3
Unsupervised Graph Learning

Unsupervised machine learning refers to the subset of machine learning algorithms that do not exploit any target information during training. Instead, they work on their own to find clusters, discover patterns, detect anomalies, and solve many other problems for which there is no teacher and no correct answer known *a priori*.

As per many other machine learning algorithms, unsupervised models have found large applications in the graph representation learning domain. Indeed, they represent an extremely useful tool for solving various downstream tasks, such as node classification and community detection, among others.

In this chapter, an overview of recent unsupervised graph embedding methods will be provided. Given a graph, the goal of these techniques is to automatically learn a latent representation of it, in which the key structural components are somehow preserved.

The following topics will be covered in this chapter:

- The unsupervised graph embedding roadmap
- Shallow embedding methods
- Autoencoders
- Graph neural networks

Technical requirements

We will be using Jupyter notebooks with Python 3.9 for all of our exercises. The following is a list of the Python libraries that need to be installed for this chapter using `pip`. For example, run `pip install networkx==2.5` on the command line, and so on:

```
Jupyter==1.0.0
networkx==2.5
matplotlib==3.2.2
karateclub==1.0.19
node2vec==0.3.3
tensorflow==2.4.0
scikit-learn==0.24.0
git+https://github.com/palash1992/GEM.git
git+https://github.com/stellargraph/stellargraph.git
```

In the rest of this book, if not clearly stated, we will refer to the Python commands `import networkx` as `nx`.

All the code files relevant to this chapter are available at `https://github.com/PacktPublishing/Graph-Machine-Learning/tree/main/Chapter03`.

The unsupervised graph embedding roadmap

Graphs are complex mathematical structures defined in a non-Euclidean space. Roughly speaking, this means that it is not always easy to define what is close to what; it might also be hard to say what *close* even means. Imagine a social network graph: two users can be respectively connected and yet share very different features—one might be interested in fashion and clothes, while the other might be interested in sports and videogames. Can we consider them as "close"?

For this reason, unsupervised machine learning algorithms have found large applications in graph analysis. Unsupervised machine learning is the class of machine learning algorithms that can be trained without the need for manually annotated data. Most of those models indeed make use of only information in the adjacency matrix and the node features, without any knowledge of the downstream machine learning task.

How is this possible? One of the most used solutions is to learn embeddings that preserve the graph structure. The learned representation is usually optimized so that it can be used to reconstruct the pair-wise node similarity, for example, the **adjacency matrix**. These techniques bring an important feature: the learned representation can encode latent relationships among nodes or graphs, allowing us to discover hidden and complex novel patterns.

Many algorithms have been developed in relation to unsupervised graph machine learning techniques. However, as previously reported by different scientific papers (`https://arxiv.org/abs/2005.03675`), those algorithms can be grouped into macro-groups: shallow embedding methods, autoencoders, and **Graph Neural Networks** (**GNNs**), as graphically described in the following chart:

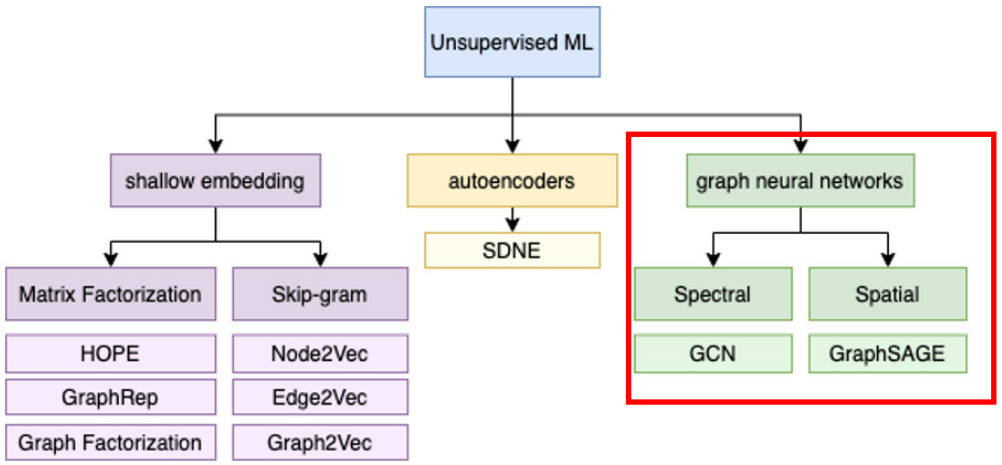

Figure 3.1 – The hierarchical structure of the different unsupervised embedding algorithms described in this book

In the following sections, you will learn the main principles behind each group of algorithms. We will try to provide the idea behind the most well-known algorithms in the field as well as how they can be used for solving real problems.

Shallow embedding methods

As already introduced in *Chapter 2*, *Graph Machine Learning*, with shallow embedding methods, we identify a set of algorithms that are able to learn and return only the embedding values for the learned input data.

In this section, we will explain in detail some of those algorithms. Moreover, we will enrich the descriptions by providing several examples of how to use those algorithms in Python. For all the algorithms described in this section, we will use the implementation provided in the following libraries: **Graph Embedding Methods (GEM)**, **Node to Vector (Node2Vec)**, and Karate Club.

Matrix factorization

Matrix factorization is a general decomposition technique widely used in different domains. A consistent number of graph embedding algorithms use this technique in order to compute the node embedding of a graph.

We will start by providing a general introduction to the matrix factorization problem. After the introduction of the basic principles, we will describe two algorithms, namely **Graph Factorization (GF)** and **Higher-Order Proximity Preserved Embedding (HOPE)**, which use matrix factorization to build the node embedding of a graph.

Let $W \in \mathbb{R}^{m \times n}$ be the input data. Matrix factorization decomposes $W \approx V \times H$ with $V \in \mathbb{R}^{m \times d}$ and $H \in \mathbb{R}^{d \times n}$ called the **source** and **abundance** matrix, respectively, and d is the number of dimensions of the generated embedding space. The matrix factorization algorithm learns the V and H matrices by minimizing a loss function that can change according to the specific problem we want to solve. In its general formulation, the loss function is defined by computing the reconstruction error using the Frobenius norm as $\|W - V \times H\|_F^2$.

Generally speaking, all the unsupervised embedding algorithms based on matrix factorization use the same principle. They all factorize an input graph expressed as a matrix in different components. The main difference between each method lies in the loss function used during the optimization process. Indeed, different loss functions allow creating an embedding space that emphasizes specific properties of the input graph.

Graph factorization

The GF algorithm was one of the first models to reach good computational performance in order to perform the node embedding of a given graph. By following the principle of matrix factorization that we previously described, the GF algorithm factorizes the adjacency matrix of a given graph.

Formally, let $G = (V, E)$ be the graph we want to compute the node embedding with and let $A \in \mathbb{R}^{|V| \times |V|}$ be its adjacency matrix. The loss function (L) used in this matrix factorization problem is as follows:

$$L = \frac{1}{2} \sum_{(i,j) \in E} (A_{i,j} - Y_{i,:} Y_{j,:}{}^{T})^2 + \frac{\lambda}{2} \sum_{i} \|Y_{i,:}\|^2$$

In the preceding equation, $(i, j) \in E$ represents one of the edges in G while $Y \in \mathbb{R}^{|V| \times d}$ is the matrix containing the d-dimensional embedding. Each row of the matrix represents the embedding of a given node. Moreover, a regularization term (λ) of the embedding matrix is used to ensure that the problem remains well-posed even in the absence of sufficient data.

The loss function used in this method was mainly designed to improve GF performances and scalability. Indeed, the solution generated by this method could be noisy. Moreover, it should be noted, by looking at its matrix factorization formulation, that GF performs a strong symmetric factorization. This property is particularly suitable for undirected graphs, where the adjacency matrix is symmetric, but could be a potential limitation for undirected graphs.

In the following code, we will show how to perform the node embedding of a given `networkx` graph using Python and the GEM library:

```
import networkx as nx
from gem.embedding.gf import GraphFactorization
G = nx.barbell_graph(m1=10, m2=4)
gf = GraphFactorization(d=2, data_set=None, max_iter=10000,
eta=1*10**-4, regu=1.0)
gf.learn_embedding(G)
embeddings = gf.get_embedding()
```

In the preceding example, the following have been done:

1. `networkx` is used to generate a **barbell graph** (G) used as input for the GF factorization algorithm.
2. The `GraphFactorization` class is used to generate a `d=2`-dimensional embedding space.
3. The computation of the node embeddings of the input graph is performed using `gf.learn_embedding(G)`.
4. The computed embeddings are extracted by calling the `gf.get_embedding()` method.

The results of the previous code are shown in the following graph:

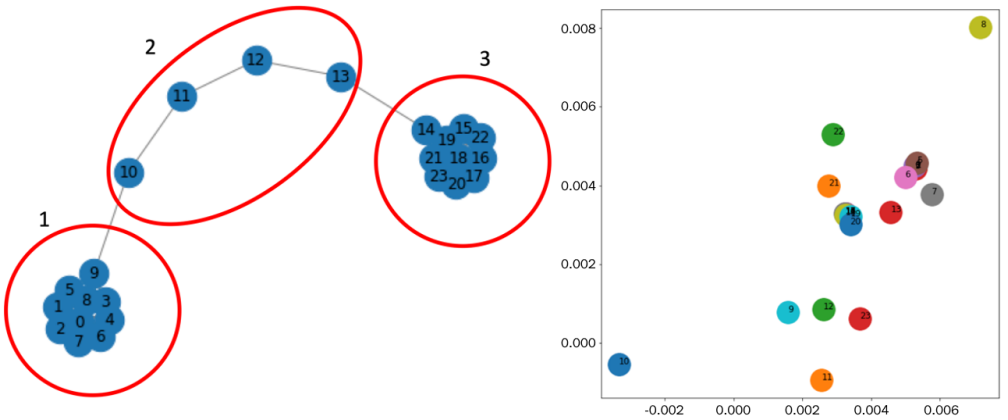

Figure 3.2 – Application of the GF algorithm to a graph (left) to generate the embedding vector of its nodes (right)

From *Figure 3.2*, it is possible to see how nodes belonging to groups 1 and 3 are mapped together in the same region of space. Those points are separated by the nodes belonging to group 2. This mapping allows us to well separate groups 1 and 3 from group 2. Unfortunately, there is no clear separation between groups 1 and 3.

Higher-order proximity preserved embedding

HOPE is another graph embedding technique based on the matrix factorization principle. This method allows preserving higher-order proximity and does not force its embeddings to have any symmetric properties. Before starting to describe the method, let's understand what first-order proximity and high-order proximity mean:

- **First-order proximity**: Given a graph, $G = (V, E)$, where the edges have a weight, w_{ij}, for each vertex pair (v_i, v_j), we say they have a first-order proximity equal to w_{ij} if the edge $(v_i, v_j) \in E$. Otherwise, the first-order proximity between the two nodes is 0.
- **Second- and high-order proximity**: With the second-order proximity, we can capture the two-step relations between each pair of vertices. For each vertex pair (v_i, v_j), we can see the second-order proximity as a two-step transition from v_i to v_j. High-order proximity generalizes this concept and allows us to capture a more global structure. As a consequence, high-order proximity can be viewed as a k-step $(k \geq 3)$ transition from v_i to v_j.

