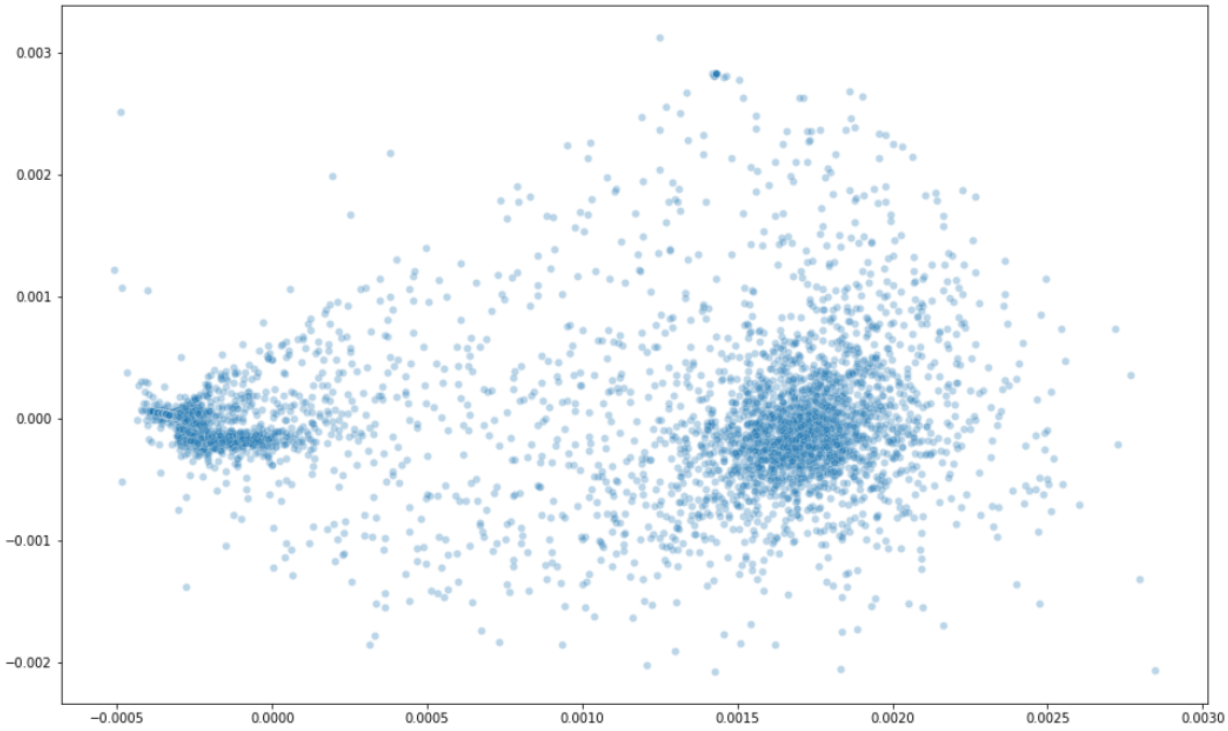


Figure 8: Visualization of genes embeddings in 2D

nodes-genes could be seen in 10, a visualization of the genes embeddings(with dimensionality reduction on the embeddings to be 2-3 dimensional data, PCA was used for this task) is provided in figure 8 and 9. Results for non-normalized data were really similar.

In the visualization, two clear clusters seem to form, these results raise the possibility and the idea of using these embeddings to control different data for possible discrepancies on the results (different clusters, no clusters at all, distant or shifted embeddings).

2.1.5 Further optimization and ideas

To further improve the model, **probability calibration** could be done to calibrate parameters and get a more flexible GNN to predict embeddings for other measures that shares the same structure behind.

The data at hand was not normalized, other experiments were done with normalized data (with different techniques) but, overall, the results were almost the same, therefore the data was left as it was because It was useless to normalize It(the GraphSage model implements a strategy that normalizes the results directly, as can be seen in figure 10, where the values are small enough).

