6 GRAPH EMBEDDINGS

6.1 GRAPH EMBEDDINGS

As we have seen all along this course, graphs are a very powerful data representation tool, capable of properly modeling complex interaction patterns among the items of a dataset. However, this complexity makes any operation on graphs intimately hard to even define. Think for instance for instance of the problem of defining the distance between two nodes, or two graphs or a concept of continuity on graph: all these problem have been approached and have solutions, but they are not unique and they need to be defined. This is because graphs live in a high dimensional non-Euclidean space in which most mathematical operations are not easily defined. Consequently, a powerful method to deal with graph is to project them into an Euclidean space. This operation is called *embedding* and *Spectral clustering* (SC) actually exploits an embedding that leverages on an appropriate graph matrix representation. Now, there is not a unique way to embed graphs and the embedding itself should be defined so that it preserves some relevant graph properties. Nonetheless, once it is defined, we move to a space in which several mathematical operations and algorithms (such as clustering) can be deployed. We can formally define a node embedding as follows

Embeddings allow one to more simply represent graphs

Node embeddings are at the basis of graph neural networks and allow one to provide meaningful representations of complex objects. In the remainder we describe the `Node2Vec` algorithm, that is one of the most popular algorithms to obtain non-linear node embeddings. Note that this embedding method can also be used to perform *Community detection* (CD) as well, by performing clustering on the embedded space, equivalently to SC.

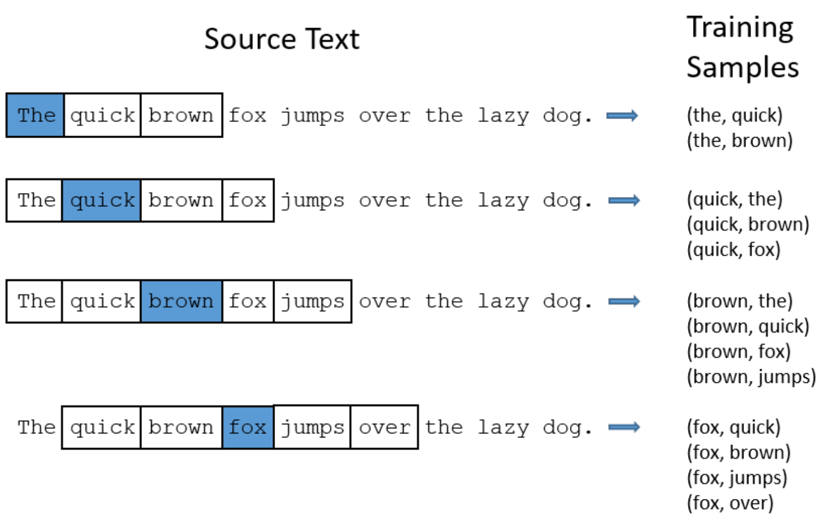

Figure 6.1: **Visual representation of the Skip-Gram algorithm**. The central word is highlighted in blue, while the context words are surrounded by white boxes. In this example the window size is set equal to 2. On the right we have all the pairs (central, context) that are obtained scanning through the document that constitute the output of the Skip-Gram algorithm. Picture taken from mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/.

6.2 WORD2VEC

The `Node2Vec` algorithm builds on another (even more popular) algorithm, called `Word2Vec`. This algorithm was introduced to embed the words appearing in a text and capturing the semantic similarity between them. For instance, if $i =$ milk, $j =$ cow, $k =$ gun, for the corresponding embedding vectors, we expect that $\boldsymbol{x}_i^T \boldsymbol{x}_j$ is large (milk is related to cow), while $\boldsymbol{x}_i^T \boldsymbol{x}_k$ is small. Let us now detail the method to obtain this desiderata.

6.2.1 SKIP-GRAM

Defining context words

If our goal is to provide similar representations to words that appear in similar contexts, we must first define what contexts are and we must do so in a simple, content-agnostic way. To do so, the skip-gram algorithm takes an arbitrary word of the text and considers the surrounding ones (within a certain distance, called *window size*) as the context of that word. To give an increasing weight to the words that are closer to the central one, one may draw for each word the window size from a uniform distribution between 1 and a maximal value. The Skip-Gram algorithm then takes a text as input and a value of the (maximal) window size and outputs a list of pairs $(\mathrm{center}, \mathrm{context})$ that relate every words appearing in the text with the surrounding ones. Figure 6.1 depicts the Skip-Gram procedure. If two words a, b are closely related – such as *gold* and *crown* –, one expects the pair (a, b) to appear several times. However, thinking of the case of synonyms, one can expect them to rarely appear in the same context, even if they have the same meaning. The Skip-Gram algorithm, however, can properly deal also with this type of similarity because synonyms will be surrounded by similar context words, thus allowing one to recover their similarity.