Given the definition of proximity, we can now describe the HOPE method. Formally, let $G = (V, E)$ be the graph we want to compute the embedding for and let $A \in \mathbb{R}^{|V|\times|V|}$ be its adjacency matrix. The loss function (L) used by this problem is as follows:

$$L = \|S - Y_s \times {Y_t}^T\|_F^2$$

In the preceding equation, $S \in \mathbb{R}^{|V|\times|V|}$ is a similarity matrix generated from graph G and $Y_s \in \mathbb{R}^{|V|\times d}$ and $Y_t \in \mathbb{R}^{|V|\times d}$ are two embedding matrices representing a d-dimensional embedding space. In more detail, Y_s represents the source embedding and Y_t represents the target embedding.

HOPE uses those two matrices in order to capture asymmetric proximity in directed networks where the direction from a source node and a target node is present. The final embedding matrix, Y, is obtained by simply concatenating, column-wise, the Y_s and Y_t matrices. Due to this operation, the final embedding space generated by HOPE will have $2 * d$ dimensions.

As we already stated, the S matrix is a similarity matrix obtained from the original graph, G. The goal of S is to obtain high-order proximity information. Formally, it is computed as $S = M_g \cdot M_l$, where M_g and M_l are both polynomials of matrices.

In its original formulation, the authors of HOPE suggested different ways to compute M_g and M_l. Here we report a common and easy method to compute those matrices, **Adamic-Adar** (**AA**). In this formulation, $M_g = I$ (the identity matrix) while $M_l = A \cdot D \cdot A$, where D is a diagonal matrix computed as $D_{ij} = 1/(\sum (A_{ij} + A_{ji}))$. Other formulations to compute M_g and M_l are the **Katz Index**, **Rooted PageRank** (**RPR**), and **Common Neighbors** (**CN**).

In the following code, we will show how to perform the node embedding of a given `networkx` graph using Python and the GEM library:

```
import networkx as nx
from gem.embedding.hope import HOPE
G = nx.barbell_graph(m1=10, m2=4)
gf = HOPE(d=4, beta=0.01)
gf.learn_embedding(G)
embeddings = gf.get_embedding()
```

The preceding code is similar to the one used for GF. The only difference is in the class initialization since here we use `HOPE`. According to the implementation provided by GEM, the `d` parameter, representing the dimension of the embedding space, will define the number of columns of the final embedding matrix, Y, obtained after the column-wise concatenation of Y_s and Y_t.

As a consequence, the number of columns of Y_s and Y_t is defined by the floor division (the `//` operator in Python) of the value assigned to `d`. The results of the code are shown in the following graph:

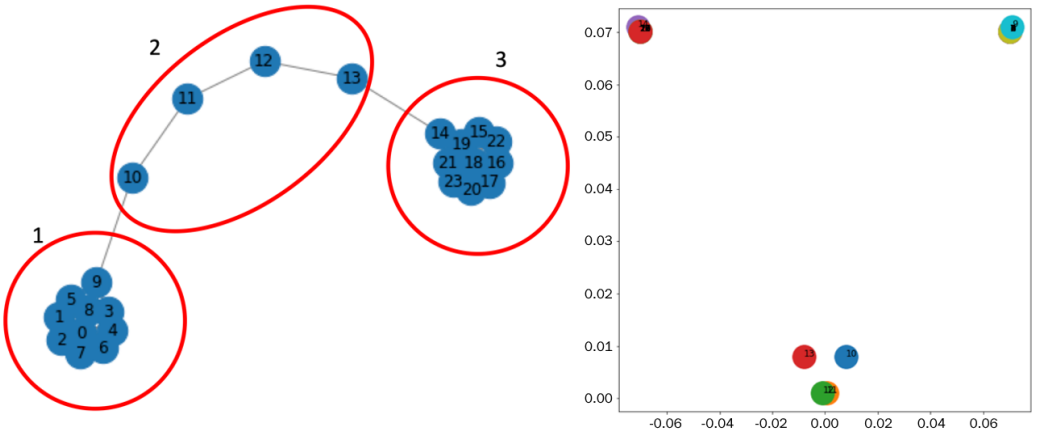

Figure 3.3 – Application of the HOPE algorithm to a graph (left) to generate the embedding vector of its nodes (right)

In this case, the graph is undirected and thus there is no difference between the source and target nodes. *Figure 3.3* shows the first two dimensions of the `embeddings` matrix representing Y_s. It is possible to see how the embedding space generated by HOPE provides, in this case, a better separation of the different nodes.

Graph representation with global structure information

Graph representation with global structure information (GraphRep), such as HOPE, allows us to preserve higher-order proximity without forcing its embeddings to have symmetric properties. Formally, let $G = (V, E)$ be the graph we want to compute the node embeddings for and let $A \in \mathbb{R}^{|V| \times |V|}$ be its adjacency matrix. The loss function (L) used by this problem is as follows:

$$L_k = \| X^k - Y_s^k \times {Y_t^k}^T \|_F^2 \; 1 \leq k \leq K$$

In the preceding equation, $X^k \in \mathbb{R}^{|V|\times|V|}$ is a matrix generated from graph *G* in order to get the *k*th order of proximity between nodes.

$Y_s^k \in \mathbb{R}^{|V|\times d}$ and $Y_t^k \in \mathbb{R}^{|V|\times d}$ are two embedding matrices representing a *d*-dimensional embedding space of the *k*th order of proximity for the source and target nodes, respectively.

The X^k matrix is computed according to the following equation: $X^k = \prod_k (D^{-1}A)$.
Here, D is a diagonal matrix known as the **degree matrix** computed using the following equation:

$$D_{ij} = \begin{cases} \sum_p A_{ip}, & i = j \\ 0, & i \neq j \end{cases}$$

$X^1 = D^{-1}A$ represents the (one-step) probability transition matrix, where X_{ij}^1 is the probability of a transition from $v_i$to vertex v_j within one step. In general, for a generic value of *k*, X_{ij}^k represents the probability of a transition from $v_i$to vertex v_j within *k* steps.

For each order of proximity, *k*, an independent optimization problem is fitted. All the *k* embedding matrices generated are then column-wise concatenated to get the final source embedding matrices.

In the following code, we will show how to perform the node embedding of a given `networkx` graph using Python and the `karateclub` library:

```
import networkx as nx
from karateclub.node_embedding.neighbourhood.grarep import
GraRep
G = nx.barbell_graph(m1=10, m2=4)
gr = GraRep(dimensions=2, order=3)
gr.fit(G)
embeddings = gr.get_embedding()
```

We initialize the `GraRep` class from the `karateclub` library. In this implementation, the `dimension` parameter represents the dimension of the embedding space, while the `order` parameter defines the maximum number of orders of proximity between nodes. The number of columns of the final embedding matrix (stored, in the example, in the `embeddings` variable) is `dimension*order`, since, as we said, for each proximity order an embedding is computed and concatenated in the final embedding matrix.

To specify, since two dimensions are computed in the example, `embeddings[:,:2]` represents the embedding obtained for k=1, `embeddings[:,2:4]` for k=2, and `embeddings[:,4:]` for k=3. The results of the code are shown in the following graph:

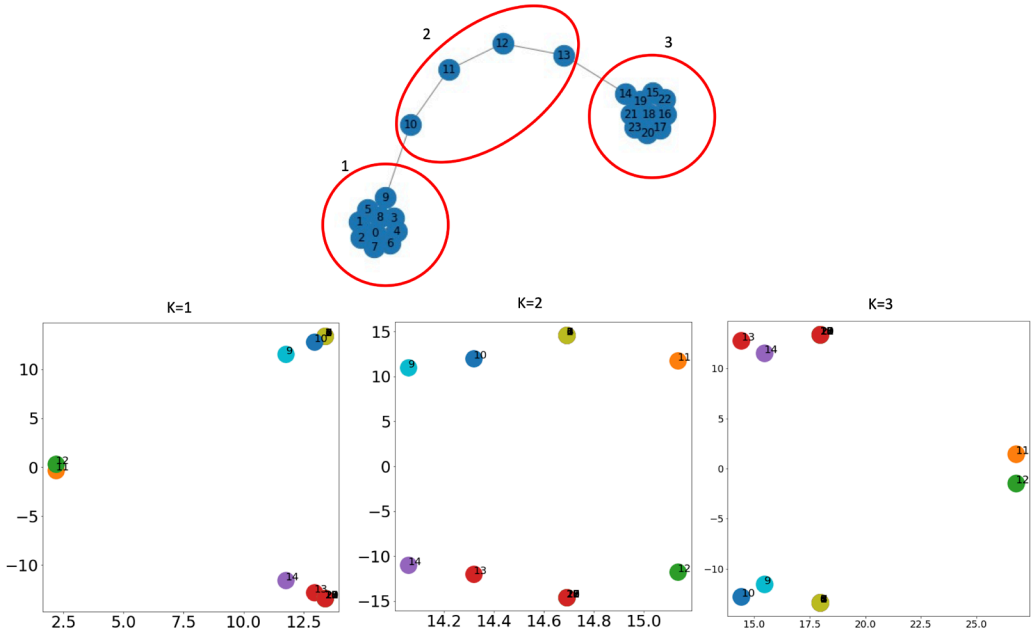

Figure 3.4 – Application of the GraphRep algorithm to a graph (top) to generate the embedding vector of its nodes (bottom) for different values of k

From the preceding graph, it is easy to see how different orders of proximity allow us to get different embeddings. Since the input graph is quite simple, in this case, already with k=1, a well-separated embedding space is obtained. To specify, the nodes belonging to groups 1 and 3 in all the proximity orders have the same embedding values (they are overlapping in the scatter plot).

In this section, we described some matrix factorization methods for unsupervised graph embedding. In the next section, we will introduce a different way to perform unsupervised graph embedding using skip-gram models.

Skip-gram

In this section, we will provide a quick description of the skip-gram model. Since it is widely used by different embedding algorithms, a high-level description is needed to better understand the different methods. Before going deep into a detailed description, we will first give a brief overview.

The skip-gram model is a simple neural network with one hidden layer trained in order to predict the probability of a given word being present when an input word is present. The neural network is trained by building the training data using a text corpus as a reference. This process is described in the following chart:

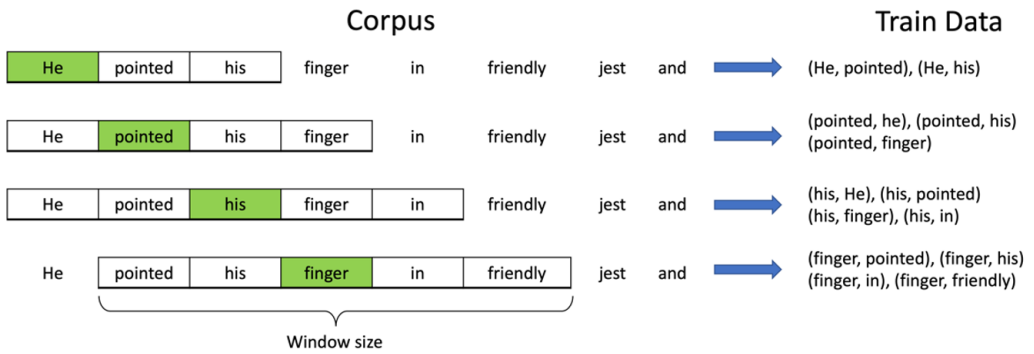

Figure 3.5 – Example of the generation of training data from a given corpus. In the filled boxes, the target word. In the dash boxes, the context words identified by a window size of length 2

The example described in *Figure 3.5* shows how the algorithm to generate the training data works. A *target* word is selected and a rolling window of fixed size *w* is built around that word. The words inside the rolling windows are known as *context* words. Multiple pairs of *(target word, context word)* are then built according to the words inside the rolling window.

