Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Dipartimento di Fisica e Astronomia
Corso di Laurea in Fisica

Spectral Methods in Geometric Deep Learning

Relatore:
Prof.ssa Rita Fioresi

Presentata da:
Alessandro Azzani

Anno Accademico 2021/2022

Abstract

Questa tesi propone una panoramica sul funzionamento interno delle architetture alla base del deep learning e in particolare del geometric deep learning. Iniziando a discutere dalla storia degli algoritmi di intelligenza artificiale, vengono introdotti i principali costituenti di questi. In seguito vengono approfonditi alcuni elementi della teoria dei grafi, in particolare il concetto di laplaciano discreto e il suo ruolo nello studio del fenomeno di diffusione sui grafi. Infine vengono presentati alcuni algoritmi utilizzati nell'ambito del geometric deep learning su grafi per la classificazione di nodi. I concetti discussi vengono poi applicati nella realizzazione di un'architettura in grado di classficiare i nodi del dataset Zachary Karate Club.

Abstract

This thesis gives an overview of the inner functioning of the architectures underlying deep learning and in particular geometric deep learning. Starting with the discussion on the history of artificial intelligence algorithms, the main ingredients of deep learning algorithms are introduced. Subsequently, some elements of graph theory are discussed, in particular the concept of discrete Laplacian and its role in the study of the diffusion phenomenon on graphs. Finally, some algorithms used in the field of geometric deep learning on graphs for node classification are presented. The concepts discussed are then applied in the creation of an architecture capable of classifying the nodes of the Zachary Karate Club dataset.

Contents

Introduction

The new machine learning technologies, and in particular deep learning ([1]) or convolutional neural networks ([2]), have achieved unprecedented results in some applications such as object detection ([3]) or image classification ([4]). Although deep learning is a very successful algorithm in processing traditional data such as images, sounds or texts, with the emergence of larger data scale and more powerful GPU computing ability, people begin to wonder how to process other type of data organized in a less ordered form. This application of machine learning led to another family of algorithms inspired by deep learning called *geometric deep learning*. It operates on data organized as graphs, this type of data arise in numerous applications. For instance, in social networks a graph can be used to represent the relationship between people ([5]). Furthermore, in neuroscience, graph models are used to represent anatomical structures of the brain. These types of data are arranged in more involved forms, which makes it difficult to define convolution operations on them and, thus, they cannot be processed through “normal” artificial or convolutional neural networks. On the other hand, grid-like data can be regarded as special graph data and therefore fed to a graph neural network. Geometric deep learning algorithms are more “flexible”, since they can work on less structured data. This makes this new filed of great interest nowadays and with the prospect of finding exciting new applications in the next years.

In this thesis, we will start with a brief journey on the history of artificial neural networks. We will begin with the introduction of the *perceptron*, which allowed the invention of multi-layer perceptrons, convolutional neural networks and ultimately leading to the evolution of such algorithms to operate on graphs. Then, we will discuss the main ingredients characterizing neural networks: we will study the functioning of the perceptron, how to work with datasets and we will present the key elements at the foundation of the learning process. We will examine the different kind of loss functions used in applications. We will then end the first chapter with a brief discussion on the difference between a neural network and the human brain.

We will then focus on graphs in the second chapter. We will present the definition of graph and some related concepts like the adjacency matrix. We will find the analogues of the gradient operator and the divergence operator on a graph and we will use them to find the *graph Laplacian*. We will discuss some properties of this operator and its eigenvalues and eigenfunctions. We will then use the graph Laplacian to study the diffusion mechanism of a signal on a graph and to find the discrete analogous of the heat diffusion equation.

Finally, in the last chapter, we will introduce some of the methods developed for the task of node classification on a graph. We will discuss the encoder-decoder framework, which is the fundation for building geometric convolutional networks. We will implement these methods through graphSAGE convolutional networks, which are based on the concept of message passing. We will also highlight how the process of message passing on a graph is similar to the diffusion of heat. Then, we will present graph attention networks which associate an attention coefficient to every link of the graph, encoding the importance of the link itself. At the end, we will see a concrete example by implementing a graphSAGE convolutional network to classify the nodes in the Zachary Karate Club dataset.

Chapter 1

Neural Networks

In this chapter we give a brief account of neural networks. After some historical remarks, we describe the main concepts lying at the foundations of neural networks architecture like the loss functions, the linear classifier and the stochastic gradient descent. We also introduce Graph Neural Networks (GNNs) that will be however studied in detail in the last chapter.

1.1 History of Artificial Neural Networks

The developing and study of deep learning architectures originates from our understanding of the mechanisms behind the human brain and our cognitive system. In this section, we give a brief history of deep learning, from neural networks and convolutional neural networks, up to the modern graph neural networks.

Donald O. Hebb is considered the father of Neural Networks (NNs) because of the introduction of the *Hebbian Learning Rule* in 1949 [8], which states: "Cells that fire together, wire together", which means that the connection between two neurons is strengthened when these two neurons fire simultaneously. This rule can be written mathematically as:

$$\Delta w_i = \eta x_i y \tag{1.1}$$

where Δw_i represents the change of the synaptic strength w_i of the i-th neuron, x_i is the input signal, y is the postsynaptic response e finally η denotes the learning rate, that we will study later on. However, this model has a drawback: as co-occurrences appear more, the weights keep increasing and eventually grow exponentially. To avoid this behaviour, Erkki Oja [9] introduced a normalization term in order to prevent the weights to "explode", that is to become larger and larger.

The first model of a neuron, called MCP neural model after its inventors' initials, can be traced back to the work of Warren McCulloch and Walter Pitts [10], who in 1943

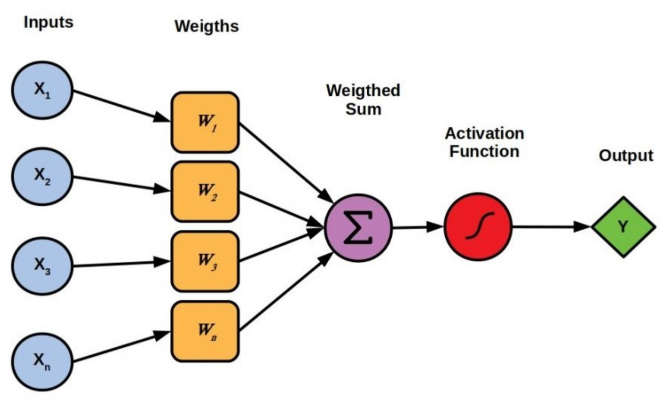


Figure 1.1: Basic modern perceptron scheme

suggested to model the working of a neuron as a linear step function such as:

$$y = \begin{cases} 1, & \sum_i w_i x_i \geq \theta \text{ and } z_j = 0, \forall j \\ 0, & \text{otherwise} \end{cases} \tag{1.2}$$

where y stands for output, x_i is the input, w_i is called weight, z_j are the inhibitory inputs and θ is the threshold. The weight w_i is a characteristic of the neuron and it is adjusted depending on the task we want the network to perform. This is the first step leading to the introduction of the *Perceptron* by Frank Rosenblatt in 1958 [11]. It was basically a linear function of input signals. The main difference with modern perceptrons is the absence of non-linear activation functions (e.g. sigmoid function σ, ReLU). A basic scheme of a common perceptron is shown in Figure 1.1.

A basic neural network is simply built by putting perceptrons together. By placing them side by side, we get a *single one-layer neural network* and, by putting them in a sequence, we get a *multi-layer perceptrons* (MLP)[12].

In 1974 Paul Werbos introduced the concept of *backpropagation* [13]. This is an algorithm that computes the gradient in order to update the parameters or weights of a neural network. Backpropagation first spreads the error from the output layer back to where the parameters need to be updated, by means of the chain rule. Then uses standard gradient descent to update the parameters by a step called the learning rate, but we will discuss these methods later on.

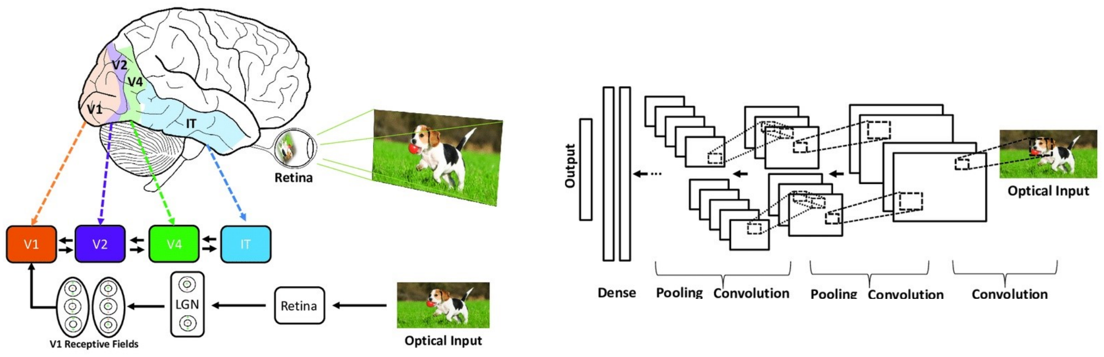


Figure 1.2: Convolutional neural networks originate from our understanding of the primary visual cortex

1.2 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) were first introduced in order to mimic our vision. “Neocognitron” [14] is generally recognised as the model that inspires CNNs on the computation side. It consists of two different kinds of layer: S-layer and C-layer. The former is inspired by the primary visual cortex and works as a feature extractor: it is in fact trained to be responsive to a particular feature of the image. The latter is mainly introduced for shift invariant property of features extracted by the S-layer. The first CNN, known as “LeNet”, was introduced in the 1990s by Yann LeCun [15], a postdoctoral computer science researcher. LeNet was able to recognize handwritten digits and was therefore used in banking and postal services. Other successful CNNs, such as

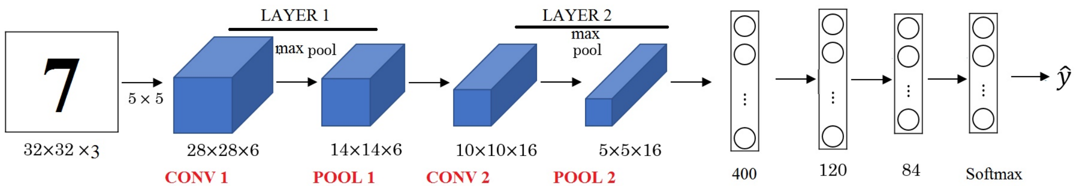


Figure 1.3: Architecture of a CNN to recognise digits such as LeNet

AlexNet [16] or VGG [17], got their popularity through the ImageNet Challenge [18], whose goal was to foster the development of CNNs by proposing a challenge regarding image classification task. From Figure 1.4 it is clear how neural networks with deeper architectures were a turning point in the history of computer vision algorithms, reducing the classification error from 26% to 16% in a few years.

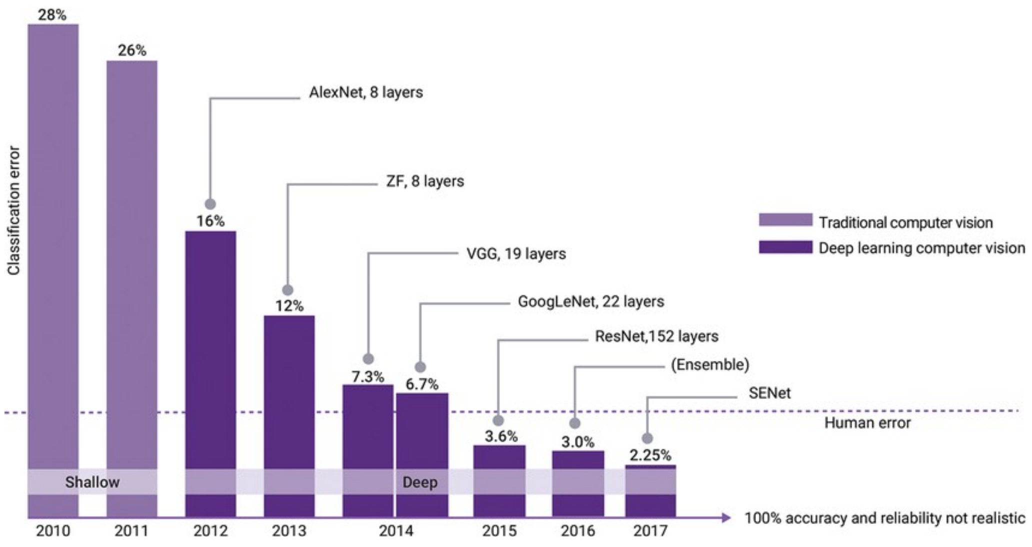


Figure 1.4: Winners of the ImageNet Challenge through years

1.3 Graph Neural Networks: the beginning

Deep learning models are very successful in processing traditional data like images or texts, but they are limited to grid-like data. However, people began to wonder how non-Euclidean data, such as graphs or manifolds, could be processed through some similar mechanism. This led to the creation of a distinct branch of Machine Learning called *geometric deep learning* (GDL), focused on more complex data. The goal is to modify the existing algorithms, in order to handle graphs and manifolds encoding the data. The term "geometric deep learning" was first introduced by Micheal Bronstein in his ERC grant in 2015 and then popularized through the paper [19]. Early forms of *Graph Neural Networks* (GNNs) can be traced back to the 90's, for example Alessandro Sperduti's work [20] of 1994, the "backpropagation through structure" of Goller and Kuchler [21] and the adaptive processing of data structures [22]. Nevertheless, the first actual Graph Neural Network was proposed in 2005 by M. Gori *et al.* to process graphs [23]. The first attempt, to generalize neural networks methods on graphs, dates back to 2008 by Scarselli *et al.* [24]. Later on, in the paper [25], Bruna *et al.* proposed the use of spectral CNNs on graphs. With the spreading of interest towards this new research field, many other algorithms were proposed, such as *Graph Attention Networks* (GATs) [26] or graph auto-encoders. This resulted in many applications of GDL to a wide range of practical problems, from biochemistry [27], skeleton-based human motion recognition tasks [28], to the recommender systems [29].