There could be a problem with the overall model because the prediction of the edges aim at

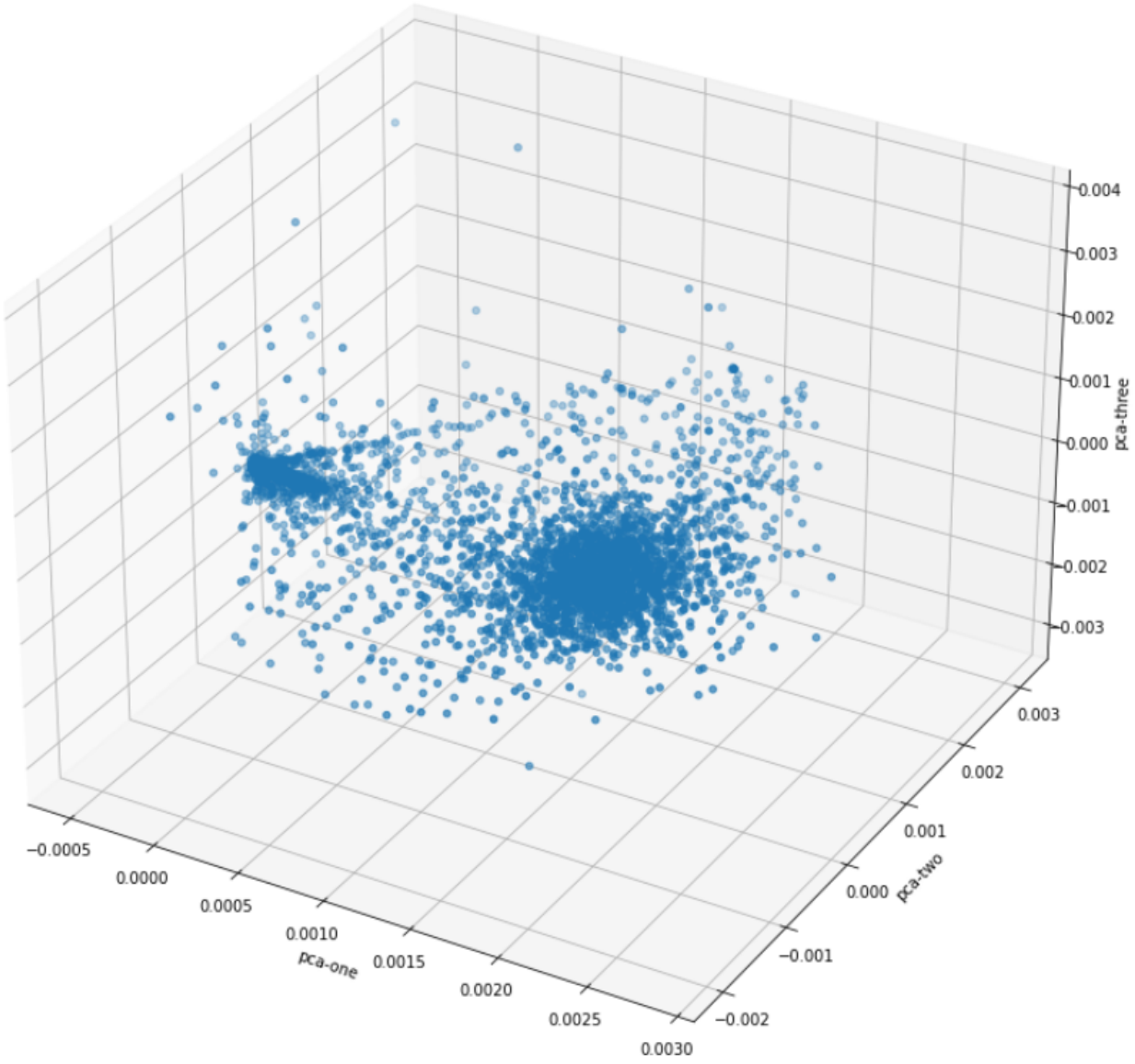


Figure 9: Visualization of genes embeddings in 3D

the minimization of the loss function defined in 4, but the models will probably underestimate the values because It will minimize the loss, so almost all genes embeddings will be close to 0 with little difference between them.