Once the training data is generated from the whole corpus, the skip-gram model is trained to predict the probability of a word being a context word for the given target. During its training, the neural network learns a compact representation of the input words. This is why the skip-gram model is also known as **Word to Vector** (**Word2Vec**).

The structure of the neural network representing the skip-gram model is described in the following chart:

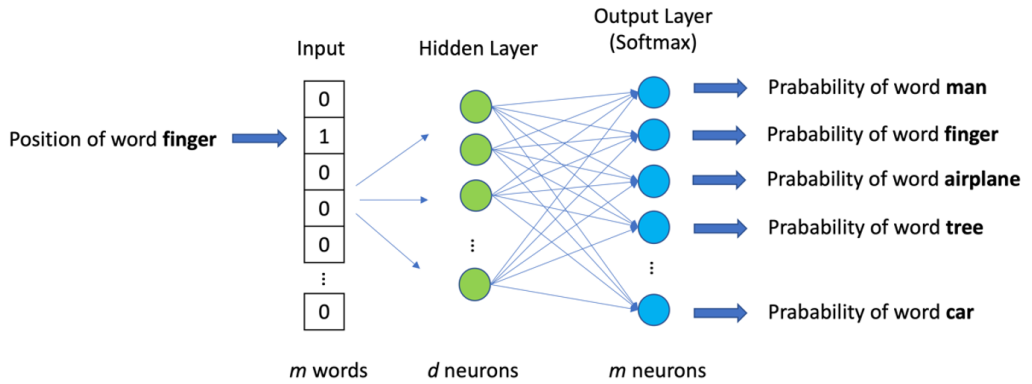

Figure 3.6 – Structure of the neural network of the skip-gram model. The number of d neurons in the hidden layer represents the final size of the embedding space

The input of the neural network is a binary vector of size *m*. Each element of the vector represents a word in the dictionary of the language we want to embed the words in. When, during the training process, a *(target word, context word)* pair is given, the input array will have 0 in all its entries with the exception of the entry representing the "target" word, which will be equal to 1. The hidden layer has *d* neurons. The hidden layer will learn the embedding representation of each word, creating a *d*-dimensional embedding space.

Finally, the output layer of the neural network is a dense layer of *m* neurons (the same size as the input vector) with a *softmax* activation function. Each neuron represents a word of the dictionary. The value assigned by the neuron corresponds to the probability of that word being "related" to the input word. Since softmax can be hard to compute when the size of *m* increases, a *hierarchical softmax* approach is always used.

The final goal of the skip-gram model is not to actually learn the task we previously described but to build a compact *d*-dimensional representation of the input words. Thanks to this representation, it is possible to easily extract an embedding space for the words using the weight of the hidden layer. Another common approach to creating a skip-gram model, which will be not described here, is *context-based*: **Continuous Bag-of-Words** (**CBOW**).

Since the basic concepts behind the skip-gram model have been introduced, we can start to describe a series of unsupervised graph embedding algorithms built upon this model. Generally speaking, all the unsupervised embedding algorithms based on the skip-gram model use the same principle.

Starting from an input graph, they extract from it a set of walks. Those walks can be seen as a text corpus where each node represents a word. Two words (representing nodes) are near each other in the text if they are connected by an edge in a walk. The main difference between each method lies in the way those walks are computed. Indeed, as we will see, different walk generation algorithms can emphasize particular local or global structures of the graph.

DeepWalk

The DeepWalk algorithm generates the node embedding of a given graph using the skip-gram model. In order to provide a better explanation of this model, we need to introduce the concept of **random walks**.

Formally, let G be a graph and let v_i be a vertex selected as the starting point. We select a neighbor of v_i at random and we move toward it. From this point, we randomly select another point to move. This process is repeated t times. The random sequence of t vertices selected in this way is a random walk of length t. It is worth mentioning that the algorithm used to generate the random walks does not impose any constraint on how they are built. As a consequence, there is no guarantee that the local neighborhood of the node is well preserved.

Using the notion of random walk, the DeepWalk algorithm generates a random walk of a size of at most *t* for each node. Those random walks will be given as input to the skip-gram model. The embedding generated using skip-gram will be used as the final node embedding. In the following figure (*Figure 3.7*), we can see a step-by-step graphical representation of the algorithm:

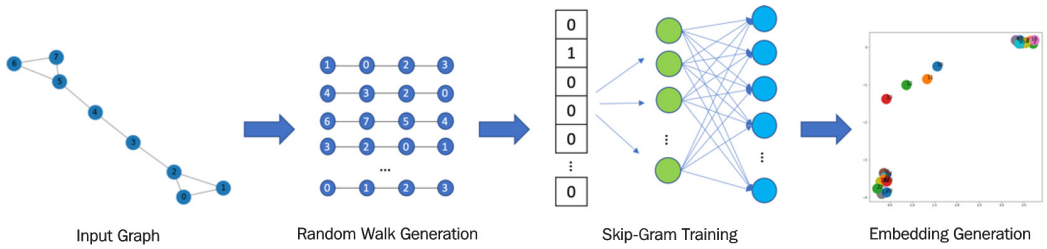

Figure 3.7 – All the steps used by the DeepWalk algorithm to generate the node embedding of a given graph

Here is a step-by-step explanation of the algorithm graphically described in the preceding chart:

1. **Random Walk Generation**: For each node of input graph *G*, a set of γ random walks with a fixed maximum length (*t*) is computed. It should be noted that the length *t* is an upper bound. There are no constraints forcing all the paths to have the same length.
2. **Skip-Gram Training**: Using all the random walks generated in the previous step, a skip-gram model is trained. As we described earlier, the skip-gram model works on words and sentences. When a graph is given as input to the skip-gram model, as visible in *Figure 3.7*, a graph can be seen as an input text corpus, while a single node of the graph can be seen as a word of the corpus.

   A random walk can be seen as a sequence of words (a sentence). The skip-gram is then trained using the "fake" sentences generated by the nodes in the random walk. The parameters for the skip-gram model previously described (window size, *w*, and embed size, *d*) are used in this step.
3. **Embedding Generation**: The information contained in the hidden layers of the trained skip-gram model is used in order to extract the embedding of each node.

In the following code, we will show how to perform the node embedding of a given `networkx` graph using Python and the `karateclub` library:

```
import networkx as nx
from karateclub.node_embedding.neighbourhood.deepwalk import
DeepWalk
G = nx.barbell_graph(m1=10, m2=4)
dw = DeepWalk(dimensions=2)
dw.fit(G)
embeddings = dw.get_embedding()
```

The code is quite simple. We initialize the `DeepWalk` class from the `karateclub` library. In this implementation, the `dimensions` parameter represents the dimension of the embedding space. Other parameters worth mentioning that the `DeepWalk` class accepts are as follows:

- `walk_number`: The number of random walks to generate for each node
- `walk_length`: The length of the generated random walks
- `window_size`: The window size parameter of the skip-gram model

Finally, the model is fitted on graph *G* using `dw.fit(G)` and the embeddings are extracted using `dw.get_embedding()`.

The results of the code are shown in the following figure:

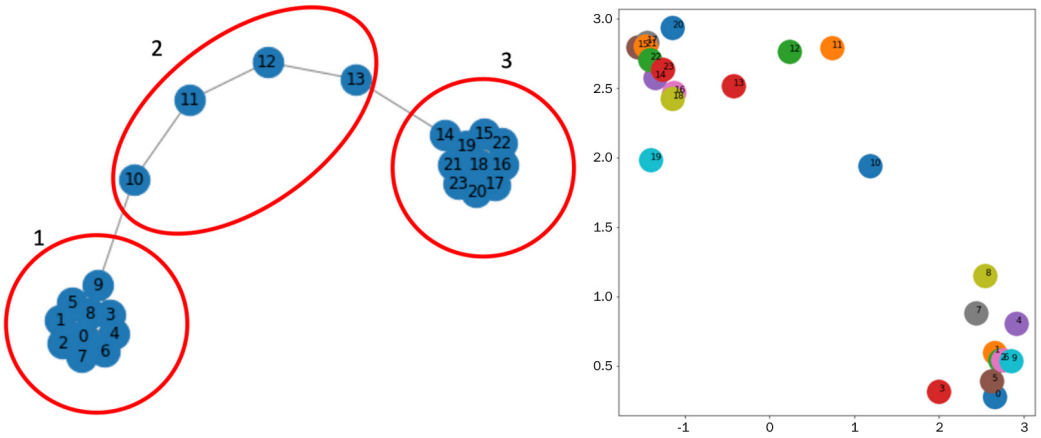

Figure 3.8 – Application of the DeepWalk algorithm to a graph (left) to generate the embedding vector of its nodes (right)

From the previous graph, we can see how DeepWalk is able to separate region 1 from region 3. Those two groups are contaminated by the nodes belonging to region 2. Indeed, for those nodes, a clear distinction is not visible in the embedding space.

Node2Vec

The **Node2Vec** algorithm can be seen as an extension of DeepWalk. Indeed, as with DeepWalk, Node2Vec also generates a set of random walks used as input to a skip-gram model. Once trained, the hidden layers of the skip-gram model are used to generate the embedding of the node in the graph. The main difference between the two algorithms lies in the way the random walks are generated.

Indeed, if DeepWalk generates random walks without using any bias, in Node2Vec a new technique to generate biased random walks on the graph is introduced. The algorithm to generate the random walks combines graph exploration by merging **Breadth-First Search** (**BFS**) and **Depth-First Search** (**DFS**). The way those two algorithms are combined in the random walk's generation is regularized by two parameters, p and q. p defines the probability of a random walk getting back to the previous node, while q defines the probability that a random walk can pass through a previously unseen part of the graph.

Due to this combination, Node2Vec can preserve high-order proximities by preserving local structures in the graph as well as global community structures. This new method of random walk generation allows solving the limitation of DeepWalk preserving the local neighborhood properties of the node.

In the following code, we will show how to perform the node embedding of a given `networkx` graph using Python and the `node2vec` library:

```
import networkx as nx
from node2vec import Node2Vec
G = nx.barbell_graph(m1=10, m2=4)
draw_graph(G)
node2vec = Node2Vec(G, dimensions=2)
model = node2vec.fit(window=10)
embeddings = model.wv
```

Also, for Node2Vec, the code is straightforward. We initialize the `Node2Vec` class from the `node2vec` library. In this implementation, the `dimensions` parameter represents the dimension of the embedding space. The model is then fitted using `node2vec.fit(window=10)`. Finally, the embeddings are obtained using `model.wv`.