We conclude our history journey with a quote taken from the paper [30], written by M. Bronstein, J. Bruna, T. Cohen and P. Veličković:

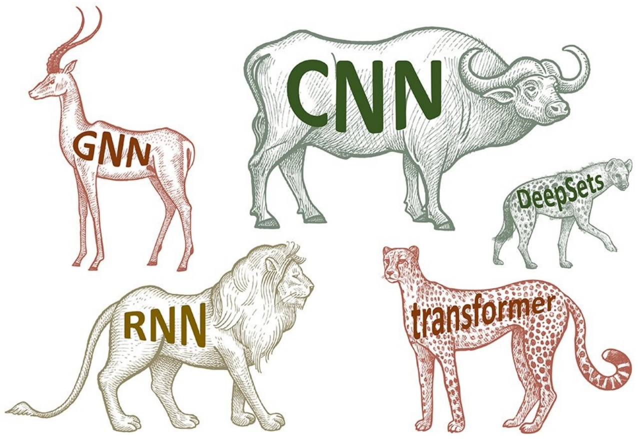


Figure 1.5: Deep learning today: a zoo of architectures, few unifying principles

"There is a veritable zoo of neural network architectures for various kinds of data, but few unifying principles..., this makes it difficult to understand the relations between various methods, inevitably resulting in the reinvention and re-branding of the same concepts in different application domains."

1.4 Datasets

We now introduce few basic concepts common to all neural networks for supervised learning.

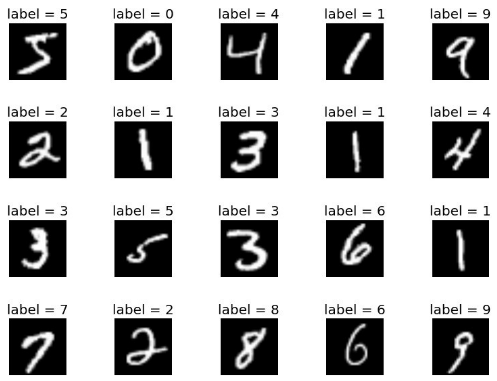


Figure 1.6: Examples of labeled images in the MNIST database

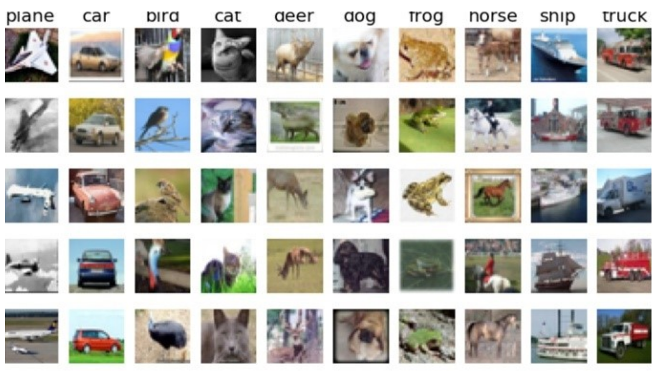


Figure 1.7: Some images in CIFAR-10 dataset divided by class

One of the main and most important ingredients is an homogeneous and clearly labelled dataset to train the neural network. Some benchmark datasets are the MNIST database, the CIFAR-10 dataset [31] for CNNs and the Zachary Club and Cora datasets for GNNs, that we will discuss in detail in Section 3.4. The MNIST dataset, shown in Figure 1.6, contains 70 000 black and white images, 28×28 pixels, representing digits from 0 to 9. Each image is labeled by the digit that represents and the task is to teach the neural network to recognise the hand-written digits. The CIFAR-10 dataset, instead, is a collection of 60 000 colored images 32×32, and the task is to predict what they are showing and classifying it between 10 different classes: cat, dog, bird, deer, frog, horse, ship, truck and plane. In Figure 1.7 are shown the classes in the dataset, as well as 5 random images from each.

A very common and useful way to organize the dataset for a supervised classification task is to split it in 3 parts:

- Training set;
- Validation set;
- Test set.

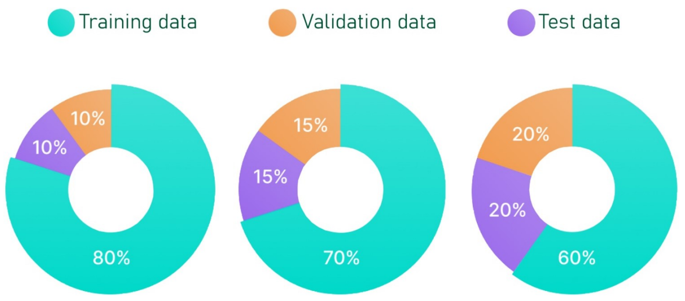


Figure 1.8: Splitting of the dataset between training, validation and test set

The *test set* is set aside from the very beginning and it is used to verify how well the neural network is performing. The evaluation criterion, called *accuracy*, measures the fraction of correct prediction on the test set over all predictions.

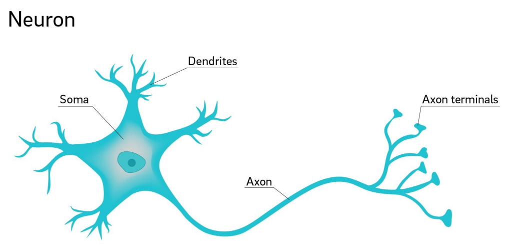
The remaining data is usually further divided into *training* and *validation sets*. The reason for this is because, generally, when building a neural network, there are many options to choose regarding the architecture. This options consist in choosing values for some variables called *hyperparameters*. For instance, an hyperparameter is the size of a single-layer or the type of loss function to use. We want to choose the options that will make our neural network work best, but we have to be careful not to use the test set for this purpose. Setting the hyperparameters to maximise the accuracy calculated on the test set will give us excellent results on the training set, but most likely the neural network will not performance so good, when it is used to process other data. For this reason it is fundamental to evaluate the accuracy of the neural network on the test set only at the very end. It is therefore necessary to pick a validation set aside from the test set, that is what will be used to tune the hyperparameters.

Finally, the training set is used to train the network. In Figure 1.8 are shown some standards splits of the dataset for CNNs.

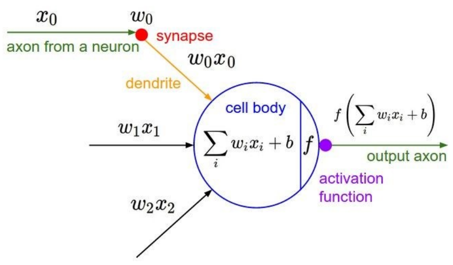
1.5 Perceptron

The first perceptron model was introduced in 1958 by Frank Rosenblatt [11] as a mathematical model of a biological neuron. In fact, as a neuron is the computational unit of the brain, similarly the perceptron functions as the neural network unit that can tell whether the input will activate the firing, and therefore give an output, or not.

Since a perceptron is modelled on the working of a neuron, let us focus briefly on how it operates. A biological neuron receives signals from other neurons, through his dendrites. When the incoming signal reaches a certain level of intensity, it triggers the firing of the neuron, that is a sharp electrical potential generated across the membrane called *spike*. This impulse travels through the axon and then ramifies into dendrites of other connected neurons as shown in Figure 1.9a. In this way, the neurons communicate via dendrites and axons and together create a network able to spread a signal made of more than 100 trillion connections in the average human brain.



(a) Biological model



(b) Mathematical model

Figure 1.9: Two models of a brain neuron

As mentioned before, the perceptron was introduced in order to mimic the working of the biological neuron. Being the mathematical model, it is composed of the main features as described before, as illustrated in Figure 1.9b. If a neuron receives an input signal x_i from another neuron, the signal will travel through the dendrite. In order to mimic the synaptic strength, we multiply the signal for the weight w_i associated with the dendrite. Since a single neuron receives input signals from thousand of dendrite at the same time, we simulate this behaviour by summing them, to obtain the total signal arriving at the neuron, that can be written as: $\sum_i w_i x_i + b$ (b is the bias term). To simulate the firing of the biological neuron, when the input reaches a certain threshold level, we use an *activation function* f.

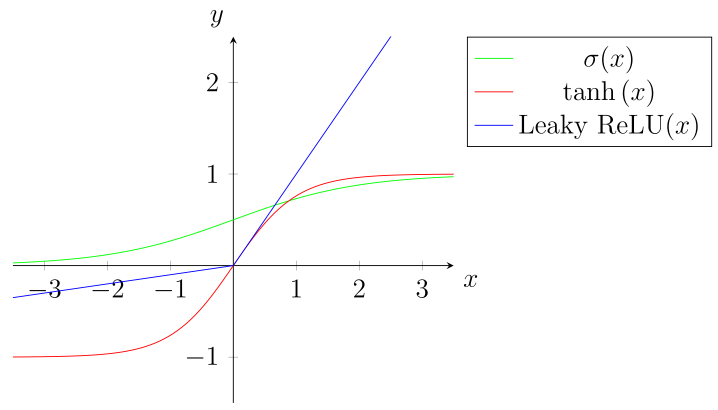


Figure 1.10: Activation functions

Here are some of the most famous activation functions, whose graphs are shown in Figure 1.10:

- **Sigmoid**. $\sigma(x) = 1/(1 + e^{-x})$.
- **Tanh** $\tanh(x) = 2\sigma(2x) - 1$
- **Leaky ReLU** $f(x) = \begin{cases} \alpha x & \text{if } x < 0 \\ x & \text{otherwise} \end{cases}$ where $\alpha \ll 1$.

The next step to build a neural network is to put perceptrons together as it happens for neurons in the human brain. By placing them side-by-side, we get a single layer. Two neurons belonging to the same layer are generally not connected. If we stack the layers in a sequence, we build a MLP (Multi-Layer Perceptrons), also called *Artificial Neural Networks* (ANNs). Typically these networks have fully-connected layers, which means that each neuron in a layer sends his output signals to all of the neurons in the following layer, as shown in Figure 1.11. Note how the neurons are arranged in lines. The neurons in between the input and the output layers form what are called *hidden layers*.

This is the basic idea which is behind almost all of the deep learning architectures. In this way, we are able to build many kinds of neural networks, by just changing the size or the number of hidden layers. In CNNs for image processing, for example, the neurons are not arranged in one row, but they acquire also a depth dimension, forming 3D layers.

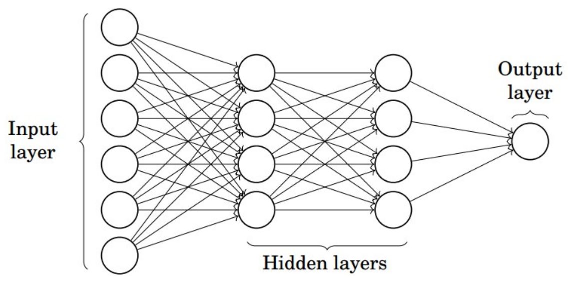


Figure 1.11: Example of a MLP

1.6 Training

The previous sections give us enough information to understand the fundamental bases of the training process. In order for a neural network to learn, there has to be an element of feedback involved. The idea is to compare the desired outcome with what the network produces and to use the difference between such outcomes to modify the parameters of the network in order to improve its accuracy.

To clarify the concepts, let us consider a simple task such as the classification of the CIFAR-10 images. An image consists of pixels and the color of each pixel is defined by three numbers representing the three color channels (Red, Green, Blue). Therefore, a CIFAR-10 image is determined by $32 \times 32 \times 3 = 3072$ numbers that can be arranged in a vector $\mathbf{x}_i$. This relatively easy problem becomes much more difficult when working with graph-like data, as we shall see later on.

Once we have stretched our image into a 3072-dimension vector $\mathbf{x}_i$, we can insert it in the input layer and perform the forward pass. Recall that a perceptron executes the sum of its input signals as $\sum_i w_i x_i + b$, where x_i is the signal coming from the i-th neuron of the previous layer, w_i the weight of the connection and b represents the bias. We can write the operation of passing the signal from a layer to the next one in a more compact notation, using vectors and matrices, as

$$\mathbf{y} = W\mathbf{x} + \mathbf{b} \tag{1.3}$$

where $\mathbf{x}$ is the input signal represented by a vector, $\mathbf{y}$ is the output signal, that is y_k stands for the signal received by the k-th neuron in the next layer. W and $\mathbf{b}$ contain the parameters and they represent, respectively, the weight matrix and the bias vector. In particular, W_{ij} stands for the connection from the j-th neuron of a layer to the i-th neuron of the next one. In Figure 1.12 is shown a simple example of forward pass process from a layer with 4 neurons to another layer with 3 neurons. In fact, in Figure 1.12 we start with a 4-dimensional vector $\mathbf{x}$ representing the signal and, after the matrix multiplication, we end up with a 3-dimensional vector as a result. Afterwards, the outcome of Equation (1.3) has to go through one of the activation functions discussed in Section 1.5, and only then we have computed a forward pass of the signal between

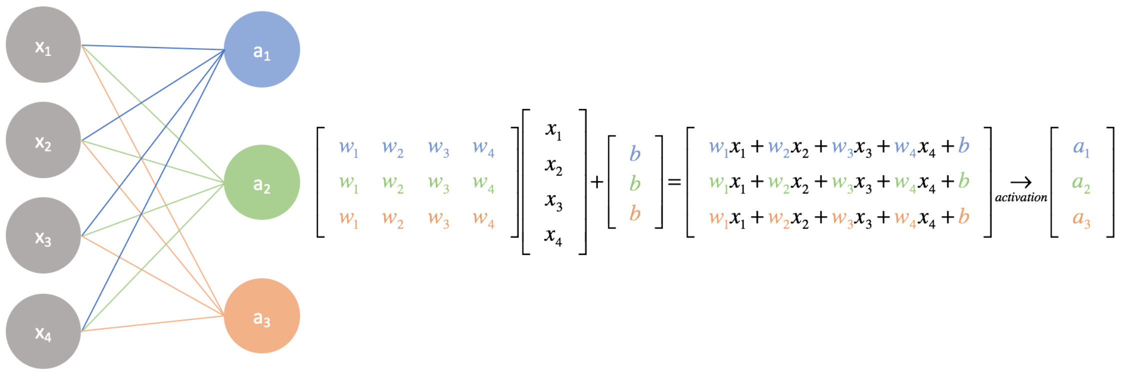


Figure 1.12: Matrix multiplication in forward pass process

two layers, that is basically one matrix multiplication followed by a bias offset and an activation function. We can therefore write the forward pass in a more complete way as

$$\mathbf{x}_{i+1} = f\left(W_i \mathbf{x}_i + \mathbf{b}_i\right)$$

where $\mathbf{x}_i$ stands for the signal on the i-th layer of the neural network, while W_i and $\mathbf{b}_i$ are the weight matrix and bias vector of the i-th layer. In simple deep learning architectures, this process has to be repeated for every pair of connected layers, until reaching the output layer.