Another problem with the overall structure is that the relation between these genes embeddings reflects the relation in the original data(even with the GNN model), so if there is no relation(or linear/sub-linear relation within the data), the model will not work really well to estimate embeddings, because there is no way of knowing if two genes are really close or not. This problem will be seen in 2.2 and a discussion in 3 will be seen that takes into account all

	TCGA-BH-A0BC-11A-22R-A089-07	TCGA-BH-A0HA-11A-31R-A12P-07	TCGA-E2-A1LB-11A-22R-A144-07	TCGA-E2-A1LH-11A-22R-A14D-07	TCGA-E2-A1BC-11A-32R-A12P-07	TCGA-E9-A1NF-11A-73R-A14D-07	TCGA-E9-A1N9-11A-71R-A14D-07	TCGA-BH-A0H5-11A-62R-A115-07	TCGA-BH-A0DT-11A-12R-A12D-07	TCGA-A7-A0CH-11A-32R-A089-07	...
2	0.008032	0.009386	0.007723	0.009461	0.008655	0.008479	0.009695	0.008177	0.008907	0.008037	...
9	0.007986	0.009356	0.007811	0.009358	0.008553	0.008552	0.009701	0.008147	0.008892	0.007989	...
10	0.007910	0.009362	0.007758	0.009505	0.008528	0.008637	0.009653	0.008136	0.008889	0.007855	...
15	0.008079	0.009448	0.007750	0.009454	0.008537	0.008497	0.009703	0.008127	0.008908	0.008027	...
16	0.008074	0.009427	0.007745	0.009452	0.008551	0.008510	0.009707	0.008136	0.008903	0.008006	...
...	...	...	...	...	...	...	...	...	...	...	...

Figure 10: genes features

the possible problems with this research and with the data at hand.

2.2 Pathway embedding

As already introduced in 2.1.2 section and at the start of 2, the pathway embedding is done with a GCN-like formula that is 5, and the resulting model to predict this pathways embeddings is trained with the loss 6 that minimizes the distance between pathways with a big intersection of genes.

2.2.1 Gradient descent

To find the parameters of the model, Gradient Descent is used among all data and It is implemented in tensorflow [11].

The first thing to do is to prepare the data to compute the current pathway embedding, for this task firstly the genes are filtered through a hash table that takes the genes embeddings that belong to a pathway:

Algorithm 7: Pre-processing of the data

```
tfDatasetGenesExpressions = tf.data.Dataset.
    from_tensor_slices((genesEmbeddings.index,
                        genesEmbeddings.values))
normalizer_constant = genesEmbeddings.max().max()

table = tf.lookup.StaticHashTable(
    tf.lookup.KeyValueTensorInitializer(keys_tensor, vals_tensor),
    default_value=0)  # If index not in table, return 0.


def hash_table_filter(index, value):
    table_value = table.lookup(index)  # 1 if index in arr, else 0.
    index_in_arr =  tf.cast(table_value, tf.bool) # 1 -> True, 0 -> False
```

```
        return index_in_arr

    def filterdatasetOnIndex(ds):
        return ds.filter(hash_table_filter)

    filtered = tfDatasetGenesExpressions.apply(filterdatasetOnIndex)
    single = filtered.
        map(lambda index,value : value).
        reduce(np.float64(0.0),lambda x,y: x+y)/ len(listatest)
```

The next step is to get the partial vector that will be used for every computation of the final pathway embedding to speed up the process, this vector is the sum of all the genes in the pathway divided by the cardinality of the pathway, that is $\sum_{u \in P_i} \frac{z_u}{|P_i|}$ as seen in 5, this partial vector is shared among all steps of the gradient descent.

Algorithm 8: filtering and preparing the constant tensor to speed up computation

```
def getPartialVec(Nodes):
    listatest =  Nodes.split(";")
    listatest = list(map(int, listatest))

    keys_tensor = tf.constant(listatest,dtype=tf.int64)
    vals_tensor = tf.ones_like(keys_tensor)  # Ones will be casted to True.

    table = tf.lookup.StaticHashTable(
        tf.lookup.KeyValueTensorInitializer(keys_tensor, vals_tensor),
        default_value=0)  # If index not in table, return 0.

    def hash_table_filter(index, value):
        table_value = table.lookup(index)  # 1 if index in arr, else 0.
        index_in_arr =  tf.cast(table_value, tf.bool) # 1 -> True, 0 -> False
        return index_in_arr

    tfFiltered = tfDatasetGenesExpressions.filter(hash_table_filter)
    tfFiltered = tfFiltered.
        map(lambda index,value : value).
        reduce(np.float64(0.0),lambda x,y: x+y)/
        (len(listatest) * normalizer_constant)
    #even though data is already within a close range to 0
    return tfFiltered.numpy()


pathways["partialsVectors"] = pathways["nodes"].map(getPartialVec)
```

To compute the final embedding, the W tensor(firstly a diagonal matrix, after what is estimated by the gradient descent) is multiplied by the partial vector and the result is passed to a non-linear function(leaky-relu in this case) that transforms values of the vector point-wise. This function will also be used to compute the gradient descent to estimate the W matrix.