It should be noted that `model.wv` is an object of the `Word2VecKeyedVectors` class. In order to get the embedding vector of a specific node with `nodeid` as the ID, we can use the trained model, as follows: `model.wv[str(nodeId)]`. Other parameters worth mentioning that the `Node2Vec` class accepts are as follows:

- `num_walks`: The number of random walks to generate for each node
- `walk_length`: The length of the generated random walks
- `p, q`: The *p* and *q* parameters of the random walk's generation algorithm

The results of the code are shown in *Figure 3.9*:

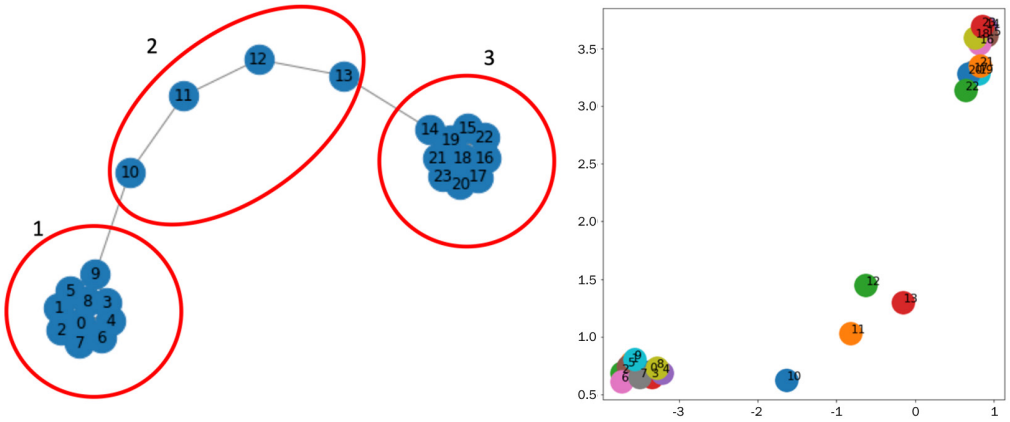

Figure 3.9 – Application of the Node2Vec algorithm to a graph (left) to generate the embedding vector of its nodes (right)

As is visible from *Figure 3.9*, Node2Vec allows us to obtain a better separation between nodes in the embedding space compared to DeepWalk. To specify, regions 1 and 3 are well clustered in two regions of space. Region 2 instead is well placed in the middle of the two groups without any overlap.

Edge2Vec

Contrary to the other embedding function, the **Edge to Vector** (**Edge2Vec**) algorithm generates the embedding space on edges, instead of nodes. This algorithm is a simple side effect of the embedding generated by using Node2Vec. The main idea is to use the node embedding of two adjacent nodes to perform some basic mathematical operations in order to extract the embedding of the edge connecting them.

Formally, let v_i and v_j be two adjacent nodes and let $f(v_i)$ and $f(v_j)$ be their embeddings computed with Node2Vec. The operators described in *Table 3.1* can be used in order to compute the embedding of their edge:

Operator	Equation	Class Name
Average	$\frac{f(v_i) + f(v_j)}{2}$	`AverageEmbedder`
Hadamard	$f(v_i) * f(v_j)$	`HadamardEmbedder`
Weighted-L1	$\lvert f(v_i) - f(v_j) \rvert$	`WeightedL1Embedder`
Weighted-L2	$\lvert f(v_i) - f(v_j) \rvert^2$	`WeightedL2Embedder`

Table 3.1 – Edge embedding operators with their equation and class name in the Node2Vec library

In the following code, we will show how to perform the node embedding of a given `networkx` graph using Python and the Node2Vec library:

```
from node2vec.edges import HadamardEmbedder
embedding = HadamardEmbedder(keyed_vectors=model.wv)
```

The code is quite simple. The `HadamardEmbedder` class is instantiated with only the `keyed_vectors` parameter. The value of this parameter is the embedding model generated by Node2Vec. In order to use other techniques to generate the edge embedding, we just need to change the class and select one from the ones listed in *Table 3.1*. An example of the application of this algorithm is shown in the following figure:

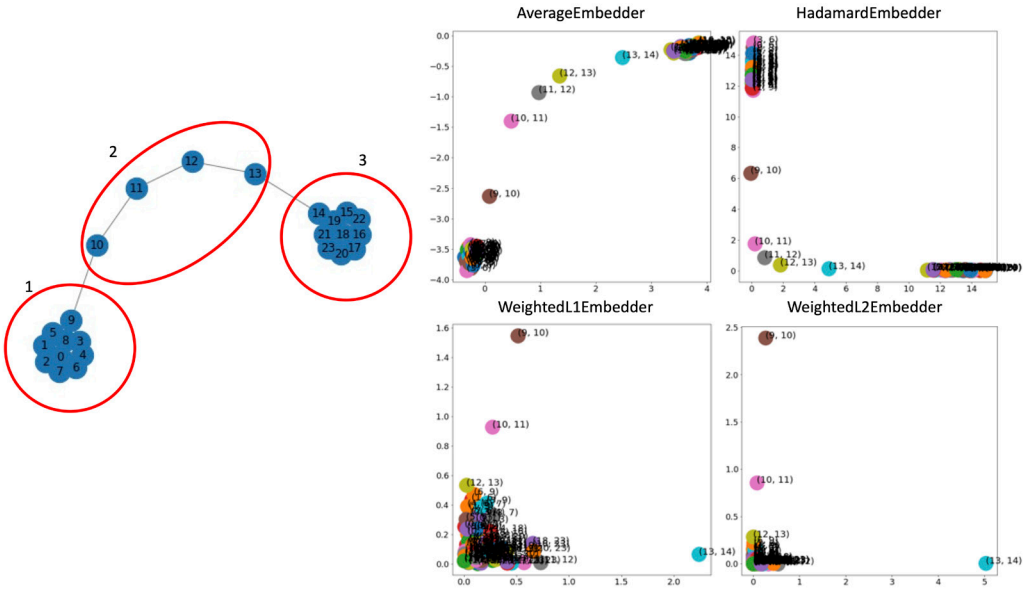

Figure 3.11 – Application of the Edge2Vec algorithm to a graph (top) to generate the embedding vector of its nodes (bottom) using different methods

From *Figure 3.11*, we can see how different embedding methods generate completely different embedding spaces. `AverageEmbedder` and `HadamardEmbedder`, in this example, generate well-separated embeddings for regions 1, 2, and 3.

For `WeightedL1Embedder` and `WeightedL2Embedder`, however, the embedding space is not well separated since the edge embeddings are concentrated in a single region without showing clear clusters.

Graph2Vec

The methods we previously described generated the embedding space for each node or edge on a given graph. **Graph to Vector** (**Graph2Vec**) generalizes this concept and generates embeddings for the whole graph.

To specify, given a set of graphs, the Graph2Vec algorithms generate an embedding space where each point represents a graph. This algorithm generates its embedding using an evolution of the Word2Vec skip-gram model known as **Document to Vector** (**Doc2Vec**). We can graphically see a simplification of this model in *Figure 3.12*:

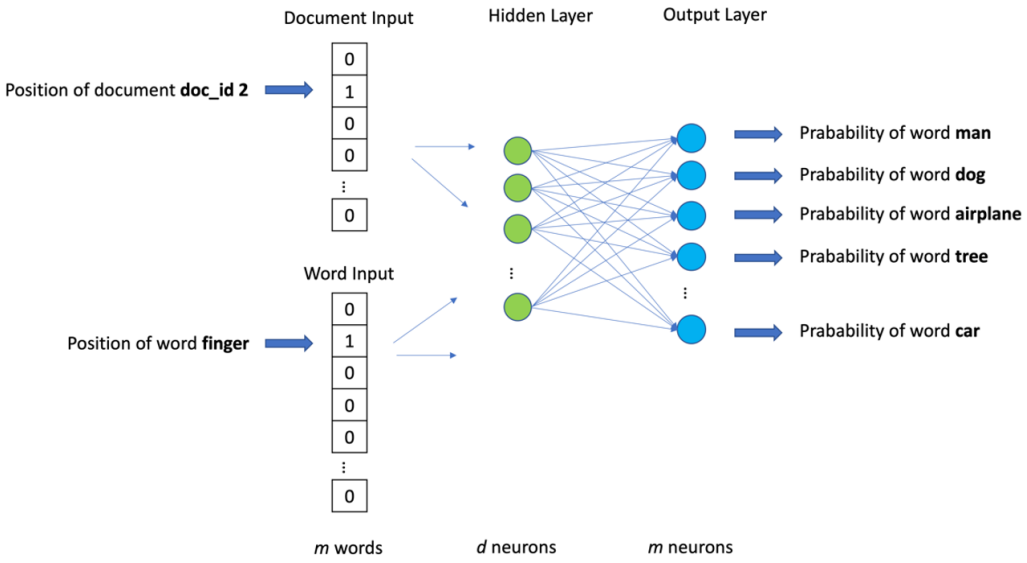

Figure 3.12 – Simplified graphical representation of the Doc2Vec skip-gram model. The number of d neurons in the hidden layer represents the final size of the embedding space

Compared to the simple Word2Vec, Doc2Vec also accepts another binary array representing the document containing the input word. Given a "target" document and a "target" word, the model then tries to predict the most probable "context" word with respect to the input "target" word and document.

With the introduction of the Doc2Vec model, we can now describe the Graph2Vec algorithm. The main idea behind this method is to view an entire graph as a document and each of its subgraphs, generated as an ego graph (see *Chapter 1*, *Getting Started with Graphs*) of each node, as words that comprise the document.

In other words, a graph is composed of subgraphs as a document is composed of sentences. According to this description, the algorithm can be summarized into the following steps:

1. **Subgraph generation**: A set of rooted subgraphs is generated around every node.
2. **Doc2Vec training**: The Doc2Vec skip-gram is trained using the subgraphs generated by the previous step.
3. **Embedding generation**: The information contained in the hidden layers of the trained Doc2Vec model is used in order to extract the embedding of each node.

In the following code, as we already did in *Chapter 2*, *Graph Machine Learning*, we will show how to perform the node embedding of a set of `networkx` graphs using Python and the `karateclub` library:

```
import matplotlib.pyplot as plt
from karateclub import Graph2Vec
n_graphs = 20
def generate_random():
    n = random.randint(5, 20)
    k = random.randint(5, n)
    p = random.uniform(0, 1)
    return nx.watts_strogatz_graph(n,k,p)

Gs = [generate_random() for x in range(n_graphs)]

model = Graph2Vec(dimensions=2)
model.fit(Gs)
embeddings = model.get_embedding()
```

In this example, the following have been done:

1. 20 Watts-Strogatz graphs have been generated with random parameters.
2. We then initialize the `Graph2Vec` class from the `karateclub` library with two dimensions. In this implementation, the `dimensions` parameter represents the dimension of the embedding space.
3. The model is then fitted on the input data using `model.fit(Gs)`.

