2
Graph Machine Learning

Machine learning is a subset of artificial intelligence that aims to provide systems with the ability to *learn* and improve from data. It has achieved impressive results in many different applications, especially where it is difficult or unfeasible to explicitly define rules to solve a specific task. For instance, we can train algorithms to recognize spam emails, translate sentences into other languages, recognize objects in an image, and so on.

In recent years, there has been an increasing interest in applying machine learning to *graph-structured data*. Here, the primary objective is to automatically learn suitable representations to make predictions, discover new patterns, and understand complex dynamics in a better manner with respect to "traditional" machine learning approaches.

This chapter will first review some of the basic machine learning concepts. Then, an introduction to graph machine learning will be provided, with a particular focus on **representation learning**. We will then analyze a practical example to guide you through the comprehension of the theoretical concepts.

The following topics will be covered in this chapter:

- A refresher on machine learning
- What is machine learning on graphs and why is it important?
- A general taxonomy to navigate among graph machine learning algorithms

Technical requirements

We will be using Jupyter notebooks with *Python 3.8* 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
node2vec==0.3.3
karateclub==1.0.19
scipy==1.6.2
```

All the code files relevant to this chapter are available at `https://github.com/PacktPublishing/Graph-Machine-Learning/tree/main/Chapter02`.

Understanding machine learning on graphs

Of the branches of artificial intelligence, **machine learning** is one that has attracted the most attention in recent years. It refers to a class of computer algorithms that automatically learn and improve their skills through experience *without being explicitly programmed*. Such an approach takes inspiration from nature. Imagine an athlete who faces a novel movement for the first time: they start slowly, carefully imitating the gesture of a coach, trying, making mistakes, and trying again. Eventually, they will improve, becoming more and more confident.

Now, how does this concept translate to machines? It is essentially an optimization problem. The goal is to find a mathematical model that is able to achieve the best possible performance on a particular task. Performance can be measured using a specific performance metric (also known as a **loss function** or **cost function**). In a common learning task, the algorithm is provided with data, possibly lots of it. The algorithm uses this data to iteratively make decisions or predictions for the specific task. At each iteration, decisions are evaluated using the loss function. The resulting *error* is used to update the model parameters in a way that, hopefully, means the model will perform better. This process is commonly called **training**.

More formally, let's consider a particular task, T, and a performance metric, P, which allows us to quantify how good an algorithm is performing on T. According to Mitchell (Mitchell et al., 1997), an algorithm is said to learn from experience, E, if its performance at task T, measured by P, improves with experience E.

Basic principles of machine learning

Machine learning algorithms fall into three main categories, known as *supervised*, *unsupervised*, and *semi-supervised* learning. These learning paradigms depend on the way data is provided to the algorithm and how performance is evaluated.

Supervised learning is the learning paradigm used when we know the answer to the problem. In this scenario, the dataset is composed of samples of pairs of the form <*x*,*y*>, where *x* is the input (for example, an image or a voice signal) and *y* is the corresponding desired output (for example, what the image represents or what the voice is saying). The input variables are also known as *features*, while the output is usually referred to as *labels*, *targets*, and *annotations*. In supervised settings, performance is often evaluated using a *distance function*. This function measures the differences between the prediction and the expected output. According to the type of labels, supervised learning can be further divided into the following:

- **Classification**: Here, the labels are discrete and refer to the "class" the input belongs to. Examples of classification are determining the object in a photo or predicting whether an email is spam or not.
- **Regression**: The target is continuous. Examples of regression problems are predicting the temperature in a building or predicting the selling price of any particular product.

Unsupervised learning differs from supervised learning since the answer to the problem is not known. In this context, we do not have any labels and only the inputs, <*x*>, are provided. The goal is thus deducing structures and patterns, attempting to find similarities.

Discovering groups of similar examples (clustering) is one of these problems, as well as giving new representations of the data in a high-dimensional space.

In **semi-supervised learning**, the algorithm is trained using a combination of labeled and unlabeled data. Usually, to direct the research of structures present in the unlabeled input data, a limited amount of labeled data is used.

It is also worth mentioning that **reinforcement learning** is used for training machine learning models to make a sequence of decisions. The artificial intelligence algorithm faces a game-like situation, getting *penalties* or *rewards* based on the actions performed. The role of the algorithm is to understand how to act in order to maximize rewards and minimize penalties.

Minimizing the error on the training data is not enough. The keyword in machine learning is *learning*. It means that algorithms must be able to achieve the same level of performance even on unseen data. The most common way of evaluating the generalization capabilities of machine learning algorithms is to divide the dataset into two parts: the **training set** and the **test set**. The model is trained on the training set, where the loss function is computed and used to update the parameters. After training, the model's performance is evaluated on the test set. Moreover, when more data is available, the test set can be further divided into **validation** and **test** sets. The validation set is commonly used for assessing the model's performance during training.

When training a machine learning algorithm, three situations can be observed:

- In the first situation, the model reaches a low level of performance over the training set. This situation is commonly known as **underfitting**, meaning that the model is not powerful enough to address the task.
- In the second situation, the model achieves a high level of performance over the training set but struggles at generalizing over testing data. This situation is known as **overfitting**. In this case, the model is simply memorizing the training data, without actually understanding the true relations among them.
- Finally, the ideal situation is when the model is able to achieve (possibly) the highest level of performance over both training and testing data.

An example of overfitting and underfitting is given by the risk curve shown in *Figure 2.1*. From the figure, it is possible to see how the performances on the training and test sets change according to the complexity of the model (the number of parameters to be fitted):

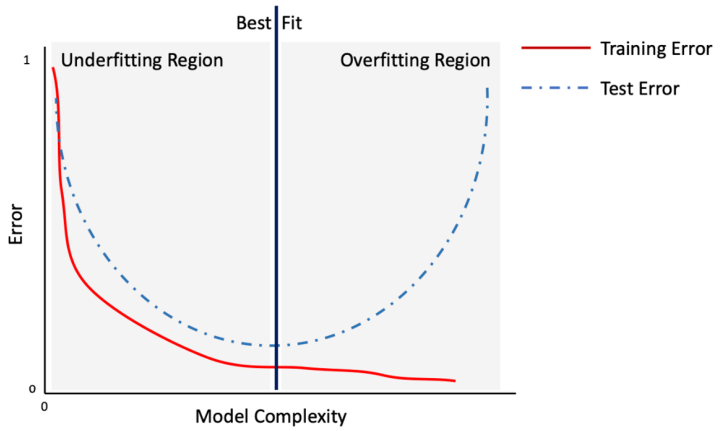

Figure 2.1 – Risk curve describing the prediction error on training and test set error in the function of the model complexity (number of parameters of the model)

Overfitting is one of the main problems that affect machine learning practitioners. It can occur due to several reasons. Some of the reasons can be as follows:

- The dataset can be ill-defined or not sufficiently representative of the task. In this case, adding more data could help to mitigate the problem.