Typically, the values of the last hidden layer are mapped into the class scores in the output layer through a *linear classifier*, that is simply a forward pass without the activation function (Equation (1.3)). Therefore, in the case of CIFAR-10 challenge, the output layer will have 10 neurons, each giving us the score of every class.

Once we have the network's "guess", we have to give it a feedback, that is the network need to know weather it did good or not and what can be improved in order to perform a better job the next time. The work of telling the network how much its guess matches the actual classification is carried out by the loss function, of which we'll discuss in the next section.

Finally there is the actual learning, this is the process which allows the network to improve its accuracy by changing the parameters. In our network model there are two kinds of parameters: the weights matrix W and the bias vector $\mathbf{b}$, both between every two connected layers. Therefore, our goal now is to find the set of values of W and $\mathbf{b}$ that minimizes the loss function. This process is called *optimization*. From now on we'll refer to the set of all the parameters of the neural network simply as W.

The most straightforward method to optimize the loss function is through the *gradient descent*. In order to catch the idea of how this method works, we have to think about the loss function as a function of the parameters $f\left(W\right)$, thus it will be a scalar function in a very high dimensional space. Therefore, our goal is to find a set of parameters that minimizes the loss function, that is looking for a local minimum of $f(W)$. The first step is

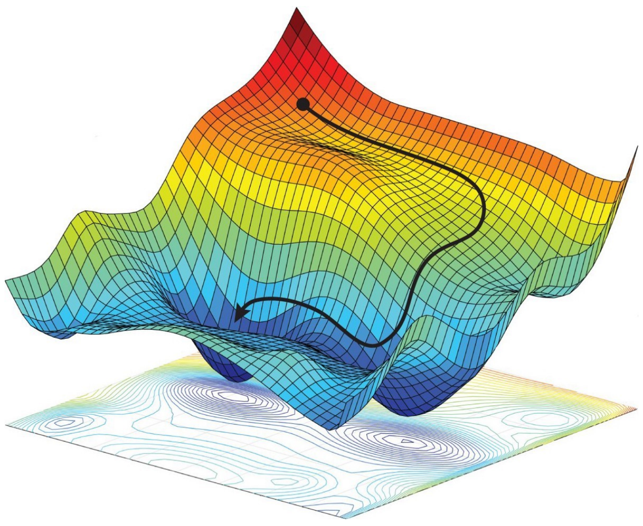


Figure 1.13: Representation of the gradient descent method

to perform the gradient of the loss function with the initial set of parameters $\nabla f\left(W^{(0)}\right)$. As we know, the gradient tells us towards which direction the loss function at point $W^{(0)}$ has the steepest ascent. By updating the parameters in the direction opposite to the gradient, we will move towards a local minimum of the loss function. Iterating this method in the following way

$$W^{(i+1)} = W^{(i)} - \nabla f\left(W^{(i)}\right)\eta \tag{1.4}$$

will return a sequence of parameters $W^{(i)}$ getting closer and closer to the local minimum, as shown in Figure 1.13. η denotes the learning rate, that is the size of the step you take on the loss function. Typically, a good choice for the learning rate is to set it around $\eta = 0.01$.

To be honest, Equation (1.4) does not really tell us how each parameter has to be updated. This job is carried out by *backpropagation* [32], which is based on the *chain rule*. Basically, through backpropagation we can find out how every single parameter needs to change in order to get the aimed gradient descent.

1.7 Loss Function

In this section we introduce and explain the most common loss functions, also known as *cost functions* or *objectives*. As we have seen in the previous section, the role of the loss function is to give the neural network a feedback in order for it to be able to learn and improve its accuracy. In particular, the loss function will return a high value if the actual outcome is far from the expected one and, vice versa, a low value if the outcome matches the correct class.

The first loss function we introduce is the *Multiclass Support Vector Machine loss* (SVM) or *Hinge loss*. It is based on the concept that we want the neuron in the output layer corresponding to the correct class to have a higher score than the others. In particular, we want its score to be higher than the wrong neurons by a certain real value called Δ. If we are evaluating a certain input $\mathbf{x}_i$, whose correct class is y_i, we can easily implement the SVM loss as

$$L_i = \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + \Delta\right) \tag{1.5}$$

where s_{y_i} and s_j are respectively the score of the correct class and the score of the other classes. In this way, the loss function will be zero only if the score of the correct class s_{y_i} is higher than all the other scores by at least Δ.

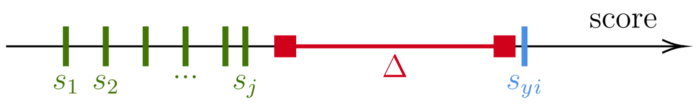


Figure 1.14: Representation of how SVM computes the loss

A little drawback of the SVM loss is that the set of parameters W is not unique. For instance, if the loss is already zero, any multiple of the set of parameters will also give zero loss. In order to avoid this behaviour, a *regularization penalty* $R\left(W\right)$ is added to the SVM loss. A common way to implement the SVM loss is to evaluate it on all the data in parallel and then compute the mean. Thus, the Multiclass SVM loss becomes

$$L = \frac{1}{N} \sum_{i}^{N} L_i + \lambda \sum_{k} \sum_{l} W_{k,l}^2 \tag{1.6}$$

where N is the number of training examples and λ is a hyperparameter.

Another popular choice for the loss function is the *cross-entropy loss*, which instead treats the class scores s_{y_i} and s_j as unnormalized log probabilities. Therefore, the *softmax function*, as shown below, returns the probability for the content of the image $\mathbf{x}_i$ to be assigned to the correct class y_i according to the current set of parameters W

$$P\left(y_i | \mathbf{x}_i ; W\right) = \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \tag{1.7}$$

The cross-entropy loss has the form

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \tag{1.8}$$

and can be interpreted as the negative log likelihood of the correct class. The full loss for the dataset is computed through Equation (1.6). Minimizing the cross-entropy function means performing the Maximum Likelihood Estimation on the parameters W, that is looking for the set of parameters that will maximize the probability on the correct class.

1.8 Neural Networks and Human Brain

We conclude this first chapter with a quick parenthesis on a popular misconception about neural networks. In fact, often in the media is stated that modern deep learning machines "think" or are able to act in a sentient manner. Even though it is true that these algorithms are impressive, because of their skills to learn and often succeed in beating the human brain in some restricted fields, it is necessary to understand the differences between thinking and the ability to complete a particular task.

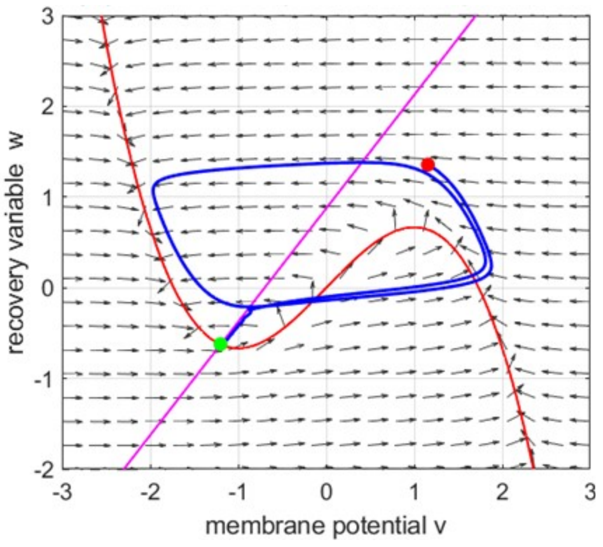


Figure 1.15: FitzHugh-Nagumo model behaviour

Despite the complexity and the extraordinary number of parameters that constitutes modern neural networks, they are just a very simplified model of the human brain. In fact, we have over 100 billion neurons in our brains connected with more than 100 trillion synapses. Besides the substantial quantitative difference, also a single biological neuron overcomes the mathematical model in terms of complexity. Even if simulating the firing of the neuron through a linear product followed by an activation function works well when building an artificial neural network, our real neurons behave in a much more complicated and sophisticated way. A popular and definitely more realistic model simulating the firing of a neuron is the FitzHugh-Nagumo model, that describes a prototype of an excitable system through two differential equations, whose trend is illustrated in Figure 1.15:

$$\begin{cases} \dot{v} = v - \frac{v^3}{3} - w + RI_{ext} \\ \tau\dot{w} = v + a - bw \end{cases} . \tag{1.9}$$

In addition, our actual neurons are connected to each other in all sort of arrangements, certainly they are not lined up and sorted to form organised layers as in ANNs.

Despite these facts, often when it comes to learn and perform well in a specific task, artificial neural networks can beat the human brain. But we have to be careful to suggest that deep learning machines have the ability to think. The reason is because they are not able to take what they learn and apply it to any other problem. Therefore, despite the fact that "thought" is something that we do not fully understand, I believe it is clear that learning machines do not actually think (yet).

Chapter 2

Graphs

In this chapter we define the concept of graph and some matrices associated with it. We study how such matrices reflects some algebraic and topological properties of the graph. We are going to focus in particular on the Laplacian matrix since it plays a key role in spectral graph theory. In the end, we will see some applications of these methods.

2.1 Graphs

Let us start from the definition of a graph. There are many real-life examples and applications of graphs, from social networks and protein interactions, to communication networks.

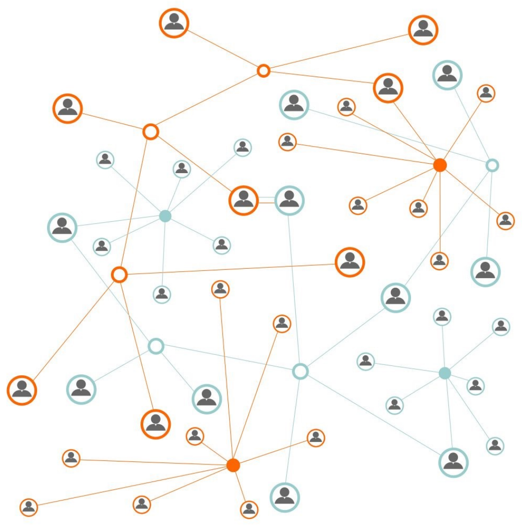
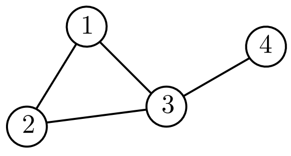


Figure 2.1: Social network

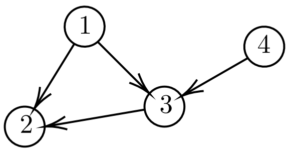
Definition 2.1. A *graph* $\mathcal{G}$ is a pair of sets $(\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \{1, \dots, n\}$ and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$. The elements of $\mathcal{V}$ are called nodes or vertices. The elements of $\mathcal{E}$ are called edges.

The nodes of a graph can be labelled as $v_1, \dots, v_n$. The edges can be also numbered and referred to with $e_1, \dots, e_k$, where k indicates the number of edges, or can be labelled in order to show the two nodes that they connect; for example, if an edge links node v_i and v_j, we label it with e_{ij}.

A graph can be *undirected* if the edges do not have a direction, therefore if v_i is connected to v_j, v_j is connected to v_i as well; otherwise the graph is called *directed.*



(a) Undirected graph



(b) Directed graph

Figure 2.2: Graph examples

A way to represent the connections and the topological structure of a graph is through the adjacency matrix.

Definition 2.2. The *adjacency matrix* A of a graph $\mathcal{G}$ is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix and can be defined as

$$A_{ij} = \begin{cases} 1 & e_{ij} \in \mathcal{E} \\ 0 & \text{otherwise} \end{cases}.$$

It is easy to understand that in a undirected graph $A_{ij} = 1$ if and only if $A_{ji} = 1$, since $e_{ij} \in \mathcal{E}$ if and only if $e_{ji} \in \mathcal{E}$, hence A is a symmetric matrix. Let us see the adjacency matrices for the simple graphs shown in Figure 2.2.

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 2.3: Adjacency matrix for the undirected graph in Figure 2.2a

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 2.4: Adjacency matrix for the directed graph in Figure 2.2b

In addition, a graph can also have weights associated with its edges, in this case the adjacency matrix is more commonly denoted with W and called weight matrix.

Definition 2.3. The *weight matrix* W of a graph $\mathcal{G}$ is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix, where W_{ij} is the weight of the edge e_{ij}. If $e_{ij} \notin \mathcal{E}$, then $W_{ij} = 0$.