Algorithm 9: Function to compute final embedding

```
@tf.function
def sigmoid_log(x):
    return 1.0 / (1.0 + tf.math.exp(-tf.math.log(x)))

@tf.function( experimental_relax_shapes=True)
def getPathwayEmbeddingEnhancedFinal(partialsVec_tensor,Wmat):
    W_tensor = Wmat #tf.constant(Wmat,dtype=tf.float64)
    #Wmat could be already a tensor
    trasformedPartialVec = W_tensor @ partialsVec_tensor
    finalVec = tf.map_fn(lambda values:
                tf.map_fn(tf.nn.leaky_relu,values),trasformedPartialVec)
    return finalVec

npPartial = np.stack(pathways["partialsVectors"]).transpose()
partialsVectors_tensor = tf.constant(npPartial)
finalVectors_tensors = getPathwayEmbeddingEnhancedFinal(
                            partialsVectors_tensor,W_tensor)
pathways["pathwayEmbeddings"]= finalVectors_tensors.numpy().
                                    transpose().
                                    tolist()

loss = tf.Variable(0.0,dtype="float64")
```

The following code implements the loss function 6

Algorithm 10: Loss function code

```
@tf.function( experimental_relax_shapes=True)
def getLoss(pathwaysEmbeddingTensor,loss):
    for zPi in pathwaysEmbeddingTensor:
        #print(zPi.get_shape())
        for zPj in pathwaysEmbeddingTensor:
            loss = loss + tf.sqrt(tf.
                        reduce_sum(tf.square(zPi - zPj)) + 1.0e-12)
            #tf.norm is unstable
    return loss
```

Lastly, the computation for gradient descent is done using the tensorflow framework as defined in [11] and with the previous auxiliary function defined.

Algorithm 11: Gradient descent computation

```
W_numpy = np.diag(np.full(len(pathways["partialsVectors"][0]),0.01))
W = tf.Variable(W_numpy,dtype=tf.float64,name = "W")
npPartial = np.stack(pathways["partialsVectors"]).transpose()
partialsVectors_tensor = tf.constant(npPartial)
loss = tf.Variable(0.0,dtype="float64")

with tf.GradientTape(persistent=True) as tape:
    loss.assign(0.0)
```

```
[0.0008614577877624723,
 0.0008686239967118825,
 0.000866007645638882,
 0.0007685363010010587,
 0.0008507110699452298,
 0.0008636097696191665,
 0.0008286009022825203,
 0.0008728383001085555,
 0.0006490175548810436,
 0.0006597512180694226,
 0.0007910211448600898,
 0.0007013890958183493,
 0.0006712640547292728,
 0.0008150293070178545,
 0.0008540394796349374,
 0.0007055741332311652,
 0.0008515530087366063,
 0.0007283004046750776,
 0.0006510316864320179,
 0.0008609924030235831,
 0.0007979563413553789,
 0.0006536015274313374,
 0.0007593721081703536,
 0.0006471475681545778,
 0.0006582335927214391,
 0.0007382132751783368,
 0.0007219412298010148,
 0.000842446421854718,
 0.000834358746647123,
 0.0007695325527519575,
 0.0006218636557521374,
 0.0006931608877763137,
 0.0007951531978124529,
 0.0007727570658753182,
```

(a) pathway 0

```
[0.000862344620053276,
 0.0008693077725082905,
 0.0008659459287169518,
 0.0007684823504171725,
 0.0008509693550328639,
 0.0008633573847177387,
 0.0008290827057839815,
 0.0008727576004877153,
 0.0006497445174104496,
 0.0006591501694965712,
 0.0007907066418192503,
 0.0007015052771097436,
 0.0006708259042405854,
 0.0008150163671946796,
 0.0008534005290053353,
 0.0007059135389512052,
 0.0008510236696557777,
 0.0007277134255651828,
 0.0006523400307346017,
 0.0008600723855202426,
 0.0007981517414979898,
 0.0006547041972567313,
 0.00075947537769453 1,
 0.00064624767985793 1,
 0.0006586532562239348,
 0.0007372568381090939,
 0.0007228697164184514,
 0.0008424020281989485,
 0.0008340441289727175,
 0.0007705685529173602,
 0.0006225223220641951,
 0.000693626162012576,
 0.0007954098399910949,
 0.0007720758120173652,
 0.0008612483529256442,
 0.0007504642063884482,
```