6.2.2 DEFINITION OF A LOSS FUNCTION

Now that we have identified context words, we want to define a loss function of the embedding vectors that promoted the alignment of for the pairs (central, context). Let i be an index running over all words in the text and let $\pi(i)$ be a function mapping word i to its position in the dictionary. Then, letting $\mathcal{C}_i$ be the context of word i, we write

$$\mathcal{L} = -\sum_{i\in\mathcal{T}}\sum_{j\in\mathcal{C}_i}\log\sigma\left(\boldsymbol{x}_{\pi(i)}^T\boldsymbol{x}_{\pi(j)}\right),$$

where $\mathcal{T}$ is the set of words appearing in the text, $\sigma(\cdot)$ is the sigmoid function[1] and $\boldsymbol{x}_a \in \mathbb{R}^d$ is the embedding of the word a. Now, minimizing this loss function we promote the alignment between central and context words. This loss function actually has a trivial minimum that is obtained for $\boldsymbol{x}_a = \mathbf{1}_d$ for all words a. This because it lack an *adversarial* term, like the one appearing in the modularity cost function.

We add this term with a technique called *negative sampling*. For each word $i \in \mathcal{T}$, we sample a set of $\mathcal{R}_i$ of random words sampled from the text and write the following loss function

Negative sampling

$$\mathcal{L} = -\sum_{i\in\mathcal{T}}\left[\sum_{j\in\mathcal{C}_i}\log\sigma\left(\boldsymbol{x}_{\pi(i)}^T\boldsymbol{x}_{\pi(j)}\right) + \sum_{j\in\mathcal{R}_i}\log\sigma\left(-\boldsymbol{x}_{\pi(i)}^T\boldsymbol{x}_{\pi(j)}\right)\right]. \qquad (6.1)$$

This newly added term takes random pairs of words and gives a gain in the loss function when they are misaligned, thus preventing the trivial minimum. We now detail the strategy to optimize this cost function.

6.2.3 TRAINING THE MODEL PARAMETERS

The cost function is optimized with stochastic gradient descent and back-propagation. This is a modified version of gradient descent that is a method to optimize a multivariate function. Given a random argument of the function to optimize, the idea is to move in the direction of the negative gradient of the loss function, as depicted in Figure 6.2. This means

$$\boldsymbol{x}_{\text{new}} = \boldsymbol{x}_{\text{old}} - \eta\nabla\mathcal{L}(\boldsymbol{x}_{\text{old}}),$$

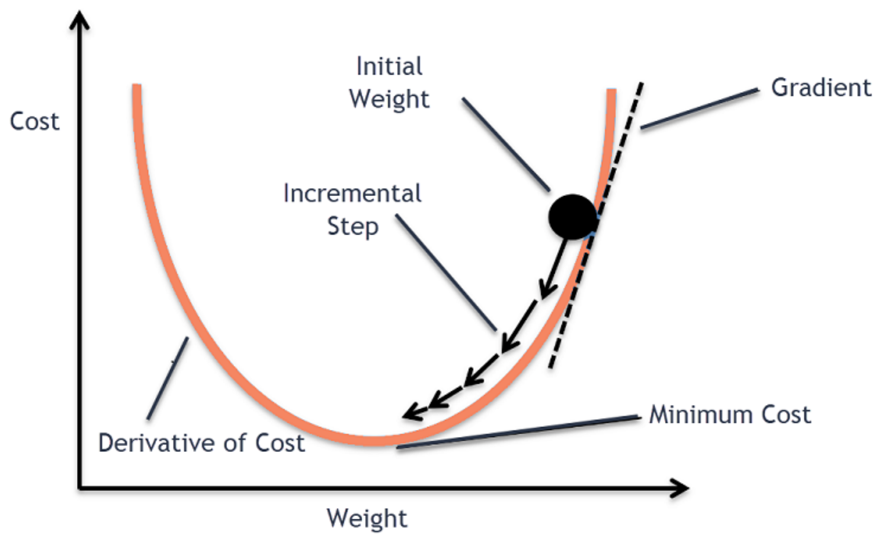

Figure 6.2: **Visualization of gradient descent**. Picture taken from https://www.analyticsvidhya.com/blog/2020/10/how-does-the-gradient-descent-algorithm-work-in-machine-learning/.

where $\eta > 0$ is the learning parameter. Now, computing the gradient may be unpractical. To simplify things we let $\mathcal{L} = \sum_{i \in \mathcal{T}} \mathcal{L}_i$ and we update the weight $\boldsymbol{x}_i$ with the value of the gradient of $\mathcal{L}_i$ (and not of the whole function $\mathcal{L}$) with respect to $\boldsymbol{x}_i$.