We define the *node degree* as the number of edges connected with a particular vertex. In the graph of Figure 2.2a, for example, the degree of the first node is 2, and we write this as $\deg(v_1) = 2$. In a directed graph, instead, we have the *indegree* and the *outdegree*, being, respectively, the number of edges leading into that node and those leading away from it. If $\deg(v) = k$ for every node v, then we call the graph *regular*. We now introduce the concept of degree matrix.

Definition 2.4. The *degree matrix* D of an undirected graph $\mathcal{G}$ is a $|\mathcal{V}| \times |\mathcal{V}|$ diagonal matrix displaying the degree of each node on its diagonal, namely $D_{ii} = \deg(v_i)$.

We refer to the set of vertices adjacent with a node v_i as the *neighbourhood* $\mathcal{N}(v_i)$ of the vertex v_i.

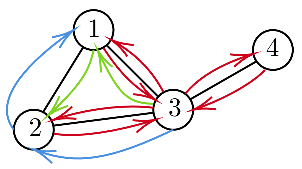
Let us now introduce and discuss briefly an interesting theorem.

Theorem 2.1. *Let $\mathcal{G}$ be a graph and A its adjacency matrix, then $(A^r)_{ij}$ represents the number of walks of length r from v_i to v_j.*

Before the proof we look at a simple example.

Example 2.1. We look at A^2 for a simple graph, like the undirected graph in Figure 2.2a, that for convenience it is reported below.

$$A^2 = \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 3 & 0 \\ 1 & 1 & 0 & 1 \end{pmatrix}$$



Let us first focus on the diagonal. It should appear clear that $(A^2)_{ii} = \deg(v_i)$. In fact, the number of walks of length 2 starting from v_i and finishing in v_i again is exactly the degree of the node, because the only possible walks are $v_i \to v_k \to v_i$ for all $v_k \in \mathcal{N}(v_i)$. We can express matrix multiplication as $C_{ij} = \sum_k A_{ik}B_{kj}$. In this way, we can also understand the meaning of the off-diagonal terms. In fact,

$$\left(A^2\right)_{ij} = \sum_k A_{ik}A_{kj}$$

and the term $A_{ik}A_{kj}$ will bring a contribution to the sum for a particular k only if both A_{ik} and A_{kj} are different from zero, which means that the edges e_{ik} and e_{kj} belong both to $\mathcal{E}$, therefore the walk $v_i \to v_k \to v_j$ is allowed.

We now prove the Theorem 2.1 by induction.

Proof. A^1_{ij} clearly represents the number of walks of length 1 from v_i to v_j. Suppose that $\left(A^{(r-1)}\right)_{ik}$ represents the number of walks of length $(r-1)$ from v_i to v_k, then

$$(A^r)_{ij} = \sum_k \left(A^{r-1}\right)_{ik} A_{kj}.$$

For a particular $\tilde{k}$, the only way for $(A^{r-1})_{i\tilde{k}} A_{\tilde{k}j}$ to be different from zero is to have at least 1 walk of length $(r-1)$ from v_i to $v_{\tilde{k}}$ and $e_{\tilde{k}j} \in \mathcal{E}$. If these conditions are satisfied, then it contributes to the sum over k with $(A^{r-1})_{i\tilde{k}}$ walks from v_i to $v_{\tilde{k}}$ of length $(r-1)$, followed by a final step from $v_{\tilde{k}}$ to v_j. Summing over k will sum all these contributions, resulting in the number of walks of length r from v_i to v_j. $\square$

2.2 Diffusion on graph

Let $\mathbb{R}^{\mathcal{V}}$ denote the real valued functions on the set of vertices $\mathcal{V}$:

$$\mathbb{R}^{\mathcal{V}} = \{f : \mathcal{V} \to \mathbb{R}\}.$$

If we give an ordering to $\mathcal{V}$, we can identify $\mathbb{R}^{\mathcal{V}}$ with $\mathbb{R}^n$, where n denotes the number of nodes. This identifies a function in $\mathbb{R}^{\mathcal{V}}$ with a vector in $\mathbb{R}^n$, namely

$$\mathbf{f} = (f_1, \dots, f_n) = (f(v_1), \dots, f(v_n)) \in \mathbb{R}^n.$$

This means we can view the adjacency matrix A as a linear mapping on $\mathbb{R}^{\mathcal{V}}$, rather than its isomorphic vector space $\mathbb{R}^n$. As a consequence, we see that

$$(Af)(u) = \sum_{v\in\mathcal{V}} A_{uv} f(v) = \sum_{v\in\mathcal{N}(u)} f(v), \quad f \in \mathbb{R}^{\mathcal{V}}.$$

Thus, the value of Af at node u is the sum of the values of f on the neighbours of u. If we now suppose that $x \in \mathbb{R}^{\mathcal{V}}$ is an eigenfunction of A, we can write

$$(Ax)(u) = \lambda x(u) \Longrightarrow \sum_{v\in\mathcal{N}(u)} x(v) = \lambda x(u) \quad \forall u \in \mathcal{V}.$$

In words, the sum of the values of the eigenfunction x on the neighbours of node u is λ times its value at point u itself.

An important matrix is the *diffusion* or *walk matrix*, defined as AD^{-1}. Being D the degree matrix, its inverse will be $D^{-1} = \operatorname{diag}(1/\deg(v_1), \dots, 1/\deg(v_n))$. Thus, if we apply AD^{-1} to a function f in $\mathbb{R}^{\mathcal{V}}$ at a node u, it will return

$$\left(\left(AD^{-1}\right)f\right)(u) = \sum_{v\in\mathcal{V}}\sum_{k\in\mathcal{V}} A_{uk} D^{-1}_{kv} f(v) = \sum_{v\in\mathcal{V}} \frac{A_{uv}}{\deg(v)} f(v) = \sum_{v\in\mathcal{N}(u)} \frac{f(v)}{\deg(v)}.$$

We can compare this with a similar but slightly different result that we will need later on, that is applying $D^{-1}A$ to the same function f at node u:

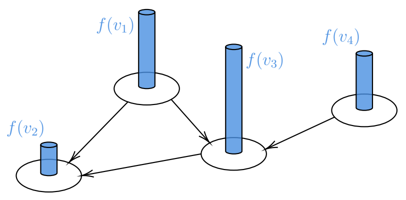
$$\left(\left(D^{-1}A\right) f\right)(u) = \sum_{k\in\mathcal{V}}\sum_{v\in\mathcal{V}} D^{-1}_{uk} A_{kv} f(v) = \sum_{v\in\mathcal{V}} \frac{A_{uv}}{\deg(u)} f(v) = \sum_{v\in\mathcal{N}(u)} \frac{f(v)}{\deg(u)}.$$

$\left((D^{-1}A)f\right)(u)$ gives an average of $f(v)$, with v node in the neighbourhood of u. Both of the results require a sum over the values of function f at the neighbours of the node u, the only difference is that in the former we divide each $f(v)$ for the degree of the node v, in the latter we normalize the sum with the degree of node u.

2.3 Incidence Matrix and Gradient

In this section we introduce the incidence matrix X and its interpretation as the discrete gradient on a graph. Assume f is a real function on the set of vertices $\mathcal{V}$ as seen before.

$$\begin{aligned} f\colon \mathcal{V} &\longrightarrow \mathbb{R} \\ v_i &\longmapsto f(v_i) = f_i. \end{aligned}$$



Recall that if we order the vertices in the graph, then we can represent the function as a vector $\mathbf{f} = (f(v_1), f(v_2), \ldots, f(v_n))$. Let us now introduce the differential df on a graph.

Definition 2.5. Let $\mathcal{G}$ be a graph and f a function in $\mathbb{R}^{\mathcal{V}}$, then we define the *differential* df of f as

$$\begin{aligned} df\colon \mathcal{E} &\longrightarrow \mathbb{R} \\ e_{ij} &\longmapsto f(v_j) - f(v_i). \end{aligned}$$

We shall also denote df with ∇f, and we call it the *gradient* of f. Notice that ∇ is an operator mapping functions on the vertices $\mathbb{R}^{\mathcal{V}}$ into functions on the edges $\mathbb{R}^{\mathcal{E}}$:

$$\nabla\colon \mathbb{R}^{\mathcal{V}} \longrightarrow \mathbb{R}^{\mathcal{E}},$$

Defining $\nabla f(e_{ij}) = f(v_j) - f(v_i)$ is the analogue of the gradient operator on continuous functions. In fact, if f is a real function on $\mathbb{R}^n$, namely

$$\begin{aligned} f\colon \mathbb{R}^n &\longrightarrow \mathbb{R} \\ \mathbf{x} &\longmapsto f(\mathbf{x}), \end{aligned}$$

then the gradient $\nabla f(\mathbf{x}) = \left(\frac{\partial f(\mathbf{x})}{\partial x_1}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_n}\right)$ tells us how much and in what direction the function f is changing at point $\mathbf{x}$. However, in a graph the topology is represented by the set of edges $\mathcal{E}$, therefore we define the analogue of ∇ only on the edges. Moreover, since the usual gradient tells us how much a function is changing, its discrete analogue returns the difference in the function values between the two vertices at the end of the edge: $\nabla f(e_{ij}) = f(v_j) - f(v_i)$.

Since $\mathbb{R}^{\mathcal{V}}$ can be identified with $\mathbb{R}^n$, we may think at the gradient ∇ as a matrix. For this purpose we introduce the incidence matrix X.

Definition 2.6. The *incidence matrix* X of an undirected graph $\mathcal{G}$ is a $|\mathcal{E}| \times |\mathcal{V}|$ matrix, defined as

$$X_{ij} = \begin{cases} 1 & \text{if } v_j \text{ is in the edge } e_i \\ 0 & \text{otherwise} \end{cases}.$$

Note that the columns of X are associated with vertices and its rows with edges. Therefore, considering a simple example of undirected graph, X becomes as shown in Figure 2.5.

$$X = \underset{\text{Edges}}{\Big\downarrow} \begin{array}{c} \\ e_1 \\ e_2 \\ e_3 \\ e_4 \end{array} \overset{\xrightarrow{\text{Vertices}}}{\begin{array}{cccc} v_1 & v_2 & v_3 & v_4 \\ \end{array}} \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

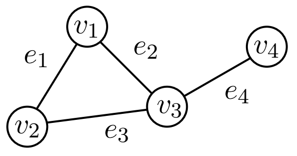


Figure 2.5: Incidence matrix of an undirected graph

The definition of X for a directed graph is slightly different, but more useful for our purpose, that is finding the discrete analogue of the gradient.

Definition 2.7. The *incidence matrix* X of a directed graph $\mathcal{G}$ is a $|\mathcal{E}| \times |\mathcal{V}|$ matrix, defined as

$$X_{ij} = \begin{cases} -1 & \text{if the edge } v_j \text{ is the tail of the edge } e_i \\ 1 & \text{if the edge } v_j \text{ is the head of the edge } e_i \\ 0 & \text{otherwise} \end{cases}.$$

In Figure 2.6 is shown the incidence matrix of a directed example graph.

We can therefore find in the incidence matrix of a directed graph the discrete analogue of the gradient operator ∇. In fact, X has to associate to each edge e_k the differential of a function f on that edge, that is $df(e_k)$.

$$X = \underset{\text{Edges}}{} \downarrow \begin{array}{c|cccc} & \multicolumn{4}{c}{\xrightarrow{\text{Vertices}}} \\ & v_1 & v_2 & v_3 & v_4 \\ e_1 & -1 & 1 & 0 & 0 \\ e_2 & -1 & 0 & 1 & 0 \\ e_3 & 0 & 1 & -1 & 0 \\ e_4 & 0 & 0 & 1 & -1 \end{array}$$

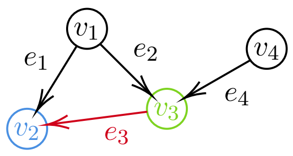


Figure 2.6: Incidence matrix of a directed graph

Example 2.2. Let us look at a simple example. If $\mathbf{f}$ is the vector containing the values of f on the vertices of the graph in Figure 2.6, then

$$\nabla\mathbf{f} = X\mathbf{f} = \begin{pmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} f(v_1) \\ f(v_2) \\ f(v_3) \\ f(v_4) \end{pmatrix} = \begin{pmatrix} -f(v_1) + f(v_2) \\ -f(v_1) + f(v_3) \\ f(v_2) - f(v_3) \\ f(v_3) - f(v_4) \end{pmatrix} = \begin{pmatrix} df(e_1) \\ df(e_2) \\ df(e_3) \\ df(e_4) \end{pmatrix}$$

and at the end we obtain the gradient $\nabla\mathbf{f}$ of function f.

2.4 Laplacian on Graphs

In this section we introduce the notion of graph Laplacian and we investigate the analogies with the usual Laplacian.

Definition 2.8. The *Laplacian matrix* L of an undirected graph is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix, defined as

$$L := A - D,$$

where A and D are the adjacency matrix and degree matrix, respectively.

It is also usual to find the Laplacian matrix defined as $L := D - A$, but we prefer the first definition for reasons that we will understand soon. Remember that we have defined L only for an undirected graph, since it will be discussed in detail later on in this section.

Example 2.3. Before we look at the analogies with the ordinary Laplacian, let us see how L looks like for a simple undirected graph, as the one shown in Figure 2.7.

$$L = A - D = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} - \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix} = \begin{pmatrix} -2 & 1 & 0 & 1 \\ 1 & -3 & 1 & 1 \\ 0 & 1 & -2 & 1 \\ 1 & 1 & 1 & -3 \end{pmatrix}$$

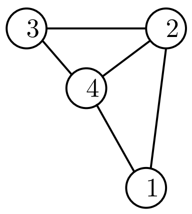


Figure 2.7: Undirected graph

Basically, L has the opposite of the degree of each node on the diagonal, and a '1' in L_{ij} if the edge $e_{ij} \in \mathcal{E}$.