4. The vector containing the embeddings is extracted using `model.get_embedding()`.

   The results of the code are shown in the following figure:

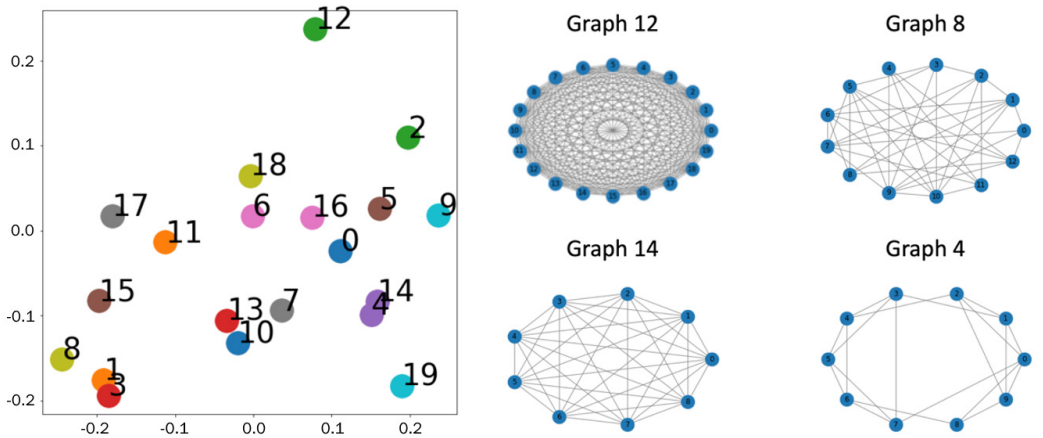

Figure 3.13 – Application of the Graph2Vec algorithm to a graph (left) to generate the embedding vector of its nodes (right) using different methods

From *Figure 3.13*, it is possible to see the embedding space generated for the different graphs.

In this section, we described different shallow embedding methods based on matrix factorization and the skip-gram model. However, in the scientific literature, a lot of unsupervised embedding algorithms exist, such as Laplacian methods. We refer those of you who are interested in exploring those methods to look at the paper *Machine Learning on Graphs: A Model and Comprehensive Taxonomy* available at `https://arxiv.org/pdf/2005.03675.pdf`.

We will continue our description of the unsupervised graph embedding method in the next sections. We will describe more complex graph embedding algorithms based on autoencoders.

Autoencoders

Autoencoders are an extremely powerful tool that can effectively help data scientists to deal with high-dimensional datasets. Although first presented around 30 years ago, in recent years, autoencoders have become more and more widespread in conjunction with the general rise of neural network-based algorithms. Besides allowing us to compact sparse representations, they can also be at the base of generative models, representing the first inception of the famous **Generative Adversarial Network** (**GAN**), which is, using the words of Geoffrey Hinton:

"The most interesting idea in the last 10 years in machine learning"

An autoencoder is a neural network where the inputs and outputs are basically the same, but that is characterized by a small number of units in the hidden layer. Loosely speaking, it is a neural network that is trained to reconstruct its inputs using a significantly lower number of variables and/or degree of freedom.

Since an autoencoder does not need a labeled dataset, it can be seen as an example of unsupervised learning and a dimensionality-reduction technique. However, different from other techniques such as **Principal Component Analysis** (**PCA**) and matrix factorization, autoencoders can learn non-linear transformation thanks to the non-linear activation functions of their neurons:

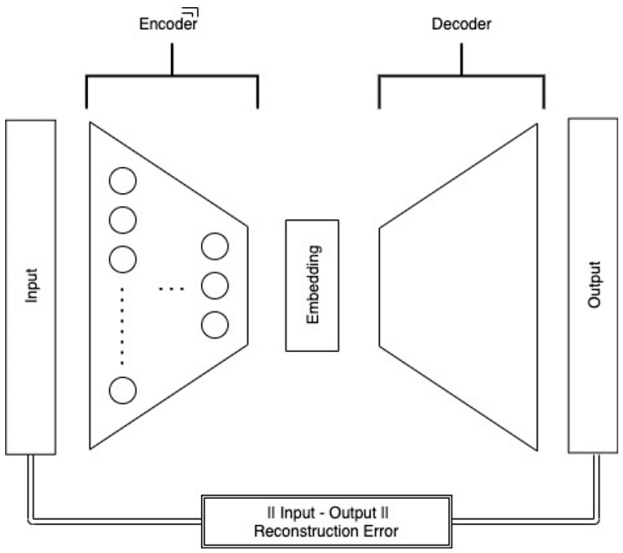

Figure 3.14 – Diagram of the autoencoder structure. The colors in the input and output layers represent the fact that the values should be as similar as possible. In fact, the training of the network is done in order to match these values and minimize the reconstruction error

Figure 3.14 shows a simple example of an autoencoder. You can see how the autoencoder can generally be seen as composed of two parts:

- An encoder network that processes the input through one or more units and maps it into an encoded representation that reduces the dimension of the inputs (under-complete autoencoders) and/or constrains its sparsity (over-complete regularized autoencoders)
- A decoder network that reconstructs the input signal from the encoded representation of the middle layer

The encoder-decoder structure is then trained to minimize the ability of the full network to reconstruct the input. In order to completely specify an autoencoder, we need a loss function. The error between the inputs and the outputs can be computed using different metrics and indeed the choice of the correct form for the "reconstruction" error is a critical point when building an autoencoder.

Some common choices for the loss functions that measure the reconstruction error are **mean square error**, **mean absolute error**, **cross-entropy**, and **KL divergence**.

In the following sections, we will show you how to build an autoencoder starting with some basic concepts and then applying those concepts to graph structures. But before diving in, we feel compelled to give you a very brief introduction to the frameworks that will allow us to do this: TensorFlow and Keras.

TensorFlow and Keras – a powerful combination

Released as open source by Google in 2017, TensorFlow is now the standard, de facto framework that allows symbolic computations and differential programming. It basically allows you to build a symbolic structure that describes how inputs are combined in order to produce the outputs, defining what is generally called a **computational graph** or a **stateful dataflow graph**. In this graph, nodes are the variable (scalar, arrays, tensors) and edges represent operations connecting the inputs (edge source) to the output (edge target) of a single operation.

In TensorFlow, such a graph is static (this is indeed one of the main differences with respect to another very popular framework in this context: `torch`) and can be executed by feeding data into it, as inputs, clearing the "dataflow" attribute mentioned previously.

By abstracting the computation, TensorFlow is a very general tool that can run on multiple backends: on machines powered by CPUs, GPUs, or even ad hoc, specifically designed processing units such as TPUs. Besides, TensorFlow-powered applications can also be deployed on different devices, ranging from single and distributed servers to mobile devices.

Besides abstracting computation, TensorFlow also allows you to symbolically differentiate your computational graph with respect to any of its variables, resulting in a new computational graph that can also be differentiated to produce higher-order derivatives. This approach is generally referred to as symbol-to-symbol derivative and it is indeed extremely powerful, especially in the context of the optimization of the generic loss function, which requires gradient estimations (such as gradient descent techniques).

As you might know, the problem of optimizing a loss function with respect to many parameters is central in the training of any neural network via backpropagation. This is surely the main reason why TensorFlow has become very popular in the past few years and why it was designed and produced in the first place by Google.

Diving in depth into the usage of TensorFlow is beyond the scope of this book and indeed you can find out more through the description given in dedicated books. In the following sections, we will use some of its main functionalities and provide you with the basic tools for building neural networks.

Since its last major release, 2.x, the standard way of building a model with TensorFlow is using the Keras API. Keras was natively a side external project with respect to TensorFlow, aimed at providing a common and simple API to use several differential programming frameworks, such as TensorFlow, Teano, and CNTK, for implementing a neural network model. It generally abstracts the low-level implementation of the computation graph and provides you with the most common layers used when building neural networks (although custom layers can also be easily implemented), such as the following:

- Convolutional layers
- Recurrent layers
- Regularization layers
- Loss functions

Keras also exposes APIs that are very similar to scikit-learn, the most popular library for machine learning in the Python ecosystem, making it very easy for data scientists to build, train, and integrate neural network-based models in their applications.

In the next section, we will show you how to build and train an autoencoder using Keras. We'll start applying these techniques to images in order to progressively apply the key concepts to graph structures.

Our first autoencoder

We'll start by implementing an autoencoder in its simplest form, that is, a simple feed-forward network trained to reconstruct its input. We'll apply this to the Fashion-MNIST dataset, which is a dataset similar to the famous MNIST dataset that features hand-written numbers on a black and white image.

MNIST has 10 categories and consists of 60k + 10k (train dataset + test dataset) 28x28 pixel grayscale images that represent a piece of clothing (`T-shirt`, `Trouser`, `Pullover`, `Dress`, `Coat`, `Sandal`, `Shirt`, `Sneaker`, `Bag`, and `Ankle boot`). The Fashion-MNIST dataset is a harder task than the original MNIST dataset and it is generally used for benchmarking algorithms.

The dataset is already integrated in the Keras library and can be easily imported using the following code:

```
from tensorflow.keras.datasets import fashion_mnist
(x_train, y_train), (x_test, y_test) = fashion_mnist.load_
data()
```

It is usually good practice to rescale the inputs with an order of magnitude of around 1 (for which activation functions are most efficient) and make sure that the numerical data is in single-precision (32 bits) instead of double-precision (64 bits). This is due to the fact that it is generally desirable to promote speed rather than precision when training a neural network, which is a computationally expensive process. In certain cases, the precision could even be lowered to half-precision (16 bits). We transform the input with the following:

```
x_train = x_train.astype('float32') / 255.
x_test = x_test.astype('float32') / 255.
```

We can grasp the type of inputs we are dealing with by plotting some of the samples from the training set using the following code:

```
n = 10
plt.figure(figsize=(20, 4))
for i in range(n):
    ax = plt.subplot(1, n, i + 1)
    plt.imshow(x_train[i])
    plt.title(classes[y_train[i]])
    plt.gray()
    ax.get_xaxis().set_visible(False)
```

```
    ax.get_yaxis().set_visible(False)
plt.show()
```

In the preceding code, `classes` represents the mapping between integers and class names, for example, `T-shirt`, `Trouser`, `Pullover`, `Dress`, `Coat`, `Sandal`, `Shirt`, `Sneaker`, `Bag`, and `Ankle boot`:

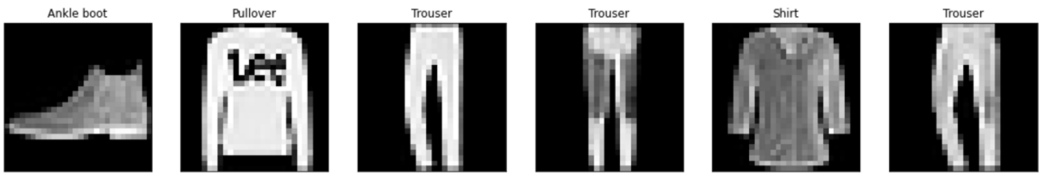

Figure 3.15 – Some samples taken from the training set of the Fashion-MNIST dataset

Now that we have imported the inputs, we can build our autoencoder network by creating the encoder and the decoder. We will be doing this using the Keras functional API, which provides more generality and flexibility compared to the so-called Sequential API. We start by defining the encoder network:

```
from tensorflow.keras.layers import Conv2D, Dropout,
MaxPooling2D, UpSampling2D, Input
input_img = Input(shape=(28, 28, 1))
x = Conv2D(16, (3, 3), activation='relu', padding='same')
(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)
```

Our network is composed of a stack of three levels of the same pattern composed of the same two-layer building block:

- `Conv2D`, a two-dimensional convolutional kernel that is applied to the input and effectively corresponds to having weights shared across all the input neurons. After applying the convolutional kernel, the output is transformed using the ReLU activation function. This structure is replicated for *n* hidden planes, with *n* being 16 in the first stacked layer and 8 in the second and third stacked layers.
- `MaxPooling2D`, which down-samples the inputs by taking the maximum value over the specified window (2x2 in this case).