- The mathematical model used for addressing the problem is too powerful for the task. In this case, proper constraints can be added to the loss function in order to reduce the model's "power." Such constraints are called **regularization** terms.

Machine learning has achieved impressive results in many fields, becoming one of the most diffused and effective approaches in computer vision, pattern recognition, and natural language processing, among others.

The benefit of machine learning on graphs

Several machine learning algorithms have been developed, each with its own advantages and limitations. Among those, it is worth mentioning regression algorithms (for example, linear and logistic regression), instance-based algorithms (for example, k-nearest neighbor or support vector machines), decision tree algorithms, Bayesian algorithms (for example, naïve Bayes), clustering algorithms (for example, k-means), and artificial neural networks.

But what is the key to all of this success?

Essentially, one thing: machine learning can automatically address tasks that are easy for humans to do. These tasks can be too complex to describe using traditional computer algorithms and, in some cases, they have shown even better capabilities than human beings. This is especially true when dealing with graphs—they can differ in several more ways than an image or audio signal because of their complex structure. By using graph machine learning, we can create algorithms to automatically detect and interpret recurring latent patterns.

For these reasons, there has been an increasing interest in *learning representations* for graph-structured data and many machine learning algorithms have been developed for handling graphs. For example, we might be interested in determining the role of a protein in a biological interaction graph, predicting the evolution of a collaboration network, recommending new products to a user in a social network, and many more (we will discuss these and more applications in *Chapter 10*, *The Future of Graphs*).

Due to their nature, graphs can be analyzed at different levels of granularity: at the node, edge, and graph level (the whole graph), as depicted in *Figure 2.2*. For each of those levels, different problems could be faced and, as a consequence, specific algorithms should be used:

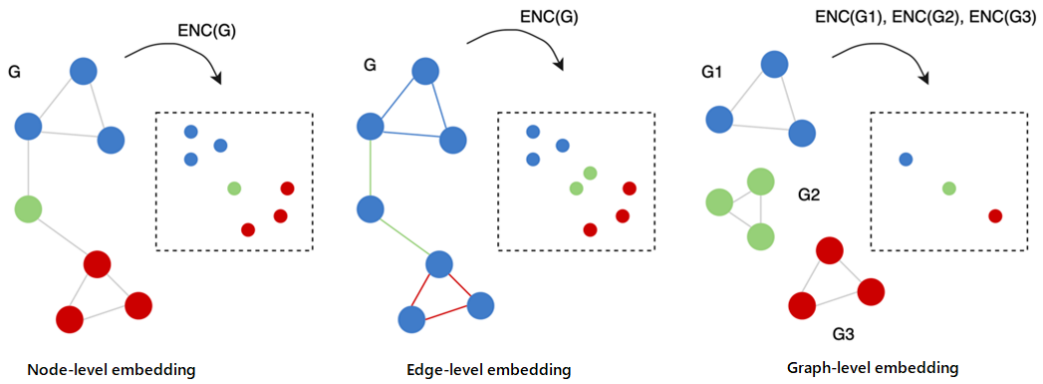

Figure 2.2 – Visual representation of the three different levels of granularity in graphs

In the following bullet points, we will give some examples of machine learning problems that could be faced for each of those levels:

- **Node level**: Given a (possibly large) graph, $G = (V, E)$, the goal is to classify each vertex, $v \in V$, into the right class. In this setting, the dataset includes *G* and a list of pairs, < *vi*,*yi* >, where *vi* is a node of graph *G* and *yi* is the class to which the node belongs.
- **Edge level**: Given a (possibly large) graph, $G = (V, E)$, the goal is to classify each edge, $e \in E$, into the right class. In this setting, the dataset includes *G* and a list of pairs, < *ei*,*yi* >, where *ei* is an edge of graph *G* and *yi* is the class to which the edge belongs. Another typical task for this level of granularity is **link prediction**, the problem of predicting the existence of a link between two existing nodes in a graph.
- **Graph level**: Given a dataset with *m* different graphs, the task is to build a machine learning algorithm capable of classifying a graph into the right class. We can then see this problem as a classification problem, where the dataset is defined by a list of pairs, <*Gi*,*yi*>, where *Gi* is a graph and *yi* is the class the graph belongs to.

In this section, we discussed some basic concepts of machine learning. Moreover, we have enriched our description by introducing some of the common machine learning problems when dealing with graphs. Having those theoretical principles as a basis, we will now introduce some more complex concepts relating to graph machine learning.

The generalized graph embedding problem

In classical machine learning applications, a common way to process the input data is to build from a set of features, in a process called **feature engineering**, which is capable of giving a compact and meaningful representation of each instance present in the dataset.