(b) pathway 1

```
[0.0008612605856481901,
 0.0008698974115773472,
 0.000865862670968907,
 0.0007674591600794479,
 0.0008503424231990908,
 0.000863452085425883 6,
 0.0008294609878385078,
 0.0008749620742118636,
 0.0006488463234121104,
 0.0006590315817609005,
 0.0007908040323713211,
 0.0007008807044511133,
 0.0006709312642137016,
 0.000815223654050638,
 0.0008535137845804716,
 0.0007047121090382639,
 0.0008522435462352216,
 0.0007275442930329726,
 0.0006512616364063945,
 0.000859152703612691,
 0.0007998534308089626,
 0.0006540780799602825,
 0.0007587641435229528,
 0.0006462796687762593,
 0.000657704105862711,
 0.0007390311754528969,
 0.0007233038804861491,
 0.0008406509256460803,
 0.0008328485732339739,
 0.000769396034781745,
 0.0006206717301731402,
 0.0006938248696821179,
 0.000795505950339252,
 0.0007729057704392543,
 0.0008618716352681657,
```

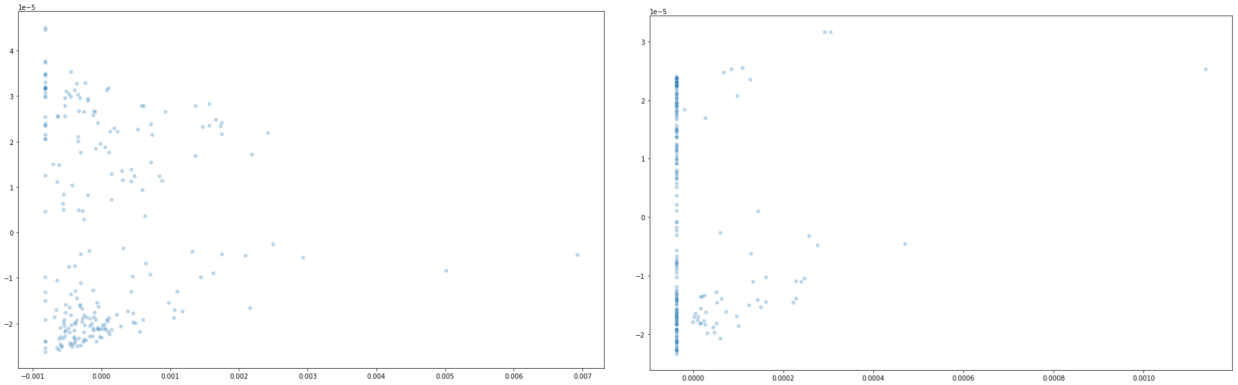
(c) pathway 29

Figure 11: Sample pathway embeddings

```
        tape.watch(W)
        finalVectors_tensors = getPathwayEmbeddingEnhancedFinal(
                                            partialsVectors_tensor,W)
        loss = getLoss(tf.transpose(finalVectors_tensors),loss)

    learning_rate = 0.0001

    for i in range(0,10):
        dl_dw = tape.gradient(loss, W)
        W.assign_sub(learning_rate * dl_dw)
```