Using the Keras API, we can also have an overview of how the layers transformed the inputs using the `Model` class, which converts the tensors into a user-friendly model ready to be used and explored:

```
Model(input_img, encoded).summary()
```

This provides a summary of the encoder network visible in *Figure 3.16*:

```
Model: "model_4"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 28, 28, 1)]       0
_________________________________________________________________
gaussian_noise (GaussianNois (None, 28, 28, 1)         0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 28, 28, 16)        160
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 14, 14, 16)        0
_________________________________________________________________
conv2d_8 (Conv2D)            (None, 14, 14, 8)         1160
_________________________________________________________________
max_pooling2d_4 (MaxPooling2 (None, 7, 7, 8)           0
_________________________________________________________________
conv2d_9 (Conv2D)            (None, 7, 7, 8)           584
_________________________________________________________________
max_pooling2d_5 (MaxPooling2 (None, 4, 4, 8)           0
=================================================================
Total params: 1,904
Trainable params: 1,904
Non-trainable params: 0
_________________________________________________________________
```

Figure 3.16 – Overview of the encoder network

As can be seen, at the end of the encoding phase, we have a (4, 4, 8) tensor, which is more than six times smaller than our original initial inputs (28x28). We can now build the decoder network. Note that the encoder and decoder do not need to have the same structure and/or shared weights:

```
x = Conv2D(8, (3, 3), activation='relu', padding='same')
(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(8, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
x = Conv2D(16, (3, 3), activation='relu')(x)
x = UpSampling2D((2, 2))(x)
```

```
decoded = Conv2D(1, (3, 3), activation='sigmoid',
padding='same')(x)
```

In this case, the decoder network resembles the encoder structure where the down-sampling of the input achieved using the `MaxPooling2D` layer has been replaced by the `UpSampling2D` layer, which basically repeats the input over a specified window (2x2 in this case, effectively doubling the tensor in each direction).

We have now fully defined the network structure with the encoder and decoder layers. In order to completely specify our autoencoder, we also need to specify a loss function. Moreover, to build the computational graph, Keras also needs to know which algorithms should be used in order to optimize the network weights. Both bits of information, the loss function and optimizer to be used, are generally provided to Keras when *compiling* the model:

```
autoencoder = Model(input_img, decoded)
autoencoder.compile(optimizer='adam', loss='binary_
crossentropy')
```

We can now finally train our autoencoder. Keras `Model` classes provide APIs that are similar to scikit-learn, with a `fit` method to be used to train the neural network. Note that, owing to the nature of the autoencoder, we are using the same information as the input and output of our network:

```
autoencoder.fit(x_train, x_train,
                epochs=50,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test, x_test))
```

Once the training is finished, we can examine the ability of the network to reconstruct the inputs by comparing input images with their reconstructed version, which can be easily computed using the `predict` method of the Keras `Model` class as follows:

```
decoded_imgs = autoencoder.predict(x_test)
```

In *Figure 3.17*, we show the reconstructed images. As you can see, the network is quite good at reconstructing unseen images, especially when considering the large-scale features. Details might have been lost in the compression (see, for instance, the logo on the t-shirts) but the overall relevant information has indeed been captured by our network:

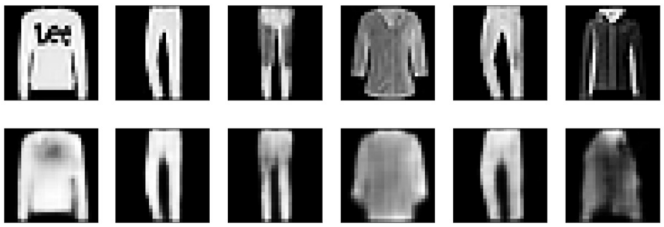

Figure 3.17 – Examples of the reconstruction done on the test set by the trained autoencoder

It can also be very interesting to represent the encoded version of the images in a two-dimensional plane using T-SNE:

```
from tensorflow.keras.layers import Flatten
embed_layer = Flatten()(encoded)
embeddings = Model(input_img, embed_layer).predict(x_test)
tsne = TSNE(n_components=2)
emb2d = tsne.fit_transform(embeddings)
x, y = np.squeeze(emb2d[:, 0]), np.squeeze(emb2d[:, 1])
```

The coordinates provided by T-SNE are shown in *Figure 3.18*, colored by the class the sample belongs to. The clustering of the different clothing can clearly be seen, particularly for some classes that are very well separated from the rest:

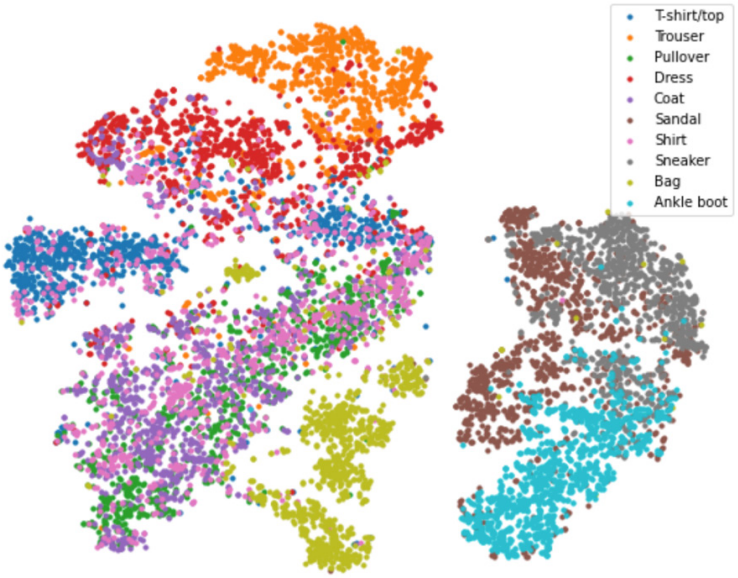

Figure 3.18 – T-SNE transformation of the embeddings extracted from the test set, colored by the class that the sample belongs to

Autoencoders are, however, rather prone to overfitting, as they tend to re-create exactly the images of the training and not generalize well. In the following subsection, we will see how overfitting can be prevented in order to build more robust and reliable dense representations.

Denoising autoencoders

Besides allowing us to compress a sparse representation into a denser vector, autoencoders are also widely used to process a signal in order to filter out noise and extract only a relevant (characteristic) signal. This can be very useful in many applications, especially when identifying anomalies and outliers.

Denoising autoencoders are a small variation of what has been implemented. As described in the previous section, basic autoencoders are trained using the same image as input and output. Denoising autoencoders corrupt the input using some noise of various intensity, while keeping the same noise-free target. This could be achieved by simply adding some Gaussian noise to the inputs:

```
noise_factor = 0.1
x_train_noisy = x_train + noise_factor * np.random.
normal(loc=0.0, scale=1.0, size=x_train.shape)
x_test_noisy = x_test + noise_factor * np.random.
normal(loc=0.0, scale=1.0, size=x_test.shape)

x_train_noisy = np.clip(x_train_noisy, 0., 1.)
x_test_noisy = np.clip(x_test_noisy, 0., 1.)
```

The network can then be trained using the corrupted input, while for the output the noise-free image is used:

```
noisy_autoencoder.fit(x_train_noisy, x_train,
                epochs=50,
                batch_size=128,
                shuffle=True,
                validation_data=(x_test_noisy, x_test))
```

Such an approach is generally valid when datasets are large and when the risk of overfitting the noise is rather limited. When datasets are smaller, an alternative to avoid the network "learning" the noise as well (thus learning the mapping between a static noisy image to its noise-free version) is to add training stochastic noise using a `GaussianNoise` layer.

Note that in this way, the noise may change between epochs and prevent the network from learning a static corruption superimposed to our training set. In order to do so, we change the first layers of our network in the following way:

```
input_img = Input(shape=(28, 28, 1))
noisy_input = GaussianNoise(0.1)(input_img)
x = Conv2D(16, (3, 3), activation='relu', padding='same')
(noisy_input)
```

The difference is that instead of having statically corrupted samples (that do not change in time), the noisy inputs now keep changing between epochs, thus avoiding the network learning the noise as well.

The `GaussianNoise` layer is an example of a regularization layer, that is, a layer that helps reduce overfitting of a neural network by inserting a random part in the network. `GaussianNoise` layers make models more robust and able to generalize better, avoiding autoencoders learning the identity function.

Another common example of a regularization layer is the dropout layers that effectively set to 0 some of the inputs (at random with a probability, p_0) and rescale the other inputs by a $1/1 - p_0$ factor in order to (statistically) keep the sum over all the units constant, with and without dropout.

Dropout corresponds to randomly killing some of the connections between layers in order to reduce output dependence to specific neurons. You need to keep in mind that regularization layers are only active at training, while at test time they simply correspond to identity layers.

In *Figure 3.19*, we compare the network reconstruction of a noisy input (input) for the previous unregularized trained network and the network with a `GaussianNoise` layer. As can be seen (compare, for instance, the images of trousers), the model with regularization tends to develop stronger robustness and reconstructs the noise-free outputs:

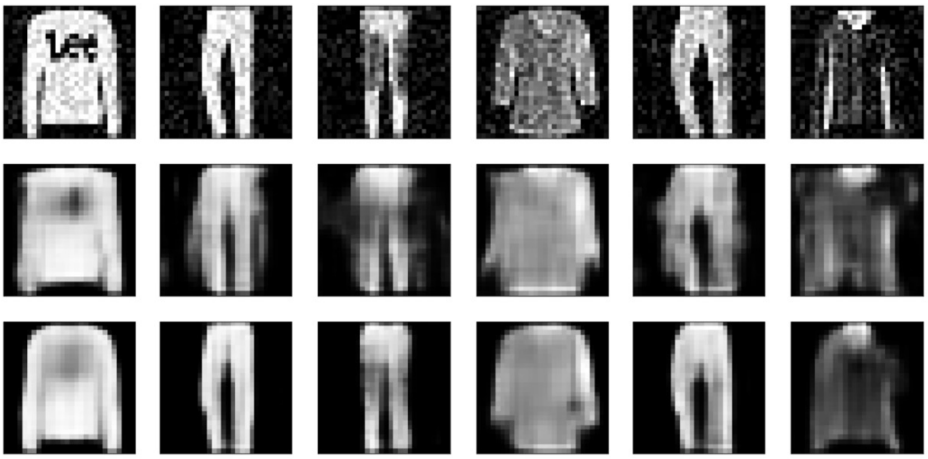

Figure 3.19 – Comparison with reconstruction for noisy samples. Top row: noisy input; middle row: reconstructed output using a vanilla autoencoder; bottom row: reconstructed output using a denoising autoencoder

Regularization layers are often used when dealing with deep neural networks that tend to overfit and are able to learn identity functions for autoencoders. Often, dropout or `GaussianNoise` layers are introduced, repeating a similar pattern composed of regularization and learnable layers that we usually refer to as **stacked denoising layers**.

Graph autoencoders

Once the basic concepts of autoencoders are understood, we can now turn to apply this framework to graph structures. If on one hand the network structure, decomposed into an encoder-decoder structure with a low-dimensional representation in between, still applies, the definition of the loss function to be optimized needs a bit of caution when dealing with networks. First, we need to adapt the reconstruction error to a meaningful formulation that can adapt to the peculiarities of graph structures. But to do so, let's first introduce the concepts of first- and higher-order proximity.