2.5 Divergence on graphs

We now try to understand how L can be seen as the discrete analogue of the Laplacian operator on multivariate continuous functions. In ordinary geometry, the Laplacian is the operator defined as

$$\Delta = \nabla \cdot \nabla$$

that is, the Laplacian is the divergence of the gradient. We have already found in the incidence matrix the discrete equivalent of the gradient. Therefore, we now look for a way to represent the divergence on a graph.

Given a vector field $\mathbf{v}(\mathbf{x})$ in $\mathbb{R}^n$, then the divergence of $\mathbf{v}$ at point $\mathbf{x}$ is

$$\nabla \cdot \mathbf{v}(\mathbf{x}) = \frac{\partial v_1}{\partial x_1}(\mathbf{x}) + \ldots + \frac{\partial v_n}{\partial x_n}(\mathbf{x}).$$

Intuitively, the divergence tells us how much flow of $\mathbf{v}$ is coming out of point $\mathbf{x}$. Let us now try to find its correspondent on a graph. We know that the divergence is a scalar function that takes a vector field as input. Therefore, we assume to have a discrete version of a vector field, that is a function g assigning a real value to each edge, namely

$$\begin{aligned} g\colon \mathcal{E} &\longrightarrow \mathbb{R} \\ e_i &\longmapsto g(e_i). \end{aligned}$$

We say that the function g is in $\mathbb{R}^{\mathcal{E}}$, that is the set of real valued functions on the edges. If we give an ordering to the edges, then we can introduce the vector $\mathbf{g} = (g(e_1), \ldots, g(e_k))$ in $\mathbb{R}^k$, where k is the number of edges. Keeping in mind that the divergence $\nabla \cdot \mathbf{v}(\mathbf{x})$ quantifies the flow of $\mathbf{v}$ going out of $\mathbf{x}$, we give the following definition.

Definition 2.9. Let $\mathcal{G}$ be a directed graph and $g \in \mathbb{R}^{\mathcal{E}}$, we define the *divergence* of g at node v as

$$\nabla \cdot \mathbf{g}(v) = \sum_{u \in \mathcal{V}} g(e_{vu}) - \sum_{u \in \mathcal{V}} g(e_{uv}),$$

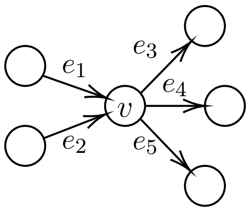


Figure 2.8: Directed graph to compute the divergence on v

where e_{uv} denotes an edge from node u to v.

Thus, the divergence $\nabla \cdot \mathbf{g}(v)$ returns the sum of the values of g on the edges going outwards from node v, minus the sum of the values of g on those going towards v. Notice that the divergence is an operator mapping functions on edges into functions on vertices, namely

$$\begin{aligned}\nabla\cdot : \mathbb{R}^{\mathcal{E}} &\longrightarrow \mathbb{R}^{\mathcal{V}}\\ \mathbf{g} &\longmapsto \nabla \cdot \mathbf{g}.\end{aligned}$$

Example 2.4. Assume we have the graph in Figure 2.8, then if we compute the divergence of g on node v according to the definition, we obtain

$$\nabla \cdot \mathbf{g}(v) = g(e_3) + g(e_4) + g(e_5) - g(e_1) - g(e_2),$$

since the edges going towards v will decrease the flow of g moving outwards from v.

In general, the opposite of the transpose of the incidence matrix is the discrete equivalent of the divergence.

$$\nabla \cdot \mathbf{g} = -X^\top \mathbf{g}$$

Example 2.5. Assume we have the graph in Figure 2.9. Therefore, we can write

$$-X^\top \mathbf{g} = \begin{pmatrix} -1 & 0 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 & -1 \\ 0 & 1 & -1 & -1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{pmatrix} \begin{pmatrix} g(e_1) \\ g(e_2) \\ g(e_3) \\ g(e_4) \\ g(e_5) \end{pmatrix} = \begin{pmatrix} -g(e_1) + g(e_3) \\ g(e_1) - g(e_2) - g(e_5) \\ g(e_2) - g(e_3) - g(e_4) \\ g(e_4) + g(e_5) \end{pmatrix}.$$

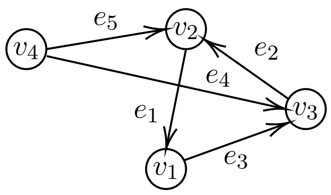


Figure 2.9: Directed graph

Before putting everything together we have to consider a small problem between the two formulations of Laplacian. In fact, we defined $L = A - D$ only for an undirected graph, whereas, in order to write the incidence matrix of a graph, it has to be directed. How can this problem be solved?

We have to introduce the concept of orientation of an undirected graph.

Definition 2.10. An *orientation* of an undirected graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a function σ from $\mathcal{E}$ to $\{-1, 1\}$, such that $\sigma(e_{ij}) = -\sigma(e_{ji})$.

Definition 2.11. An *oriented* graph is a graph with a particular orientation.

Basically, an orientation gives a direction to each edge. In particular, if $\sigma(e_{ij}) = 1$, then e_{ij} has the tail on v_i and the head on v_j. By orienting a graph, we will substantially transform an undirected graph in a directed one, but there is a slight difference between the two that we are going to address. We can say that an oriented graph is a directed graph. In fact, once we have given an orientation to the graph, we can write the incidence matrix equal to when we had a directed graph. The slight difference between the oriented and the directed graph is that, while the first will have at most one edge between two nodes, the second one can also have two edges, as shown in Figure 2.10. Therefore, although we are right to say that every oriented graph is also directed, the opposite is not always true.

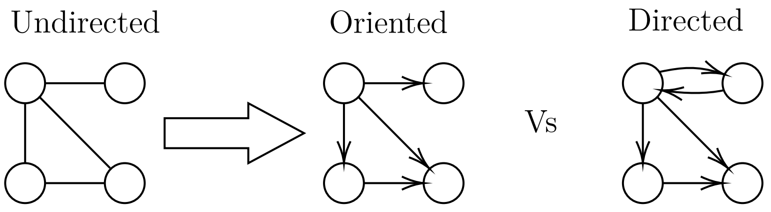


Figure 2.10: Difference between oriented and directed graph

2.6 Laplacian of an Oriented Graph

In ordinary geometry we have that the Laplacian operator is

$$\Delta = \partial_{x_1}^2 + \ldots + \partial_{x_n}^2 = \nabla \cdot \nabla, \tag{2.1}$$

where ∇ denotes the gradient operator. In our previous discussion, we saw that the incidence matrix X and the opposite of its transpose $-X^\top$ represent, respectively, the discrete gradient and divergence operator. Considering Equation (2.1), we want to introduce the following proposition.

Proposition 2.1. *Assume $\mathcal{G}$ to be a directed or an oriented graph and X its incidence matrix, then we can write the Laplacian matrix as*

$$L = -X^\top X.$$

However, we need to make some observations regarding the orientation of the graph. In fact, we have defined the Laplacian matrix (Definition 2.8) only for an undirected graph. Nevertheless, in order to write the incidence matrix X, the graph needs to be directed, or at least oriented. Thus, if we want to show the correspondence between the definition $L = A - D$ and the proposition $L = -X^\top X$, we need to take an undirected graph and give it an arbitrary orientation.

We first check the dimensions of the matrices. L is a $|\mathcal{V}| \times |\mathcal{V}|$ matrix while X is $|\mathcal{E}| \times |\mathcal{V}|$. Thus, $-X^\top X$ is a $(|\mathcal{V}| \times |\mathcal{E}|) \cdot (|\mathcal{E}| \times |\mathcal{V}|) = |\mathcal{V}| \times |\mathcal{V}|$ matrix, like L itself.

Now, we look as usual at a simple example to convince ourselves that Proposition 2.1 is true.

Example 2.6. Assume we have the graph in Figure 2.11. We can write the Laplacian according to the definition as

$$L = A - D = \begin{pmatrix} -1 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -1 \end{pmatrix}.$$

This is equal to writing L as

$$L = -X^\top X = \begin{pmatrix} 1 & 0 \\ -1 & -1 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix} = \begin{pmatrix} -1 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -1 \end{pmatrix}.$$

We now try to understand why computing $-X^\top X$ actually returns L. Starting from the diagonal, we have

$$L_{ii} = \sum_j \left(-X^\top\right)_{ij} X_{ji},$$

where L_{ii} denotes the i-th diagonal element of L. Since $\left(-X^\top\right)_{ij} = -X_{ji}$, it becomes

$$L_{ii} = \sum_j -X_{ji}X_{ji} = -\sum_j \left(X_{ji}\right)^2 = -\deg(v_i)$$

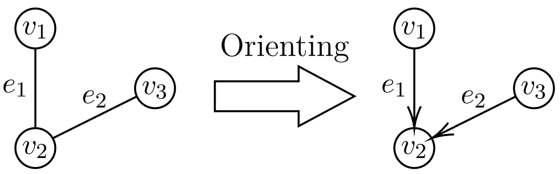


Figure 2.11: An undirected graph and the same graph with an orientation

because, whenever we have $X_{ji} = \pm 1$, it means that the node v_i is in the edge e_j. Therefore, when we sum on j we find exactly the number of edges adjacent with v_i, that is the degree of v_i. Notice also that the values on the diagonal of L do not depend on the orientation we gave the graph. In fact, the orientation decides only the sign of the entries of the incidence matrix. But since the diagonal elements of L depends on the square of the entries of X, then the orientation of the graph is irrelevant. We can try to visualize it with the following image:

$$\begin{pmatrix} 1 & 0 \\ -1 & -1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix} = \begin{pmatrix} -1 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -1 \end{pmatrix}$$

When we compute an off-diagonal entry of $-X^\top X$, for example L_{ij}, where $i \neq j$, this is the dot product between the i-th row of $-X^\top$ and the j-th column of X.

$$L_{ij} = \sum_k \left(-X^\top\right)_{ik} X_{kj} = \sum_k -X_{ki} X_{kj}.$$

Therefore, we can think of this also as the opposite of a dot product between the i-th and the j-th columns of X. Recalling that the incidence matrix was defined as

$$X_{ij} = \begin{cases} -1 & \text{if the edge } v_j \text{ is the tail of the edge } e_i \\ 1 & \text{if the edge } v_j \text{ is the head of the edge } e_i \\ 0 & \text{otherwise} \end{cases},$$

then, when we compute the dot product between the i-th and the j-th columns of X, we are checking if there is an edge e_k between v_i and v_j. If that is the case, then one between X_{kj} and X_{ki} will be 1 and the other -1, because one between v_i and v_j will be the tail and the other the head of e_k, regardless of the orientation we give to e_k. Notice also that, since we can have at most one edge between two nodes, then L_{ij} can only assume the values 1 or 0, depending whether or not there is an edge between node v_i and v_j. We can again visualize the matrix multiplication with the following figure:

$$\begin{pmatrix} 1 & 0 \\ -1 & -1 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix} = \begin{pmatrix} -1 & 1 & 0 \\ 1 & -2 & 1 \\ 0 & 1 & -1 \end{pmatrix} \qquad X = \begin{pmatrix} -1 & 1 & 0 \\ 0 & 1 & -1 \end{pmatrix}$$

Notice also that the orientation we give the graph in order to write the Laplacian matrix as $-X^\top X$ can be arbitrary, since the final Laplacian L will not depend on it.

We now prove the following theorem.

Theorem 2.2. *The Laplacian matrix L of an oriented graph is a symmetric and negative semi-definite matrix.*

Proof. L is clearly a symmetric matrix, since it is defined as $L = A - D$, therefore as a sum of symmetric matrices. This can also be proved from the formula $L = -X^\top X$, because:

$$L_{ij} = \sum_k \left(-X^\top\right)_{ik} X_{kj} = \sum_k -X_{ki}X_{kj} = \sum_k -X_{kj}X_{ki} = \sum_k \left(-X^\top\right)_{jk} X_{ki} = L_{ji}.$$

Let us now prove that L is negative semi-definite. Consider the quadratic form associated with L, it is:

$$\langle \mathbf{v}, L\mathbf{v}\rangle = \langle \mathbf{v}, -X^\top X\mathbf{v}\rangle = -\langle \mathbf{v}, X^\top X\mathbf{v}\rangle = -\langle X\mathbf{v}, X\mathbf{v}\rangle = -|X\mathbf{v}|^2 \leq 0 \quad \forall \mathbf{v} \in \mathbb{R}^n,$$

where $\langle\ ,\ \rangle$ represents the euclidean product. $\square$

In the previous proof we found that all of the eigenvalues of L will be either zero or negative. If we sort them and denote with λ_i the i-th eigenvalue of L, then we can write

$$0 \geq \lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n.$$

Theorem 2.3. *Let $\mathcal{G}$ be an oriented graph with n nodes, then* $\mathrm{rk}L \leq n-1$.

In order to prove Theorem 2.3, we first introduce and prove the following Lemma.

Lemma 2.1. *Let $\mathcal{G}$ be an oriented graph with n nodes, then the rank of the incidence matrix is* $\mathrm{rk}X \leq n-1$.

Proof. Suppose that $\mathbf{z} = (z_1, \ldots, z_n) \in \mathbb{R}^n$ is a vector such that $X\mathbf{z} = \mathbf{0}$. If we consider the edge e_k which goes from vertex v_j to vertex v_i, then the incidence matrix will have only zeros in the k-th row, except for $X_{kj} = -1$ and $X_{ki} = 1$.

$$(X\mathbf{z})_k = \sum_l X_{kl}z_l = X_{ki}z_i + X_{kj}z_j = z_i - z_j,$$

therefore, since $X\mathbf{z} = \mathbf{0}$, we have

$$z_i - z_j = 0 \Longrightarrow z_i = z_j.$$

We can thus conclude that the null space of X is not void, since a constant vector has to belong to it. If the null space is not void, then $\mathrm{rk}X \leq n-1$. $\square$

Notice that the previous proof is intuitively obvious. In fact, since X represents the discrete gradient, then $\mathbf{z}$ can be thought as usual as a real function on the vertices and $X\mathbf{z}$ as its gradient. If $\mathbf{z}$ is a constant function, then its gradient will be naturally null for every edge.