$$\frac{\partial \mathcal{L}_i}{\partial x_{\pi(i)a}} = -\left[\sum_{j \in \mathcal{C}_i} \frac{\sigma'\left(\boldsymbol{x}_{\pi(i)}^T \boldsymbol{x}_{\pi(j)}\right)}{\sigma\left(\boldsymbol{x}_{\pi(i)}^T \boldsymbol{x}_{\pi(j)}\right)} x_{\pi(j)a} - \sum_{j \in \mathcal{R}_i} \frac{\sigma'\left(-\boldsymbol{x}_{\pi(i)}^T \boldsymbol{x}_{\pi(j)}\right)}{\sigma\left(-\boldsymbol{x}_{\pi(i)}^T \boldsymbol{x}_{\pi(j)}\right)} x_{\pi(j)a} \right], \tag{6.2}$$

where σ' denotes the derivative of the sigmoid function. One can easily verify that the following relation holds

$$\sigma'(x) = \sigma(x)\sigma(-x).$$

Plugging these relations in Equation (6.2) we obtain

Stochastic gradient descent

$$\begin{aligned} g_{\pi(i)a} &= \frac{\partial \mathcal{L}}{\partial x_{\pi(i)a}} \\ &= -\left[\sum_{j \in \mathcal{C}_i} \sigma\left(-\boldsymbol{x}_{\pi(i)}^T \boldsymbol{x}_{\pi(j)}\right) x_{\pi(i)a} + \sum_{j \in \mathcal{R}_i} \sigma\left(\boldsymbol{x}_{\pi(i)}^T \boldsymbol{x}_{\pi(j)}\right) x_{\pi(j)a} \right]. \end{aligned} \tag{6.3}$$

By iteratively updating the weights, we find an approximation of a minimum of the loss function $\mathcal{L}$ that provides us with a good representation of the words. Algorithm 6.1 summarizes the `Word2Vec` algorithm.

1 The sigmoid function is $\sigma(x) = (1 + e^{-x})^{-1}$. This is an increasing function, bounded between 0 and 1.

Algorithm 6.1 : `Word2Vec`

```
Input : Text $\mathcal{T}$ with $N$ words, dictionary with $n$ words, embedding
        dimension $d$, number of training epochs $\mathrm{n_{epochs}}$, learning
        rate $\eta$, window size $\omega$, number of negative samples $m$
Output : $\{x_a\}_{a\in[n]}$ word embedding vectors in $\mathbb{R}^d$
1  begin
2      Randomly intialize $x_a$ for all $a \in [n]$ ;
3      for epoch $= 1, \ldots, \mathrm{n_{epochs}}$ do
4          for $i \in \mathcal{T}$ do
5              Draw $w$ uniformy at random $w \leftarrow \mathcal{U}(1, \omega)$;
6              Get $\mathcal{C}_i$ for window width $w$ with Skip-Gram;
7              Get $\mathcal{R}_i$ selecting $m$ random words from the text;
8              Compute $g_{\pi(i)}$ as per Equation (6.2);
9              Update $x_{\pi(i)} \leftarrow x_{\pi(i)} - \eta g_{\pi(i)}$
10         end
11     end
12     return $\{x_a\}_{a\in[n]}$
13 end
```

6.3 NODE2VEC

Let us now go back to graphs. The `Node2Vec` algorithm uses the `Word2Vec` algorithm to obtain a node embedding by first "translating" the graph into a text and then embedding its words, corresponding to the graph nodes. The text is obtained performing random walks on the graph. Random walks are are paths made of sequences of adjacent nodes of the type $(v_1, v_2, v_3, \ldots, v_T)$, where T here denotes the walk length. Neighboring nodes are in contact and, in some sense, belong to the same context. Random walks are hence use to probe the network structure and to extract information out of it. The strategy is to define random walks with memory, with the definition of two parameters.

Translating a graph into a text

Suppose that the walker moves from i to j then it is at distance 1 from i. By performing a further step there are three possibilities: moving to a node that is at distance 2 from i, moving to a node that is at distance 1 from i or going back to i itself. These three options are taken with different probabilities that are proportional to $1/p$, $1/q$ and 1 respectively. The values of p and q are an input of the algorithm. For $p = q = 1$ we have a simple random walk without memory. The walking strategy is displayed in Figure 6.3. How to choose the parameters p and q? The value of p determines the probability of immediately returning to node the walker came from. Choosing large values of p thus prevents the walker from bouncing back and forth between the same nodes and instead it encourages faster exploration of the network. This is exactly what happens with non-backtracking random walks and it is particularly useful for sparse graphs in which, given that each node has very