The dataset obtained from the feature engineering step will be then used as input for the machine learning algorithm. If this process usually works well for a large range of problems, it may not be the optimal solution when we are dealing with graphs. Indeed, due to their well-defined structure, finding a suitable representation capable of incorporating all the useful information might not be an easy task.

The first, and most straightforward, way of creating features capable of representing structural information from graphs is the *extraction of certain statistics*. For instance, a graph could be represented by its degree distribution, efficiency, and all the metrics we described in the previous chapter.

A more complex procedure consists of applying specific kernel functions or, in other cases, engineering-specific features that are capable of incorporating the desired properties into the final machine learning model. However, as you can imagine, this process could be really time-consuming and, in certain cases, the features used in the model could represent just a subset of the information that is really needed to get the best performance for the final model.

In the last decade, a lot of work has been done in order to define new approaches for creating meaningful and compact representations of graphs. The general idea behind all these approaches is to create algorithms capable of *learning* a good representation of the original dataset such that geometric relationships in the new space reflect the structure of the original graph. We usually call the process of learning a good representation of a given graph **representation learning** or **network embedding**. We will provide a more formal definition as follows.

Representation learning (**network embedding**) is the task that aims to learn a mapping function, $f: G \rightarrow \mathbb{R}^n$, from a discrete graph to a continuous domain. Function f will be capable of performing a low-dimensional vector representation such that the properties (local and global) of graph G are preserved.

Once mapping f is learned, it could be applied to the graph and the resulting mapping could be used as a feature set for a machine learning algorithm. A graphical example of this process is visible in *Figure 2.3*:

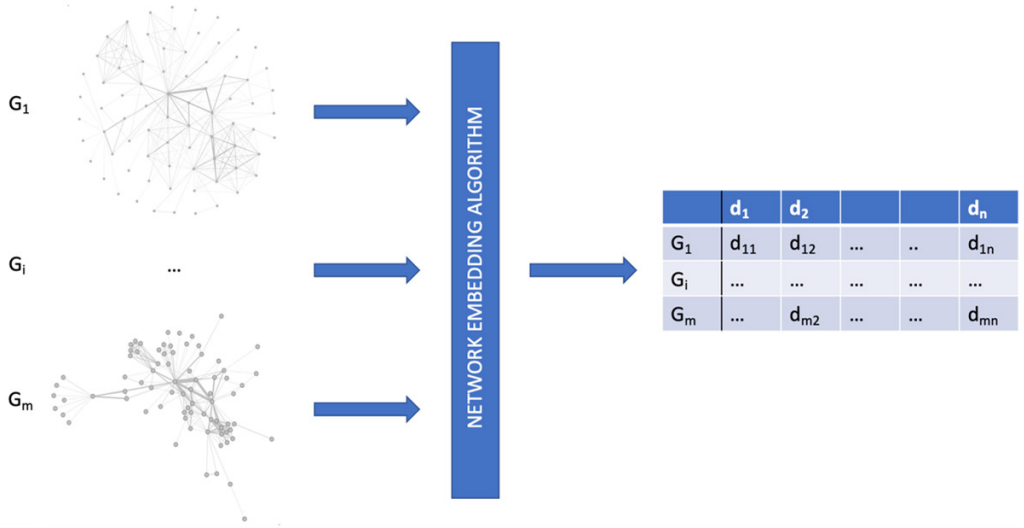

Figure 2.3 – Example of a workflow for a network embedding algorithm

Mapping function f can also be applied in order to learn the vector representation for nodes and edges. As we already mentioned, machine learning problems on graphs could occur at different levels of granularity. As a consequence, different embedding algorithms have been developed in order to learn functions to generate the vectorial representation of nodes ($f: V \rightarrow \mathbb{R}^n$) (also known as **node embedding**) or edges ($f: E \rightarrow \mathbb{R}^n$) (also known as **edge embedding**). Those mapping functions try to build a vector space such that the geometric relationships in the new space reflect the structure of the original graph, node, or edges. As a result, we will see that graphs, nodes, or edges that are similar in the original space will also be similar in the new space.

In other words, in the space generated by the embedding function, similar structures will have *a small Euclidean distance*, while dissimilar structures will have *a large Euclidean distance*. It is important to highlight that while most embedding algorithms generate a mapping in Euclidean vector spaces, there has recently been an interest in non-Euclidean mapping functions.

Let's now see a practical example of what an embedding space looks like, and how similarity can be seen in the new space. In the following code block, we show an example using a particular embedding algorithm known as **Node to Vector** (**Node2Vec**). We will describe how it works in the next chapter. At the moment, we will just say that the algorithm will map each node of graph *G* in a vector:

```
import networkx as nx
from node2vec import Node2Vec
import matplotlib.pyplot as plt

G = nx.barbell_graph(m1=7, m2=4)
node2vec = Node2Vec(G, dimensions=2)
model = node2vec.fit(window=10)

fig, ax = plt.subplots()
for x in G.nodes():
    v = model.wv.get_vector(str(x))
    ax.scatter(v[0],v[1], s=1000)
    ax.annotate(str(x), (v[0],v[1]), fontsize=12)
```

In the preceding code, we have done the following:

1. We generated a barbell graph (described in the previous chapter).
2. The Node2Vec embedding algorithm is then used in order to map each node of the graph in a vector of two dimensions.
3. Finally, the two-dimensional vectors generated by the embedding algorithm, representing the nodes of the original graph, are plotted.

The result is shown in *Figure 2.4*:

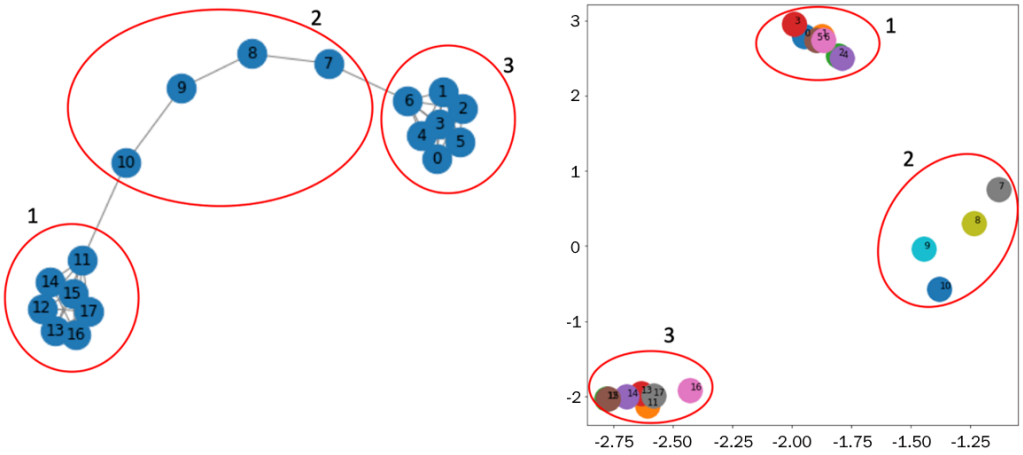

Figure 2.4 – Application of the Node2Vec algorithm to a graph (left) to generate the embedding vector of its nodes (right)

From *Figure 2.4*, it is easy to see that nodes that have a similar structure are close to each other and are distant from nodes that have dissimilar structures. It is also interesting to observe how good Node2Vec is at discriminating group 1 from group 3. Since the algorithm uses neighboring information of each node to generate the representation, the clear discrimination of those two groups is possible.

Another example on the same graph can be performed using the **Edge to Vector** (**Edge2Vec**) algorithm in order to generate a mapping for the edges for the same graph, *G*:

```
from node2vec.edges import HadamardEmbedder
edges_embs = HadamardEmbedder(keyed_vectors=model.wv)
fig, ax = plt.subplots()
for x in G.edges():
    v = edges_embs[(str(x[0]), str(x[1]))]
    ax.scatter(v[0],v[1], s=1000)
    ax.annotate(str(x), (v[0],v[1]), fontsize=12)
```

In the preceding code, we have done the following:

1. We generated a barbell graph (described in the previous chapter).
2. The `HadamardEmbedder` embedding algorithm is applied to the result of the Node2Vec algorithm (`keyed_vectors=model.wv`) used in order to map each edge of the graph in a vector of two dimensions.
3. Finally, the two-dimensional vectors generated by the embedding algorithm, representing the nodes of the original graph, are plotted.

The results are shown in *Figure 2.5*:

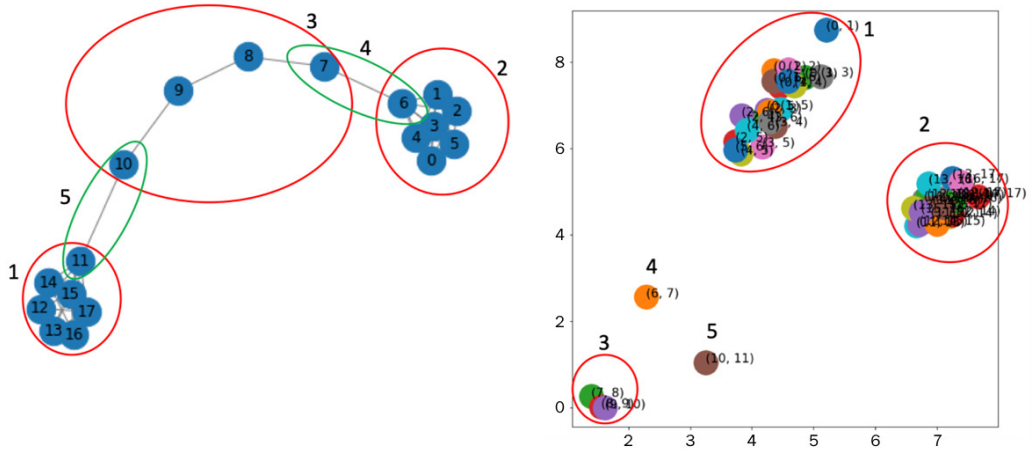

Figure 2.5 – Application of the Hadamard algorithm to a graph (left) to generate the embedding vector of its edges (right)

As for node embedding, in *Figure 2.5*, we reported the results of the edge embedding algorithm. From the figure, it is easy to see that the edge embedding algorithm clearly identifies similar edges. As expected, edges belonging to groups 1, 2, and 3 are clustered in well-defined and well-grouped regions. Moreover, the (6,7) and (10,11) edges, belonging to groups 4 and 5, respectively, are well clustered in specific groups.

Finally, we will provide an example of a **Graph to Vector** (**Grap2Vec**) embedding algorithm. This algorithm maps a single graph in a vector. As for another example, we will discuss this algorithm in more detail in the next chapter. In the following code block, we provide a Python example showing how to use the Graph2Vec algorithm in order to generate the embedding representation on a set of graphs:

```
import random
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()

fig, ax = plt.subplots(figsize=(10,10))
for i,vec in enumerate(embeddings):
    ax.scatter(vec[0],vec[1], s=1000)
    ax.annotate(str(i), (vec[0],vec[1]), fontsize=16)
```

In this example, the following has been done:

1. 20 Watts-Strogatz graphs (described in the previous chapter) have been generated with random parameters.
2. We have then executed the graph embedding algorithm in order to generate a two-dimensional vector representation of each graph.
3. Finally, the generated vectors are plotted in their Euclidean space.

The results of this example are shown in *Figure 2.6*:

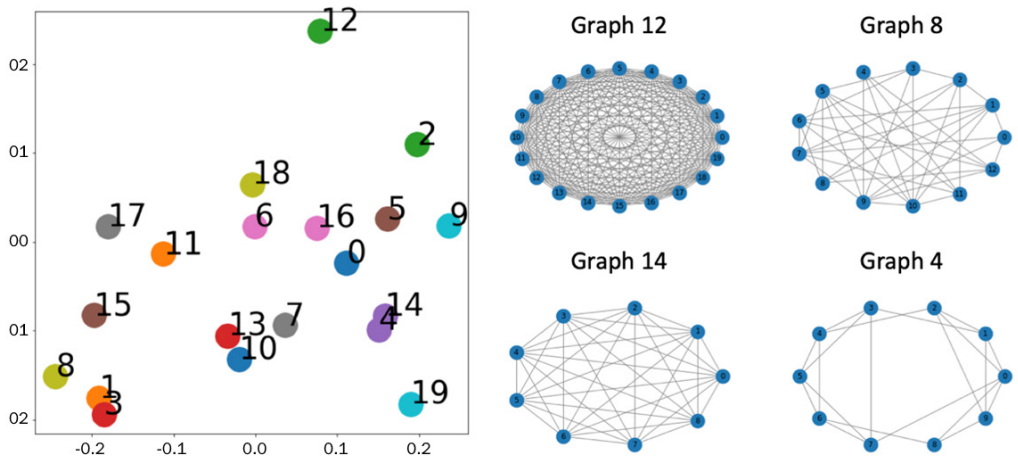

Figure 2.6 – Plot of two embedding vectors generated by the Graph2Vec algorithm applied to 20 randomly generated Watts-Strogatz graphs (left). Extraction of two graphs with a large Euclidean distance (Graph 12 and Graph 8 at the top right) and two graphs with a low Euclidean distance (Graph 14 and Graph 4 at the bottom right) is shown

As we can see from *Figure 2.6*, graphs with a large Euclidean distance, such as graph 12 and graph 8, have a different structure. The former is generated with the `nx.watts_strogatz_graph(20,20,0.2857)` parameter and the latter with the `nx.watts_strogatz_graph(13,6,0.8621)` parameter. In contrast, a graph with a low Euclidean distance, such as graph 14 and graph 8, has a similar structure. Graph 14 is generated with the `nx.watts_strogatz_graph(9,9,0.5091)` command, while graph 4 is generated with `nx.watts_strogatz_graph(10,5,0.5659)`.

In the scientific literature, a plethora of embedding methods has been developed. We will describe in detail and use some of them in the next section of this book. These methods are usually classified into two main types: *transductive* and *inductive*, depending on the update procedure of the function when new samples are added. If new nodes are provided, transductive methods update the model (for example, re-train) to infer information about the nodes, while in inductive methods, models are expected to generalize to new nodes, edges, or graphs that were not observed during training.

The taxonomy of graph embedding machine learning algorithms

A wide variety of methods to generate a compact space for graph representation have been developed. In recent years, a trend has been observed of researchers and machine learning practitioners converging toward a unified notation to provide a common definition to describe such algorithms. In this section, we will be introduced to a simplified version of the taxonomy defined in the paper *Machine Learning on Graphs: A Model and Comprehensive Taxonomy* (`https://arxiv.org/abs/2005.03675`).