When applying autoencoders to graph structures, the input and output of the network should be a graph representation, as, for instance, the adjacency matrix. The reconstruction loss could then be defined as the Frobenius norm of the difference between the input and output matrices. However, when applying autoencoders to such graph structures and adjacency matrices, two critical issues arise:

- Whereas the presence of links indicates a relation or similarity between two vertices, their absence does not generally indicate a dissimilarity between vertices.
- The adjacency matrix is extremely sparse and therefore the model will naturally tend to predict a 0 rather than a positive value.

To address such peculiarities of graph structures, when defining the reconstruction loss, we need to penalize more errors done for the non-zero elements rather than that for zero elements. This can be done using the following loss function:

$$\mathcal{L}_{2nd} = \sum_{i=1}^{n} \|(\tilde{X}_i - X_i) \odot b_i\|$$

Here, $\odot$ is the Hadamard element-wise product, where $b_{ij} = \beta > 1$ if there is an edge between nodes i and j, and 0 otherwise. The preceding loss guarantees that vertices that share a neighborhood (that is, their adjacency vectors are similar) will also be close in the embedding space. Thus, the preceding formulation will naturally preserve second-order proximity for the reconstructed graph.

On the other hand, you can also promote first-order proximity in the reconstructed graph, thus enforcing connected nodes to be close in the embedding space. This condition can be enforced by using the following loss:

$$\mathcal{L}_{1th} = \sum_{i,j=1}^{n} s_{ij} \|y_j - y_i\|_2^2$$

Here, y_i and y_j are the two representation of nodes i and j in the embedding space. This loss function forces neighboring nodes to be close in the embedding space. In fact, if two nodes are tightly connected, s_{ij} will be large. As a consequence, their difference in the embedding space, $\|y_j - y_i\|_2^2$, should be limited (indicating the two nodes are close in the embedding space) to keep the loss function small. The two losses can also be combined into a single loss function, where, in order to prevent overfitting, a regularization loss can be added that is proportional to the norm of the weight coefficients:

$$\mathcal{L}_{tot} = \mathcal{L}_{2nd} + \alpha \cdot \mathcal{L}_{tot} + \nu \cdot \mathcal{L}_{reg} = \mathcal{L}_{2nd} + \alpha \cdot \mathcal{L}_{tot} + \nu \cdot \|W\|_F^2$$

In the preceding equation, *W* represents all the weights used across the network. The preceding formulation was proposed in 2016 by Wang et al., and it is now known as **Structural Deep Network Embedding** (**SDNE**).

Although the preceding loss could also be directly implemented with TensorFlow and Keras, you can already find this network integrated in the GEM package we referred to previously. As before, extracting the node embedding can be done similarly in a few lines of code, as follows:

```
G=nx.karate_club_graph()
sdne=SDNE(d=2, beta=5, alpha=1e-5, nu1=1e-6, nu2=1e-6,
          K=3,n_units=[50, 15,], rho=0.3, n_iter=10,
          xeta=0.01,n_batch=100,
          modelfile=['enc_model.json','dec_model.json'],
          weightfile=['enc_weights.hdf5','dec_weights.hdf5'])
sdne.learn_embedding(G)
embeddings = m1.get_embedding()
```

Although very powerful, these graph autoencoders encounter some issues when dealing with large graphs. For these cases, the input of our autoencoder is one row of the adjacency matrix that has as many elements as the nodes in the network. In large networks, this size can easily be of the order of millions or tens of millions.

In the next section, we describe a different strategy for encoding the network information that in some cases may iteratively aggregate embeddings only over local neighborhoods, making it scalable to large graphs.

Graph neural networks

GNNs are deep learning methods that work on graph-structured data. This family of methods is also known as **geometric deep learning** and is gaining increasing interest in a variety of applications, including social network analysis and computer graphics.

According to the taxonomy defined in *Chapter 2*, *Graph Machine Learning*, the encoder part takes as input both the graph structure and the node features. Those algorithms can be trained either with or without supervision. In this chapter, we will focus on unsupervised training, while the supervised setting will be explored in *Chapter 4*, *Supervised Graph Learning*.

If you are familiar with the concept of a **Convolutional Neural Network** (**CNN**), you might already know that they are able to achieve impressive results when dealing with regular Euclidean spaces, such as text (one-dimensional), images (two-dimensional), and videos (three-dimensional). A classic CNN consists of a sequence of layers and each layer extracts multi-scale localized spatial features. Those features are exploited by deeper layers to construct more complex and highly expressive representations.

In recent years, it has been observed that concepts such as multi-layer and locality are also useful for processing graph-structured data. However, graphs are defined over a *non-Euclidean space*, and finding a generalization of a CNN for graphs is not straightforward, as described in *Figure 3.20*:

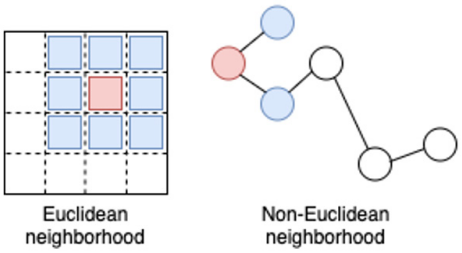

Figure 3.20 – Visual difference between Euclidean and non-Euclidean neighborhoods

The original formulation of GNN was proposed by Scarselli et al. back in 2009. It relies on the fact that each node can be described by its features and its neighborhood. Information coming from the neighborhood (which represents the concept of locality in the graph domain) can be aggregated and used to compute more complex and high-level features. Let's understand in more detail how it can be done.

At the beginning, each node, v_i, is associated with a state. Let's start with a random embedding, h_i^t (ignoring node attributes for simplicity). At each iteration of the algorithm, nodes accumulate input from their neighbors using a simple neural network layer:

$$h_i^t = \sum_{v_j \in N(v_i)} \sigma\left(W h_j^{t-1} + b\right)$$

Here, $W \in \mathbb{R}^{d \times d}$ and $b \in \mathbb{R}^d$ are trainable parameters (where d is the dimension of the embedding), σ is a non-linear function, and t represents the tth iteration of the algorithm. The equation is applied recursively until a particular objective is reached. Notice that, at each iteration, the *previous state* (the state computed at the previous iteration) is exploited in order to compute that the new state has happened with *recurrent neural networks*.

Variants of GNNs

Starting from this first idea, several attempts have been made in recent years to re-address the problem of learning from graph data. In particular, variants of the previously described GNN have been proposed, with the aim of improving its representation learning capability. Some of them are specifically designed to process specific types of graphs (direct, indirect, weighted, unweighted, static, dynamic, and so on).

Also, several modifications have been proposed for the propagation step (convolution, gate mechanisms, attention mechanisms, and skip connections, among others), with the aim of improving representation at different levels. Also, different training methods have been proposed to improve learning.

When dealing with unsupervised representation learning, one of the most common approaches is to use an encoder to embed the graph (the encoder is formulated as one of the GNN variants) and then use a simple decoder to reconstruct the adjacency matrix. The loss function is usually formulated as the similarity between the original adjacency matrix and the reconstructed one. Formally, it can be defined as follows:

$$Z = GNN(X, A)$$
$$\hat{A} = ZZ^T$$

Here, $A \in \mathbb{R}^{N \times N}$ is the adjacency matrix representation and $X \in \mathbb{R}^{N \times d}$ is the matrix of node attributes. Another common variant of this approach, especially used when dealing with graph classification/representation learning, is to train against a *target distance*. The idea is to embed two pairs of graphs simultaneously obtaining a combined representation. The model is then trained such that this representation matches the distance. A similar strategy can be also adopted when dealing with node classification/representation learning by using a node similarity function.

Graph Convolutional Neural Network (**GCN**)-based encoders are one of the most diffused variants of GNN for unsupervised learning. GCNs are GNN models inspired by many of the basic ideas behind CNN. Filter parameters are typically shared over all locations in the graph and several layers are concatenated to form a deep network.

There are essentially two types of convolutional operations for graph data, namely **spectral approaches** and **non-spectral** (**spatial**) approaches. The first, as the name suggests, defines convolution in the spectral domain (that is, decomposing graphs in a combination of simpler elements). Spatial convolution formulates the convolution as aggregating feature information from neighbors.

Spectral graph convolution

Spectral approaches are related to spectral graph theory, the study of the characteristics of a graph in relation to the characteristic polynomial, eigenvalues, and eigenvectors of the matrices associated with the graph. The convolution operation is defined as the multiplication of a signal (node features) by a kernel. In more detail, it is defined in the Fourier domain by determining the *eigendecomposition of the graph Laplacian* (think about the graph Laplacian as an adjacency matrix normalized in a special way).

While this definition of spectral convolution has a strong mathematical foundation, the operation is computationally expensive. For this reason, several works have been done to approximate it in an efficient way. ChebNet by Defferrard et al., for instance, is one of the first seminal works on spectral graph convolution. Here, the operation is approximated by using the concept of the Chebyshev polynomial of order *K* (a special kind of polynomial used to efficiently approximate functions).

Here, *K* is a very useful parameter because it determines the locality of the filter. Intuitively, for *K*=1, only the node features are fed into the network. With *K*=2, we average over two-hop neighbors (neighbors of neighbors) and so on.

Let $X \in \mathbb{R}^{N \times d}$ be the matrix of node features. In classical neural network processing, this signal would be composed of layers of the following form:

$$H^l = \sigma(XW)$$

Here, $W \in \mathbb{R}^{N \times N}$ is the layer weights and σ represents some non-linear activation function. The drawback of this operation is that it processes each node signal independently without taking into account connections between nodes. To overcome this limitation, a simple (yet effective) modification can be done, as follows:

$$H^l = \sigma(AXW)$$

By introducing the adjacency matrix, $A \in \mathbb{R}^{N \times N}$, a new linear combination between each node and its corresponding neighbors is added. This way, the information depends only on the neighborhood and parameters are applied on all the nodes, simultaneously.

It is worth noting that this operation can be repeated in sequence several times, thus creating a deep network. At each layer, the node descriptors, *X*, will be replaced with the output of the previous layer, H^{l-1}.

The preceding presented equation, however, has some limitations and cannot be applied as it stands. The first limitation is that by multiplying by *A*, we consider all the neighbors of the node but not the node itself. This problem can be easily overcome by adding self-loops in the graph, that is, adding the $\hat{A} = A + I$ identity matrix.

The second limitation is related to the adjacency matrix itself. Since it is typically not normalized, we will observe large values in the feature representation of high-degree nodes and small values in the feature representation of low-degree nodes. This will lead to several problems during training since optimization algorithms are often sensitive to feature scale. Several methods have been proposed for normalizing *A*.

In Kipf and Welling, 2017 (one of the well-known GCN models), for example, the normalization is performed by multiplying *A* by the *diagonal node degree matrix D*, such that all the rows sum to 1: $D^{-1}A$. More specifically, they used symmetric normalization ($D^{-1/2}AD^{-1/2}$), such that the proposed propagation rule becomes as follows:

$$H^l = \sigma(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}}XW)$$

Here, $\hat{D}$ is the diagonal node degree matrix of $\hat{A}$.