We can now prove Theorem 2.3 and show that $\mathrm{rk}L \leq n-1$, that is equivalent to say that at least one eigenvalue will be zero.

Proof. We would like to prove that $\text{rk}L = \text{rk}X$. Assume, like before, $\mathbf{z} \in \mathbb{R}^n$, with the condition that $L\mathbf{z} = \mathbf{0}$. We have already seen that $\langle \mathbf{z}, L\mathbf{z}\rangle = -|X\mathbf{z}|^2$. If $L\mathbf{z} = \mathbf{0}$, then $X\mathbf{z} = \mathbf{0}$, that is, if $\mathbf{z}$ belongs to the null space of L, then it is in the null space of X too. Vice versa, if $X\mathbf{z} = \mathbf{0}$, then $L\mathbf{z} = -X^\top X\mathbf{z} = -X^\top \mathbf{0} = \mathbf{0}$, which implies $\text{rk}L = \text{rk}X \leq n - 1$. □

Since L is real and symmetric then, thanks to the spectral theorem, we can find an orthonormal basis of $\mathbb{R}^n$ consisting in eigenvectors of L. Therefore, a vector $\mathbf{u} \in \mathbb{R}^n$ can be written as

$$\mathbf{u} = \langle \mathbf{u}, \mathbf{v}_1\rangle \mathbf{v}_1 + \ldots + \langle \mathbf{u}, \mathbf{v}_n\rangle \mathbf{v}_n = \sum_i \langle \mathbf{u}, \mathbf{v}_i\rangle \mathbf{v}_i,$$

where $\mathbf{v}_1, \ldots, \mathbf{v}_n$ are the eigenvectors associated with the eigenvalues $\lambda_1, \ldots, \lambda_n$. Of course, we can always look at these eigenvectors v_i as eigenfunctions $\phi_i \in \mathbb{R}^{\mathcal{V}}$, associating a real value to each vertex.

2.7 Dirichlet Energy

In this section we introduce the *Dirichlet energy* E_{Dir}, which will help us understand the meaning of the eigenfunctions of a graph. On continuous functions f on a compact domain $\Omega \in \mathbb{R}^n$, we can define the Dirichlet energy as:

$$E_{\text{Dir}}(f) := \int_\Omega \nabla f \cdot \nabla f dx = \int_\Omega ||\nabla f||^2 dx \geq 0.$$

This energy basically returns how "spiky" the function f is over the domain Ω. Its discrete analogue can be defined as follows.

Definition 2.12. Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a graph and L be its Laplacian matrix. We can define the *Dirichlet energy* of a function $\phi \in \mathbb{R}^{\mathcal{V}}$ as

$$E_{\text{Dir}}(\phi) = -q(\phi),$$

where q is the quadratic form associated with the matrix Laplacian L.

If we now look for an orthonormal basis on Ω of smoothest possible continuous functions (that is with the lowest Dirichlet energy), we have to solve the optimization problem

$$\begin{aligned} &\min_{f_1} E_{\text{Dir}}(f_1) \quad \text{with} \quad ||f_1|| = 1 \\ &\min_{f_i} E_{\text{Dir}}(f_i) \quad \text{with} \quad ||f_i|| = 1, \quad f_i \perp \text{span}\{f_1, \ldots, f_{i-1}\}. \end{aligned}$$

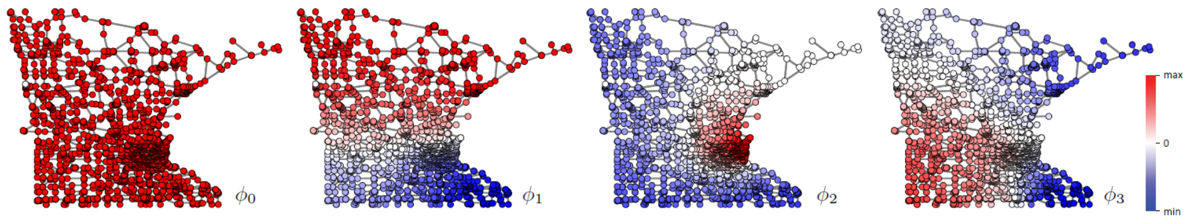


Figure 2.12: Laplacian eigenfunctions on the graph representing the roads in Minnesota (from paper [19])

In a discrete domain, the problem is solved by the eigenfunctions. In fact, for a generic function $\phi \in \mathbb{R}^{\mathcal{V}}$, we have

$$q(\phi) = \langle \phi, L\phi \rangle = \langle \sum_i \langle \phi, \phi_i \rangle \phi_i, L \sum_j \langle \phi, \phi_j \rangle \phi_j \rangle = \sum_i \sum_j \langle \phi, \phi_i \rangle \langle \phi, \phi_j \rangle \langle \phi_i, \lambda_j \phi_j \rangle,$$

where ϕ_i denotes the i-th eigenfunction of L and we have expanded in series the generic function ϕ. Since $\langle \phi_i, \phi_j \rangle = \delta_{ij}$, then

$$q(\phi) = \sum_i \sum_j \langle \phi, \phi_i \rangle \langle \phi, \phi_j \rangle \lambda_j \langle \phi_i, \phi_j \rangle = \sum_i \langle \phi, \phi_i \rangle^2 \lambda_i = \sum_i -|\lambda_i| \langle \phi, \phi_i \rangle^2.$$

Therefore, the functions with the lowest Dirichlet energy which form an orthonormal basis in $\mathbb{R}^{\mathcal{V}}$ are the eigenfunctions of L:

$$\begin{aligned} \phi_1 &\longrightarrow E_{\text{Dir}}(\phi_1) = -q(\phi_1) = |\lambda_1| = 0 \\ &\vdots \\ \phi_n &\longrightarrow E_{\text{Dir}}(\phi_n) = -q(\phi_n) = |\lambda_n|. \end{aligned}$$

$\phi_1, \ldots, \phi_n$ are then the smoothest functions we can find forming an orthonormal basis in $\mathbb{R}^{\mathcal{V}}$. In particular, we have already shown that the function $\phi_1 = \text{const}$ is the eigenfunction associated with the eigenvalue $\lambda_1 = 0$ and is the smoothest one we can find, with the lowest Dirichlet energy, $E_{\text{Dir}}(\phi_1) = 0$.

In Figure 2.12 is shown the graph representing the roads in Minnesota (USA). On the same graph are illustrated the trends of four different eigenfunctions of the Laplacian matrix. The one on the left represents the eigenfunction ϕ_0 with the lowest possible Dirichlet energy and, as we have seen, it is exactly the constant function. The other figures show other eigenfunctions of L with higher Dirichlet energy values, therefore are more "spiky" than ϕ_0.

2.8 Heat Diffusion Equation

We can employ what we have learnt in the previous sections to write the heat equation on graphs. Recall that in general the heat equation is written as

$$\begin{cases} \partial_t T(x,t) = c\Delta T(x,t) \\ T(x,0) = T_0(x) \end{cases} \tag{2.2}$$

where $T(x,t)$ is the temperature at point x at time t and Δ is the laplacian operator. The second equation represents the initial condition. c is referred to as the *thermal diffusivity constant.*

Before writing the heat equation on graphs, let us talk about others definition of the discrete Laplacian. In Definition 2.8, we defined it as

$$L \coloneqq A - D.$$

In the paper [19], M. Bronstein defines L as

$$L = M^{-1}(A - D),$$

where M is a given matrix. Clearly, if we choose $M = I$, we get the same Laplacian matrix we defined in Definition 2.8. If, however, we put $M = D$, where D is the degree matrix, we obtain what is known as *diffusive Laplacian.*

Definition 2.13. Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a graph, A its adjacency matrix and D its degree matrix. The *diffusive Laplacian* of the graph is

$$L = D^{-1}(A - D) = D^{-1}A - I.$$

We are interested in the heat diffusion equation on a graph. If we assume the thermal diffusivity constant c to be equal to 1 for the sake of simplicity, we can write the discrete version of Equation (2.2) as

$$\partial_t h(v,t) = Lh(v,t), \tag{2.3}$$

where h represents the temperature on the graph (or in general a signal) and is a function in $\mathbb{R}^{\mathcal{V}}$, depending on the time t. We can solve this equation writing h as a linear combination of the eigenfunctions ϕ_i of the Laplacian L:

$$h(t) = \sum_{i=1}^{n} a_i(t)\phi_i,$$

where $a_i(t) = \langle h(t), \phi_i \rangle$. Substituting this into Equation (2.3) gives us:

$$\sum_{i=1}^{n} (\partial_t a_i(t))\phi_i(v) = \sum_{i=1}^{n} a_i(t) L\phi_i(v) = \sum_{i=1}^{n} a_i(t)\lambda_i\phi_i(v)$$

$$\sum_{i=1}^{n} \left(\partial_t a_i(t) - \lambda_i a_i(t)\right)\phi_i(v) = 0.$$

Since the eigenfunctions ϕ_i are orthogonal to each other, we are left with n differential equations for the coefficients $a_i(t)$:

$$\partial_t a_i(t) - \lambda_i a_i(t) = 0.$$

These can be easily solved with exponential solutions:

$$a_i(t) = a_i(0)\exp(\lambda_i t) = a_i(0)\exp(-|\lambda_i|t) \Longrightarrow h(t) = \sum_{i=1}^{n} a_i(0)\exp(-|\lambda_i|t)\phi_i.$$

Notice that as t goes to infinity, the solution tends to the uniform eigenfunction ϕ_1, how we would expect:

$$\lim_{t\to\infty} h(t) = \lim_{t\to\infty} \sum_{i=1}^{n} a_i(0)\exp(-|\lambda_i|t)\phi_i = a_1(0)\phi_1.$$

We now look briefly at the heat diffusion equation with the diffusive Laplacian, since it will be useful later on. If we still assume to have the thermal diffusivity constant c equal to 1, then Equation (2.3) can be written as

$$\partial_t h(v,t) = (D^{-1}A - I)h(v,t).$$

If we also assume the time to be discrete, then the time derivative of $h(t)$ is simply $h_{t+1} - h_t$:

$$h_{t+1}(v) - h_t(v) = D^{-1}Ah_t(v) - h_t(v) \Longrightarrow h_{t+1}(v) = D^{-1}Ah_t(v).$$

We already saw that

$$D^{-1}Ah_t(v) = \sum_{u\in\mathcal{N}(v)} \frac{h_t(u)}{\deg(v)}.$$

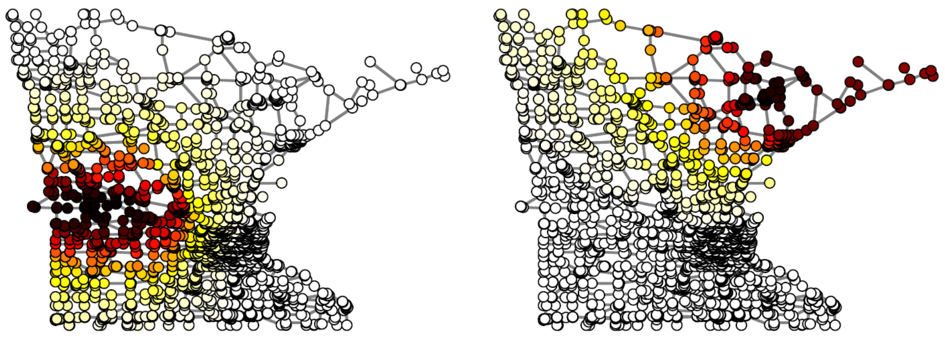


Figure 2.13: Example of heat diffusion on a graph (from paper [19])

Thus, the value of function h at time $t+1$ at node v depends on the sum of the values of the same function h at time t on the neighbours of node v, normalized with the degree $\deg(v)$. This should remind us of a diffusive process on the graph. If we iterate this over and over we will have a linear recursion

$$h_{t+1}(v) = \sum_{u \in \mathcal{N}(v)} \frac{h_t(u)}{\deg(v)}$$

where its fixed point is, again, the first eigenfunction ϕ_1.

In Figure 2.13 are illustrated two examples of heat diffusion on a graph representing the roads in Minnesota. Observe how moving the heat source to a different location changes the shape of the diffusion, which is an indication of the lack of shift-invariance on graphs.

Chapter 3

Geometric Deep Learning

In this chapter we discuss some methods and algorithms used to analyse geometric data based on the notions developed in the previous chapters. When working with graphs, we can classify the main tasks roughly as follows:

- **node classification**: we are given a graph and the features of only a small fraction of the nodes, and we want to predict the features of the remaining nodes;
- **edge classification**: similarly, we are given a graph and the features associated with some of the edges and we want to predict the features of the remaining edges;
- **graph classification**: we are given a number of different graphs, each labelled, and we want to learn to predict the labels of other graphs.

There are other classification tasks, however we shall not treat them in this thesis. For more details see [19], [33].

We shall concentrate in this chapter on the the first of these tasks: the node classification. We will see some of the methods employed to classify the nodes of a graph, like graphSAGE neural networks and GATs. Then we will apply one of these methods on the Zachary Karate Club ([33]) dataset to elucidate practically the results.

3.1 Encoder and Decoder

We start by discussing the *encoder-decoder framework*, which is the base for building *Geometric Convolutional Networks* (GCNs), the first kind of network we consider for node classification task.