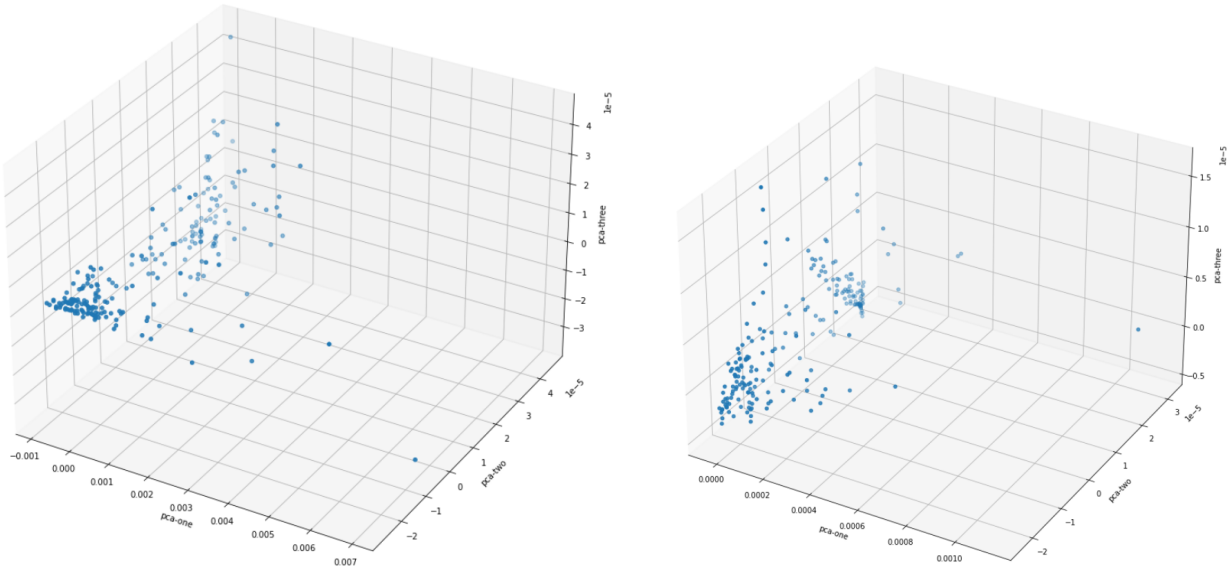
A the end of the gradient descent algorithm, the model is trained and ready to predict pathways embeddings.

An example of the pathway embeddings can be seen in figure 11, the embeddings are really small and seem to be close to one another, and by analysing these embeddings, the results lead to the conclusion that all of the embeddings are really close to one another(small variance and the values differ from one another at the 4-5th decimal digit and so on), this problem



(a) normalized data (b) non-normalized data

Figure 12: pathway embeddings visualized 2D



(a) normalized data (b) non-normalized data

Figure 13: pathway embeddings visualized 3D

is probably related to what was introduced in the previous section about the consequence of non-linearity of the data and the optimization of the 4 to find the GNN model that minimizes It, that has propagated Its effects upon the loss of the pathways. However, the overall model seem to work and the results that will be seen at the end of this chapter are quite intriguing and interesting for further research.

In figure 12 and 13, the pathways embeddings were treated with dimensionality reduction to help visualize them(PCA was used, also T-SNE but the results were really similar). Two clusters could be seen among the pathways along with a lot of outliers. For the non-normalized

data, the clusters are in the same places as the one for the normalized data, but the density seems lower in the left cluster for the non-normalized data, while it seems lower in the second cluster for the normalized data.

3 Conclusions and further research on the topic

The objective of this research was to find embeddings for genes and pathways, both of these tasks were done and analyzed during this research with the help of some external libraries (stellargraph) and python frameworks to compute gradients and models for neural networks (Keras and tensorflow).

For the embeddings of genes, the GraphSage model was used with really good results with the loss used, also a GCN model was used but with poor results, so the model that has done the embedding was the GraphSage model.

For the embeddings of pathways, a GCN-like model was used to compute the embeddings, and the model was trained with Gradient Descent (the possibility of using a Stochastic Gradient Descent or batch-based optimization techniques should not be possible because the loss function requires that all the nodes for a specifica set, that is genes in a pathway, should be in the same batch, if another framework is used to resolve the optimization problem, like a user defined procedure in Spark or Map-Reduce, these problems will not be present).

The research has demonstrated the power of the GNN and embeddings for non-euclidean data, further improvements in the field are welcome, especially in the field of the bio-sciences.

A future work will see how different model created from different genes features (obtained from the expressions of different patients affected by some pathology) will behave in the embedding space by confronting the density of the different embeddings space locations or the clusters that they create.

References

[1] Wikipedia. Gene regulatory pathways. "https://en.wikipedia.org/wiki/Gene_regulatory_network".

[2] CSIRO's Data61. Stellargraph machine learning library. https://github.com/stellargraph/stellargraph, 2018.

[3] Spektral. Spectral documentation. "https://graphneural.network".

[4] Thomas Kipf. Graph convolutional neural network. "https://tkipf.github.io/graph-convolutional-networks/".

[5] William L. Hamilton, Rex Ying, and Jure Leskovec. Graph sage snap. "https://snap.stanford.edu/graphsage/".

[6] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[7] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[8] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020.

[9] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks, 2017.

[10] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[11] Tensorflow. basic training loops. "https://www.tensorflow.org/guide/basic_training_loops".