In the following example, we will create a GCN as defined in Kipf and Welling and we will apply this propagation rule for embedding a well-known network: a Zachary's karate club graph:

1. To begin, it is necessary to import all the Python modules. We will use `networkx` to load the *barbell graph*:

```
import networkx as nx
import numpy as np
G = nx.barbell_graph(m1=10,m2=4)
```

2. To implement the GC propagation rule, we need an adjacency matrix representing `G`. Since this network does not have node features, we will use the $I \in \mathbb{R}^{N\times N}$ identity matrix as the node descriptor:

```
A = nx.to_numpy_matrix(G)
 I = np.eye(G.number_of_nodes())
```

3. We now add the self-loop and prepare the diagonal node degree matrix:

```
from scipy.linalg import sqrtm

A_hat = A + I
D_hat = np.array(np.sum(A_hat, axis=0))[0]
 D_hat = np.array(np.diag(D_hat))
 D_hat = np.linalg.inv(sqrtm(D_hat))
 A_norm = D_hat @ A_hat @ D_hat
```

4. Our GCN will be composed of two layers. Let's define the layers' weights and the propagation rule. Layer weights, *W*, will be initialized using *Glorot uniform initialization* (even if other initialization methods can be also used, for example, by sampling from a Gaussian or uniform distribution):

```
def glorot_init(nin, nout):
     sd = np.sqrt(6.0 / (nin + nout))
     return np.random.uniform(-sd, sd, size=(nin, nout))
class GCNLayer():
  def __init__(self, n_inputs, n_outputs):
      self.n_inputs = n_inputs
      self.n_outputs = n_outputs
      self.W = glorot_init(self.n_outputs, self.n_inputs)
      self.activation = np.tanh
  def forward(self, A, X):
      self._X = (A @ X).T
      H = self.W @ self._X
      H = self.activation(H)
      return H.T # (n_outputs, N)
```

5. Finally, let's create our network and compute the forward pass, that is, propagate the signal through the network:

```
gcn1 = GCNLayer(G.number_of_nodes(), 8)
 gcn2 = GCNLayer(8, 4)
 gcn3 = GCNLayer(4, 2)
H1 = gcn1.forward(A_norm, I)
 H2 = gcn2.forward(A_norm, H1)

H3 = gcn3.forward(A_norm, H2)
```

`H3` now contains the embedding computed using the GCN propagation rule. Notice that we chose `2` as the number of outputs, meaning that the embedding is bi-dimensional and can be easily visualized. In *Figure 3.21*, you can see the output:

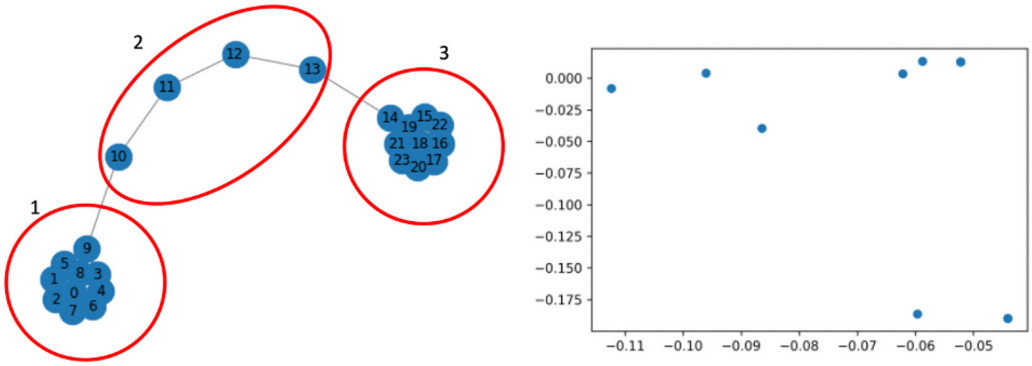

Figure 3.21 – Application of the graph convolutional layer to a graph (left) to generate the embedding vector of its nodes (right)

You can observe the presence of two quite well-separated communities. This is a nice result, considering that we have not trained the network yet!

Spectral graph convolution methods have achieved noteworthy results in many domains. However, they present some drawbacks. Consider, for example, a very big graph with billions of nodes: a spectral approach requires the graph to be processed simultaneously, which can be impractical from a computational point of view.

Furthermore, spectral convolution often assumes a fixed graph, leading to poor generalization capabilities on new, different graphs. To overcome these issues, spatial graph convolution represents an interesting alternative.

Spatial graph convolution

Spatial graph convolutional networks perform the operations directly on the graph by aggregating information from spatially close neighbors. Spatial convolution has many advantages: weights can be easily shared across a different location of the graph, leading to a good generalization capability on different graphs. Furthermore, the computation can be done by considering subsets of nodes instead of the entire graph, potentially improving computational efficiency.

GraphSAGE is one of the algorithms that implement spatial convolution. One of the main characteristics is its ability to scale over various types of networks. We can think of GraphSAGE as composed of three steps:

1. **Neighborhood sampling**: For each node in a graph, the first step is to find its k-neighborhood, where *k* is defined by the user for determining how many hops to consider (neighbors of neighbors).
2. **Aggregation**: The second step is to aggregate, for each node, the node features describing the respective neighborhood. Various types of aggregation can be performed, including average, pooling (for example, taking the best feature according to certain criteria), or an even more complicated operation, such as using recurrent units (such as LSTM).
3. **Prediction**: Each node is equipped with a simple neural network that learns how to perform predictions based on the aggregated features from the neighbors.

GraphSAGE is often used in supervised settings, as we will see in *Chapter 4*, *Supervised Graph Learning*. However, by adopting strategies such as using a similarity function as the target distance, it can also be effective for learning embedding without explicitly supervising the task.

Graph convolution in practice

In practice, GNNs have been implemented in many machine learning and deep learning frameworks, including TensorFlow, Keras, and PyTorch. For the next example, we will be using StellarGraph, the Python library for machine learning on graphs.

In the following example, we will learn about embedding vectors in an unsupervised manner, without a target variable. The method is inspired by Bai et al. 2019 and is based on the simultaneous embedding of pairs of graphs. This embedding should match a ground-truth distance between graphs:

1. First, let's load the required Python modules:

```
import numpy as np
import stellargraph as sg
from stellargraph.mapper import FullBatchNodeGenerator
from stellargraph.layer import GCN

import tensorflow as tf
from tensorflow.keras import layers, optimizers, losses,
metrics, Model
```

2. We will be using the `PROTEINS` dataset for this example, which is available in StellarGraph and consists of 1,114 graphs with 39 nodes and 73 edges on average for each graph. Each node is described by four attributes and belongs to one of two classes:

```
dataset = sg.datasets.PROTEINS()
graphs, graph_labels = dataset.load()
```

3. The next step is to create the model. It will be composed of two GC layers with 64 and 32 output dimensions followed by ReLU activation, respectively. The output will be computed as the Euclidean distance of the two embeddings:

```
generator = sg.mapper.PaddedGraphGenerator(graphs)

# define a GCN model containing 2 layers of size 64 and
32, respectively.
# ReLU activation function is used to add non-linearity
between layers
gc_model = sg.layer.GCNSupervisedGraphClassification(
 [64, 32], ["relu", "relu"], generator, pool_all_
layers=True)
# retrieve the input and the output tensor of the GC
layer such that they can be connected to the next layer

inp1, out1 = gc_model.in_out_tensors()
inp2, out2 = gc_model.in_out_tensors()
vec_distance = tf.norm(out1 - out2, axis=1)

# create the model. It is also useful to create a
specular model in order to easily retrieve the embeddings
pair_model = Model(inp1 + inp2, vec_distance)
 embedding_model = Model(inp1, out1)
```

4. It is now time to prepare the dataset for training. To each pair of input graphs, we will assign a similarity score. Notice that any notion of graph similarity can be used in this case, including graph edit distances. For simplicity, we will be using the distance between the spectrum of the Laplacian of the graphs:

```
def graph_distance(graph1, graph2):
    spec1 = nx.laplacian_spectrum(graph1.to_
```

```
networkx(feature_attr=None))
    spec2 = nx.laplacian_spectrum(graph2.to_
networkx(feature_attr=None))
    k = min(len(spec1), len(spec2))
    return np.linalg.norm(spec1[:k] - spec2[:k])

graph_idx = np.random.RandomState(0).randint(len(graphs),
size=(100, 2))
targets = [graph_distance(graphs[left], graphs[right])
for left, right in graph_idx]
train_gen = generator.flow(graph_idx, batch_size=10,
targets=targets)
```

5. Finally, let's compile and train the model. We will be using an adaptive moment estimation optimizer (Adam) with the learning rate parameter set to `1e-2`. The loss function we will be using is defined as the minimum squared error between the prediction and the ground-truth distance computed as previously. The model will be trained for 500 epochs:

```
pair_model.compile(optimizers.Adam(1e-2), loss="mse")
pair_model.fit(train_gen, epochs=500, verbose=0)
```

6. After training, we are now ready to inspect and visualize the learned representation. Since the output is 32-dimensional, we need a way to qualitatively evaluate the embeddings, for example, by plotting them in a bi-dimensional space. We will use T-SNE for this purpose:

```
# retrieve the embeddings
embeddings = embedding_model.predict(generator.
flow(graphs))
# TSNE is used for dimensionality reduction
from sklearn.manifold import TSNE
tsne = TSNE(2)
 two_d = tsne.fit_transform(embeddings)
```

Let's plot the embeddings. In the plot, each point (embedded graph) is colored according to the corresponding label (blue=0, red=1). The results are visible in *Figure 3.22*:

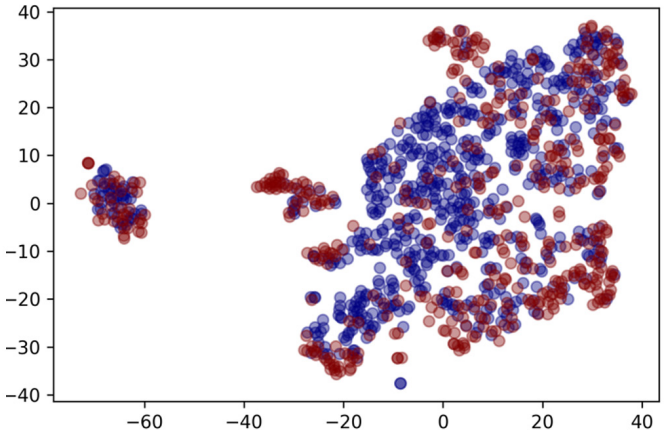

Figure 3.22 – The PROTEINS dataset embedding using GCNs

This is just one of the possible methods for learning embeddings for graphs. More advanced solutions can be experimented with to better fit the problem of interest.

Summary

In this chapter, we have learned how unsupervised machine learning can be effectively applied to graphs to solve real problems, such as node and graph representation learning.

In particular, we first analyzed shallow embedding methods, a set of algorithms that are able to learn and return only the embedding values for the learned input data.

We then learned how autoencoder algorithms can be used to encode the input by preserving important information in a lower-dimensional space. We have also seen how this idea can be adapted to graphs, by learning about embeddings that allow us to reconstruct the pair-wise node/graph similarity.

Finally, we introduced the main concepts behind GNNs. We have seen how well-known concepts, such as convolution, can be applied to graphs.

In the next chapter, we will revise these concepts in a supervised setting. There, a target label is provided and the objective is to learn a mapping between the input and the output.