We recall that in deep learning we classify images or more general data coming in a grid, that we call Euclidean data. A deep learning network or CNN training works according to the following steps:

- a series of convolutions and linear layers to compute a score associated to a datum;
- a loss function to compute the error in the prediction of the correct class of the given datum, obtained via forward pass;
- an update of the parameters through backpropagation via stochastic gradient descent pass.

In the case of non-euclidean data like graphs, the general structure of the learning process is the same. The forward pass process, however, occurs differently and has to be adapted to the data. In fact, there are some issues that have to be considered when dealing with data distributed according to a graph, like the geometric structure of the graph itself that needs to be given to the neural network. To do so, we use the encoder-decoder framework.

We now assume to have a node classification task, this means that we are given a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and a vector of m features for each node v, that we shall denote with $\mathbf{x}_v$ in $\mathbb{R}^m$. We can therefore represent the data as a matrix

$$X = (\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n) \in \mathbb{R}^{m \times n}$$

where n is the number of nodes in the graph $\mathcal{G}$.

Our goal is to make the network learn a *node embedding* for every node of the graph, that is a function which associates a p-dimensional vector to each node v. If we denote such embedding as f, we can write it as

$$\begin{aligned} f : \mathcal{V} &\longrightarrow \mathbb{R}^p \\ v &\longrightarrow \mathbf{z}_v. \end{aligned}$$

Generally, the dimension of $\mathbb{R}^p$ is chosen to be much smaller than the space of the features $\mathbb{R}^n$. This function f which provides the embedding of every node is called a *node encoder*. Even though we have defined the encoder for node classification task, it is possible to generalise this concept also for edge and graph classification tasks, but we do not treat those here.

The node embedding $f(\mathbf{x}_v)$ provided by the encoder is then given to a *decoder*, typically a simple linear classifier. The output values are similar to the scores for each class for a SVM loss or to their probabilities for the cross-entropy loss we discussed in Section 1.7. The output takes values in $\mathbb{R}^c$, where c is the number of classes.

The process of encoding and decoding described above is called *forward pass*. It is very similar to the forward pass of an ANN. In fact, the main difference is how the encoder is implemented, since it has to take into account also the geometric information, which is fundamental when working with non-Euclidean data.

Once the decoder has provided an output, it is used to compute the loss function as we have seen for deep learning, and then the parameters characterizing the encoder and the decoder are updated through backpropagation.

3.2 GraphSAGE

We now discuss a simple method to implement the encoder of an encoder-decoder framework. We do not focus on the implementation of the decoder since it is much more straightforward and similar to the architectures seen in Chapter 1. The encoder, instead, has to find a way to deal with the complex topological structure of graphs. The following algorithms are taken from the paper [34].

Assume we have a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$, and to each node v is associated a feature $\mathbf{h}_v$. We want to find a general algorithm that gives us the node embeddings. In order to find these embeddings, we have to consider the topology of our graph. An iteration of a GraphSAGE convolution can be described schematically as:

1. ${}^{k}\mathbf{h}_{\mathcal{N}(v)} = \text{AGGREGATE}(\{{}^{k-1}\mathbf{h}_u, \forall u \in \mathcal{N}(v)\})$;
2. ${}^{k}\mathbf{h}'_v = \text{CONCAT}({}^{k}\mathbf{h}_{\mathcal{N}(v)}, {}^{k-1}\mathbf{h}_v)$;
3. ${}^{k}\mathbf{h}''_v = \sigma(W^k \cdot {}^{k}\mathbf{h}'_v)$;
4. ${}^{k}\mathbf{h}_v = {}^{k}\mathbf{h}''_v / ||{}^{k}\mathbf{h}''_v||_2$.

where we use the notation:

- k stands for the k-th iteration of the convolution;
- AGGREGATE is a generic aggregator function;
- CONCAT is a function which concatenates ${}^{k}\mathbf{h}_{\mathcal{N}(v)}$ and ${}^{k-1}\mathbf{h}_v$;
- W^k is the weight matrix associated with the k-th layer of the convolution;
- σ is the activation function.

To complete the convolution, we have to repeat the same process for every node $v \in \mathcal{V}$ and iterate this process for every convolutional layer of our neural network. Notice how the change of dimensionality happens in passage 3 when multiplying ${}^{k}\mathbf{h}'_v$ with W^k, in fact the size of the matrix W^k decides the new dimension of the vector ${}^{k}\mathbf{h}_v$. The final representations output after K iterations is the node embedding $\mathbf{z}_v = {}^{K}\mathbf{h}_v, \forall v \in \mathcal{V}$.

Until now we have left the definition of the aggregator function free of conditions, but notice how ideally it should be invariant to permutations of its inputs. In fact, unlike the machine learning over images, the vertices of a graph have no natural ordering, thus the aggregator function must operate over an unordered set of vertices.

Furthermore notice how, since there is no way to use a filter scrolling through the data as in convolutional neural networks, the features of a node are passed to its neighbours. This process is often referred to as *message passing*.

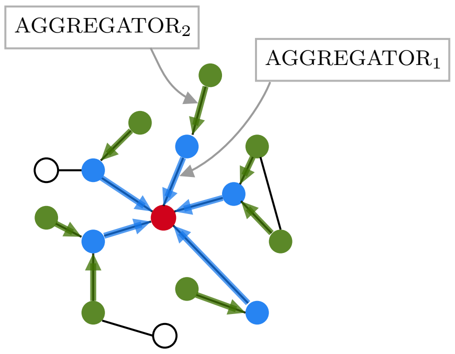


Figure 3.1: Aggregating feature information from neighbours with a two-step aggregator

One among the possible way to implement this message passing is

$$ {}^{k+1}\mathbf{h}_v = \sigma\left(W^k \sum_{u\in\mathcal{N}(v)} \frac{{}^{k}\mathbf{h}_u}{\deg(v)} + B^k \cdot {}^{k}\mathbf{h}_v \right). \tag{3.1} $$

The aggregator function is basically a mean of the vectors $\{{}^{k}\mathbf{h}_u, \forall u \in \mathcal{N}(v)\}$. W^K and B^k are the weight matrices that have to be updated through backpropagation. This aggregator is called *convolutional* since it is a rough approximation of a localized spectral convolution.

If we focus only on the aggregator function of Equation (3.1), it is of the form

$$ {}^{k+1}\mathbf{h}_v = \sum_{u\in\mathcal{N}(v)} \frac{{}^{k}\mathbf{h}_u}{\deg(v)} = (D^{-1}A)^{k}\mathbf{h}_v, $$

where D is the degree matrix and A the adjacency matrix.

$$ {}^{k+1}\mathbf{h}_v - {}^{k}\mathbf{h}_v = (D^{-1}A)^{k}\mathbf{h}_v - {}^{k}\mathbf{h}_v = (D^{-1}A - I)^{k}\mathbf{h}_v = D^{-1}(A-D)^{k}\mathbf{h}_v. $$

This is exactly the discrete heat diffusion equation we found in Section 2.8. In fact, since the time is discrete, ${}^{k+1}\mathbf{h}_v - {}^{k}\mathbf{h}_v$ represents the time derivative of ${}^{k}\mathbf{h}_v$, while $D^{-1}(A-D)$ is the definition of diffusive Laplacian L:

$$ \partial_t {}^{k}\mathbf{h}_v = L^{k}\mathbf{h}_v. $$

Thus the message passing defined in Equation (3.1) spreads the information along the graph following the discretization of the equation describing how heat diffuses across an object.

3.3 Graph Attention Networks

Graph Attention Networks (GATs) are networks designed to "pay attention" to some links with respect to others. The key idea is still to gather information during a convolutional iteration from the neighbouring nodes and put such information in a new vector. This is what in Section 3.2 we called AGGREGATE and CONCAT functions. Basically, we still want to find a way to aggregate the features, but this time we introduce the attention coefficients α_{vu} telling us the importance of the edge linking node v to node u.

Let us now see how we can find the attention coefficients and use them to build a GAT for node classification. We start by describing a single *graph attentional layer.* Let W be a weight matrix and $\mathbf{h}_v, \mathbf{h}_u \in \mathbb{R}^d$ the features of nodes v and u. We define

$$\begin{aligned} a : \mathbb{R}^{d'} \times \mathbb{R}^{d'} &\longrightarrow \mathbb{R} \\ W\mathbf{h}_v \times W\mathbf{h}_u &\longrightarrow e_{vu} = a(W\mathbf{h}_v, W\mathbf{h}_u). \end{aligned}$$

that indicates the importance associates to the link connecting v to u. Clearly, the more similar the features $W\mathbf{h}_v$ and $W\mathbf{h}_u$ are, the higher the importance associated is.

We then normalise e_{vu} using the softmax function:

$$\alpha_{vu} = \text{SOFTMAX}_u(e_{vu}) = \frac{\exp(e_{vu})}{\sum_{w \in \mathcal{N}(v) \cup \{v\}} \exp(e_{vw})}.$$

Once the attention coefficients are computed, we can use them to gather the features from the neighbourhood of node v into a new vector of features $\mathbf{h}'_v$ through a linear combination, followed by an activation function σ:

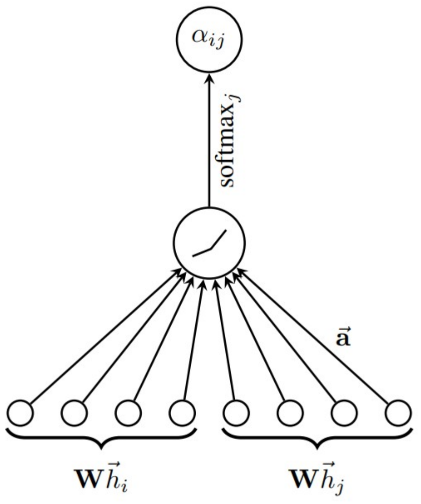
$$\mathbf{h}'_v = \sigma \left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \alpha_{vu} W \mathbf{h}_u \right).$$

In the paper [26], Veličković *et al.* choose an attention mechanism $a(W\mathbf{h}_v, W\mathbf{h}_u)$ as a single layer feedforward neural network characterised by a weight vector $\mathbf{a} \in \mathbb{R}^{2d'}$, applied to the concatenation of vectors $W\mathbf{h}_v$ and $W\mathbf{h}_u$, followed by a LeakyReLU non-linearity. We can thus express the attention coefficients as

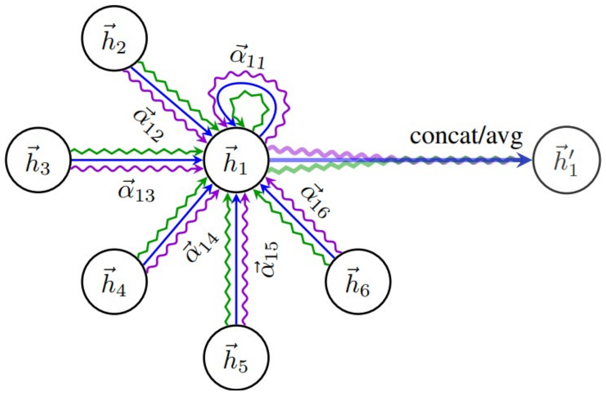
$$\alpha_{vu} = \frac{\exp\left(\sigma\left(\mathbf{a}^\top [W\mathbf{h}_v || W\mathbf{h}_u]\right)\right)}{\sum_{w \in \mathcal{N}(v) \cup \{v\}} \exp\left(\sigma\left(\mathbf{a}^\top [W\mathbf{h}_v || W_w]\right)\right)}$$

where $||$ is the concatenation operation and σ is the LeakyReLU activation function. A graphic representation of this mechanism is shown in Figure 3.2a.

Sometimes, to stabilize the learning process of self-attention, a *multi-head attention* approach is employed. K independent parallel convolutions are applied to the same node



(a) Attention mechanism used by the authors



(b) Illustration of multi-head attention with $K = 3$

Figure 3.2: Figures taken from paper [26]

v (with different weight matrices W^k) and then their features are concatenated to get a final vector $\mathbf{h}'_v \in \mathbb{R}^{Kd'}$, that is

$$\mathbf{h}'_v = ||_{k=1}^{K} \sigma \left(\sum_{u \in \mathcal{N}(v) \cup \{v\}} \alpha_{vu}^k W^k \mathbf{h}_v \right) .$$

where α_{vu}^k are the normalized attention coefficients computed by te k-th attention mechanism. In Figure 3.2b is shown an example of multi-head attention; different arrow styles denote independent attention computations that are then concatenated together.

3.4 Zachary Karate Club

In this section we introduce the Zachary Karate Club dataset and then we use a graph-SAGE convolutional neural network to classify the nodes.

The Zachary Karate Club is a social network of a karate club studied by W. Zachary in the article [35]. The dataset consists in 34 members of the karate club, documenting 154 links between pairs of members who interacted outside the club. After some time, from 1970 and 1972, the club became divided over an internal issue regarding the price of karate lessons. Every node is thus labeled by one of four classes, obtained via modularity-based clustering ([36]). See Figure 3.3 for an illustration.

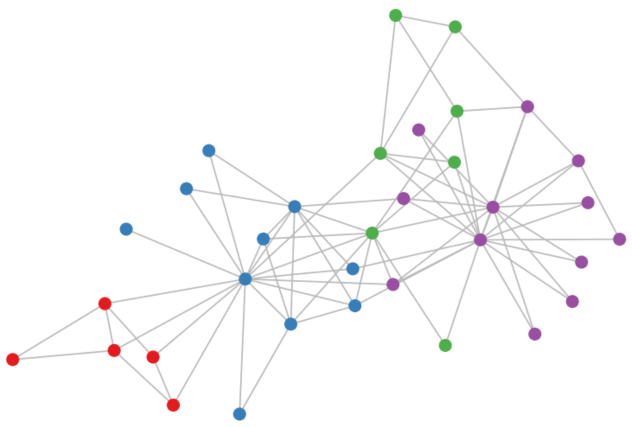


Figure 3.3: Representation of the Zachary Karate Club dataset taken from paper [33]

We now use this dataset to build a graph neural network for the task of node classification. We are given the labels for one node in each of the four groups. Our goal is to classify the remaining 30 nodes between the 4 groups, considering their relations, i.e. the links between pairs of nodes.