In this formal representation, every graph, node, or edge embedding method can be described by two fundamental components, named the encoder and the decoder. The **encoder** (**ENC**) maps the input into the embedding space, while the **decoder** (**DEC**) decodes structural information about the graph from the learned embedding (*Figure 2.7*).

The framework described in the paper follows an intuitive idea: if we are able to encode a graph such that the decoder is able to retrieve all the necessary information, then the embedding must contain a compressed version of all this information and can be used to downstream machine learning tasks:

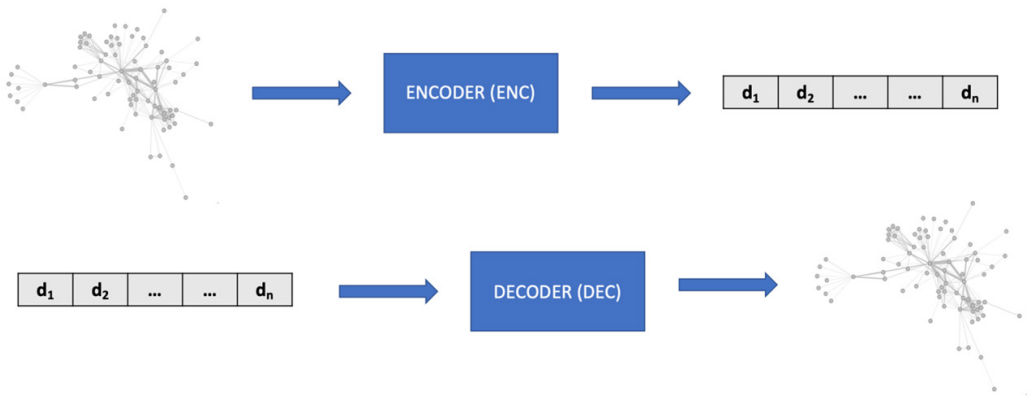

Figure 2.7 – Generalized encoder (ENC) and decoder (DEC) architecture for embedding algorithms

In many graph-based machine learning algorithms for representation learning, the decoder is usually designed to map pairs of node embeddings to a real value, usually representing the proximity (distance) of the nodes in the original graphs. For example, it is possible to implement the decoder such that, given the embedding representation of two nodes, $z_i = ENC(V_i)$ and $z_j = ENC(V_j)$, $DEC(z_i, z_j) = 1$ if in the input graph an edge connecting the two nodes, z_i, z_j, exists. In practice, more effective *proximity functions* can be used to measure the similarity between nodes.

The categorization of embedding algorithms

Inspired by the general framework depicted in *Figure 2.7*, we will now provide a categorization of the various embedding algorithms into four main groups. Moreover, in order to help you to better understand this categorization, we shall provide simple code snapshots in pseudo-code. In our pseudo-code formalism, we denote `G` as a generic `networkx` graph, with `graphs_list` as a list of `networkx` graphs and `model` as a generic embedding algorithm:

- **Shallow embedding methods**: These methods are able to learn and return only the embedding values for the learned input data. *Node2Vec*, *Edge2Vec*, and *Graph2Vec*, which we previously discussed, are examples of shallow embedding methods. Indeed, they can only return a vectorial representation of the data they learned during the *fit* procedure. It is not possible to obtain the embedding vector for unseen data. A typical way to use these methods is as follows:

```
model.fit(graphs_list)
embedding = model.get_embedding()[i]
```

  In the code, a generic shallow embedding method is trained on a list of graphs (line 1). Once the model is fitted, we can only get the embedding vector of the *i*th graph belonging to `graphs_list` (line 2). Unsupervised and supervised shallow embedding methods will be described, respectively, in *Chapter 3*, *Unsupervised Graph Learning*, and *Chapter 4*, *Supervised Graph Learning*.

- **Graph autoencoding methods**: These methods do not simply learn how to map the input graphs in vectors; they learn a more general mapping function, $f(G)$, capable of also generating the embedding vector for unseen instances. A typical way to use them is as follows:

```
model.fit(graphs_list)
embedding = model.get_embedding(G)
```

  The model is trained on `graphs_list` (line 1). Once the model is fitted on the input training set, it is possible to use it to generate the embedding vector of a new unseen graph, `G`. Graph autoencoding methods will be described in *Chapter 3*, *Unsupervised Graph Learning*.

- **Neighborhood aggregation methods**: These algorithms can be used to extract embeddings at the graph level, where nodes are labeled with some properties. Moreover, as for the graph autoencoding methods, the algorithms belonging to this class are able to learn a general mapping function, $f(G)$, also capable of generating the embedding vector for unseen instances.

A nice property of those algorithms is the possibility to build an embedding space where not only the internal structure of the graph is taken into account but also some external information, defined as properties of its nodes. For instance, with this method, we can have an embedding space capable of identifying, at the same time, graphs with similar structures and different properties on nodes. Unsupervised and supervised neighborhood aggregation methods will be described in *Chapter 3*, *Unsupervised Graph Learning*, and *Chapter 4*, *Supervised Graph Learning*, respectively.

- **Graph regularization methods**: Methods based on graph regularization are slightly different from the ones listed in the preceding points. Here, we do not have a graph as input. Instead, the objective is to learn from a set of features by exploiting their "interaction" to regularize the process. In more detail, a graph can be constructed from the features by considering feature similarities. The main idea is based on the assumption that nearby nodes in a graph are likely to have the same labels. Therefore, the loss function is designed to constrain the labels to be consistent with the graph structure. For example, regularization might constrain neighboring nodes to share similar embeddings, in terms of their distance in the L2 norm. For this reason, the encoder only uses X node features as input.

  The algorithms belonging to this family learn a function, $f(X)$, that maps a specific set of features (X) to an embedding vector. As for the graph autoencoding and neighborhood aggregation methods, this algorithm is also able to apply the learned function to new, unseen features. Graph regularization methods will be described in *Chapter 4*, *Supervised Graph Learning*.

For algorithms belonging to the group of shallow embedding methods and neighborhood aggregation methods, it is possible to define an *unsupervised* and *supervised* version. The ones belonging to graph autoencoding methods are suitable in unsupervised tasks, while the algorithms belonging to graph regularization methods are used in semi-supervised/supervised settings.

For unsupervised algorithms, the embedding of a specific dataset is performed only using the information contained in the input dataset, such as nodes, edges, or graphs. For the supervised setting, external information is used to guide the embedding process. That information is usually classed as a label, such as a pair, <*Gi,yi*>, that assigns to each graph a specific class. This process is more complex than the unsupervised one since the model tries to find the best vectorial representation in order to find the best assignment of a label to an instance. In order to clarify this concept, we can think, for instance, of the *convolutional neural networks* for image classification. During their training process, neural networks try to classify each image into the right class by performing the fitting of various convolutional filters at the same time. The goal of those convolutional filters is to find a compact representation of the input data in order to maximize the prediction performances. The same concept is also valid for supervised graph embedding, where the algorithm tries to find the best graph representation in order to maximize the performance of a class assignment task.

From a more mathematical perspective, all these models are trained with a proper loss function. This function can be generalized using two terms:

- The first is used in supervised settings to minimize the difference between the prediction and the target.
- The second is used to evaluate the similarity between the input graph and the one reconstructed after the ENC + DEC steps (which is the structure reconstruction error).

Formally, it can be defined as follows:

$$Loss = \alpha L_{sup}(y, \hat{y}) + L_{rec}(G, \hat{G})$$

Here, $\alpha L_{sup}(y, \hat{y})$ is the loss function in the supervised settings. The model is optimized to minimize, for each instance, the error between the right (y) and the predicted class ($\hat{y}$). $L_{rec}(G, \hat{G})$ is the loss function representing the reconstruction error between the input graph (G) and the one obtained after the ENC + DEC process ($\hat{G}$). For unsupervised settings, we have the same loss but $\alpha = 0$, since we do not have a target variable to use.

It is important to highlight the main role that these algorithms play when we try to solve a machine learning problem on a graph. They can be used *passively* in order to transform a graph into a feature vector suitable for a classical machine learning algorithm or for data visualization tasks. But they can also be used *actively* during the learning process, where the machine learning algorithm finds a compact and meaningful solution to a specific problem.

Summary

In this chapter, we refreshed some basic *machine learning* concepts and discovered how they can be applied to graphs. We defined basic *graph machine learning* terminology with a particular focus on *graph representation learning*. A taxonomy of the main graph machine learning algorithms was presented in order to clarify what differentiates the various ranges of solutions developed over the years. Finally, practical examples were provided to begin understanding how the theory can be applied to practical problems.

In the next chapter, we will revise the main graph-based machine learning algorithms. We will analyze their behavior and see how they can be used in practice.