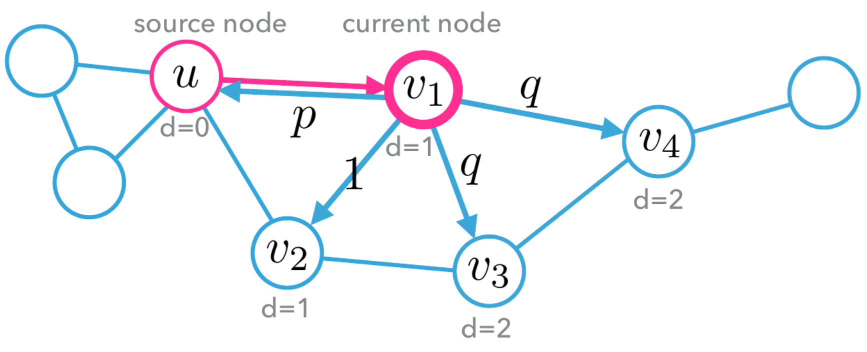

Figure 6.3: **Visualization of the random walk strategy of Node2Vec**. In the last step the random walker moved from u to v_1. In the next step it will move: i) back to u with a probability proportional to $1/p$; ii) to v_3 or v_4 with a probability proportional to $1/q$, since they are at a distance 2 from u; iii) to v_2 with a probability proportional to 1 since v_1 is at a distance 1 from both u and v_1.

few neighbors, it is likely to move back to the node of origin. The value of q determines to what extent the random walker is inclined to visit nodes that are further away from the original one. In particular, if $q > 1$ the walker is more inclined to remain close to u. This kind of transition accounts for the triangles that may be present in the network and attributes a higher chance to visit nodes that are tightly connected among them. On the other hand, small values of q will lead to avoid these paths and to more rapidly explore the rest of the network. This has a direct impact on the type of information stored in the embedding, as shown in Figure 6.4. This plot shows the result of clustering based on the embedding obtained on the same graph for $p = 1$ and $q = 0.5, 2$, respectively. For $q = 0.5$ the algorithm is more prone to find tightly connected communities, while for $q = 2$ it groups the node according to structural equivalence.

6.4 CONCLUSION

Graph embedding methods are very powerful to analyze and represent relational data. In this chapter we introduced the `Node2Vec` algorithm that is one of the most influential methods developed in the past 10 years. Given its dependence on `Word2Vec`, it is rather fast and its complexity scales as $\mathcal{O}(|\mathcal{E}|md\omega)$. Moreover, note that, similarly to `Node2Vec`, a wealth of algorithms have explored similar strategies to exploit the `Word2Vec` algorithm and provide meaningful representations to complex mathematical objects. You should then see this as a relevant example of a widely used pipeline to define representation learning algorithms.

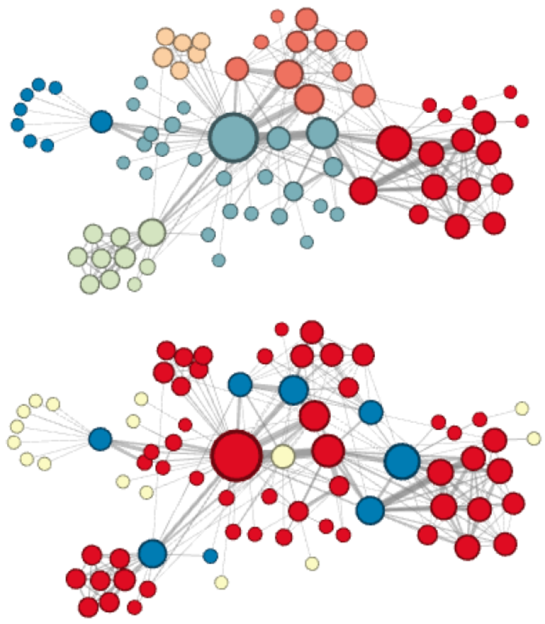

Figure 6.4: **Color coding according to the embedding obtained with two different values of q of the Node2Vec algorithm**. The input graph shows the co-appearances of characters in *Les Misérables*. For the top plot $q = 0.5$, while for the bottom plot $q = 2$. Picture taken from *Grover, Leskovec, node2vec: Scalable Feature Learning for Networks.*

6.5 REFERENCES

- Mikolov, Sutskever, Chen, Corrado, Dean: *Distributed representations of words and phrases and their compositionality*
  This is the paper in which `Word2Vec` was introduced

- Grover, Leskovec: *node2vec: Scalable feature learning for networks*
  This is the paper in which `Node2Vec` was introduced