We recall that a graphSAGE layer aggregator is implemented as

$$^{k+1}\mathbf{h}_v = \sigma\left(W^k \sum_{u\in\mathcal{N}(v)} \frac{^{k}\mathbf{h}_u}{\deg(v)} + B^k \cdot {}^{k}\mathbf{h}_v\right).$$

We take a featureless approach by setting the input matrix as

$$X = \left(\mathbf{h}_1^0, \mathbf{h}_2^0, \ldots, \mathbf{h}_{34}^0\right) = I_{34}.$$

Note that the ordering of the nodes is random, therefore the ordering does not have meaning. Our neural network takes the features in $\mathbb{R}^{34}$ and, by mean of a graphSAGE encoding convolution (SAGEconv), transforms it in a 4-dimensional vector. The non-linearity σ is chosen to be the tanh() function. In the second layer we keep the dimension of the first, while in the final passage we shrink the data into a 2-dimensional vector $\mathbf{z}_v$ for every node v. $\mathbf{z}_v$ is the node embedding of v. Thus, for each node, this algorithm gathers the information from 3-hops distance nodes and encodes them into the embeddings $\mathbf{z}_v$. Finally, we use a linear classifier to carry the embeddings in the output layer. Since the possible labels are 4, the output vectors will be of size 4. In Figure 3.4 is illustrated the encoder mechanism of our neural network.

In Appendix A is shown all the code written to build the neural network and to train it. We use Adam optimizer ([37]) with a learning rate of 0.01 on a cross-entropy loss and we train for 500 training iterations.

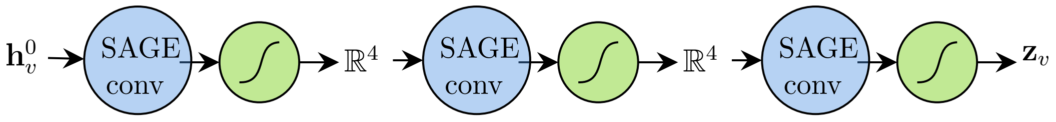


Figure 3.4: Illustration of the encoder mechanism in our neural network

In this section, however, we report only the outcome of the training. It is interesting to see how node embeddings change during the learning process. This is illustrated in Figure 3.6, together with the epoch, the value of the loss function, the training accuracy and the validation accuracy of each iteration. Notice how the same colours tend to cluster together and to get away from the different ones.

In Figure 3.5 is illustrated how the validation accuracy changes during the training process. The validation accuracy was chosen instead of the training accuracy because, since we have only four training nodes, the latter can only assume five values: 0%, 25%, 50%, 75% and 100%; therefore, it will quickly reach the maximum value and its graph would not tell us any information. From the graph in Figure 3.5, however, we observe that, at the end of the training process, our neural network is able to correctly predict up to roughly 65% of the nodes.

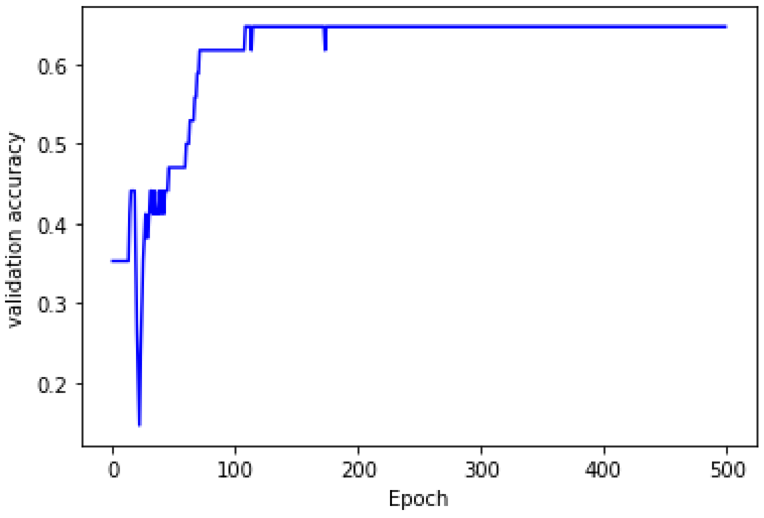


Figure 3.5: Graph showing the trend of validation accuracy

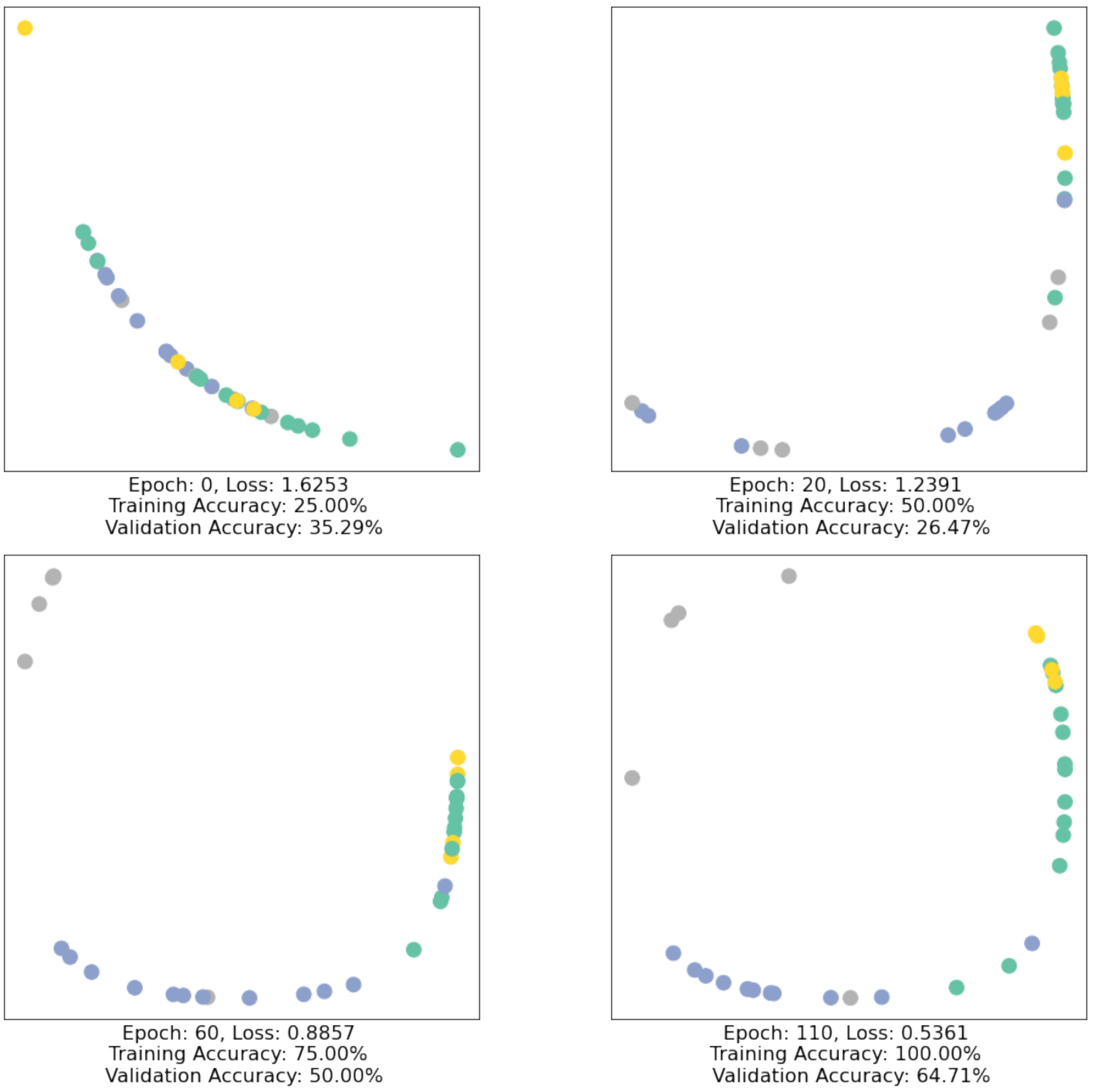


Figure 3.6: Four figures representing node embeddings at four different epochs

Conclusion

Thanks to the easy availability of computing resources and datasets, as well as the rapid development in deep learning technologies on domain such as texts, images and videos, geometric deep learning methods have achieved great success in theoretical research and applications. In this thesis, we have seen the process of diffusion on graphs and some graph neural networks such as graph attention networks. We have also implemented a graphSAGE convolutional network for the task of node classification on the Zachary Karate Club dataset able to correctly classify 64.71% of the nodes. However, W. Hamilton *et al.* in the paper [34] manage to reach an accuracy of 90.8% with a graphSAGE convolution network on the Reddit dataset. Furthermore, since geometric data is ubiquitous in the real life, geometric deep learning has the potential to find exciting new applications fields and to become the leading technology in machine learning in the near future.

Appendix A

Code

First of all, we have to import all the packages that we are going to need.

```
import torch
print("PyTorch has version {}".format(torch.__version__))

# Install torch geometric
!pip install -q torch-scatter -f
   https://pytorch-geometric.com/whl/torch-1.11.0+cu113.html
!pip install -q torch-sparse -f
   https://pytorch-geometric.com/whl/torch-1.11.0+cu113.html
!pip install -q torch-geometric

>>>PyTorch has version 1.11.0+cu113
```

A.1 Upload dataset

We then define a function useful for the sake of visualization.

```
# Helper function for visualization.
%matplotlib inline
import torch
import networkx as nx
import matplotlib.pyplot as plt

# Visualization function for NX graph or PyTorch tensor
def visualize(h, color, epoch=None, loss=None, accuracy=None):
   plt.figure(figsize=(7,7))
   plt.xticks([])
   plt.yticks([])
```

```python

    if torch.is_tensor(h):
        h = h.detach().cpu().numpy()
        plt.scatter(h[:, 0], h[:, 1], s=140, c=color, cmap="Set2")
        if epoch is not None and loss is not None and accuracy['train'] is not None and accuracy['val'] is not None:
            plt.xlabel((f'Epoch: {epoch}, Loss: {loss.item():.4f} \n'
                    f'Training Accuracy: {accuracy["train"]*100:.2f}% \n'
                    f' Validation Accuracy: {accuracy["val"]*100:.2f}%'),
                    fontsize=16)
    else:
        nx.draw_networkx(G, pos=nx.spring_layout(G, seed=42), with_labels=False,
                     node_color=color, cmap="Set2")
    plt.show()
```

We now access the dataset via the `torch_geometric.datasets` subpackage:

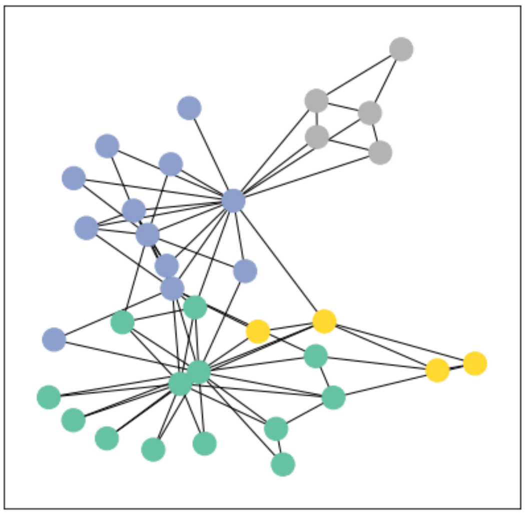
```python
from torch_geometric.datasets import KarateClub

dataset = KarateClub()
print(f'Dataset: {dataset}:')
print('======================')
print(f'Number of graphs: {len(dataset)}')
print(f'Number of features: {dataset.num_features}')
print(f'Number of classes: {dataset.num_classes}')

>>>Dataset: KarateClub():
   ======================
   Number of graphs: 1
   Number of features: 34
   Number of classes: 4
```

After initializing the `KarateClub()` dataset, we inspect some of its properties:

```python
data = dataset[0] # Get the first graph object.

print(data)
print('==============================================================')

# Gather some statistics about the graph.
print(f'Number of nodes: {data.num_nodes}')
print(f'Number of edges: {data.num_edges}')
print(f'Average node degree: {data.num_edges / data.num_nodes:.2f}')
print(f'Number of training nodes: {data.train_mask.sum()}')
print(f'Training node label rate: {int(data.train_mask.sum()) /
    data.num_nodes:.2f}')
print(f'Contains isolated nodes: {data.contains_isolated_nodes()}')
print(f'Contains self-loops: {data.contains_self_loops()}')
print(f'Is undirected: {data.is_undirected()}')

>>>Data(x=[34, 34], edge_index=[2, 156], y=[34], train_mask=[34])
   ==============================================================
   Number of nodes: 34
   Number of edges: 156
   Average node degree: 4.59
   Number of training nodes: 4
   Training node label rate: 0.12
   Contains isolated nodes: False
   Contains self-loops: False
   Is undirected: True
```

We can visualize the graph by converting it to the `networkx` library format:

```
from torch_geometric.utils import to_networkx

G = to_networkx(data, to_undirected=True)
visualize(G, color=data.y)
```



A.2 Code's architecture

```python
import torch_geometric
torch_geometric.__version__

import torch_scatter
import torch.nn as nn
import torch.nn.functional as F

import torch_geometric.nn as pyg_nn
import torch_geometric.utils as pyg_utils

from torch import Tensor
from typing import Union, Tuple, Optional
from torch_geometric.typing import (OptPairTensor, Adj, Size, NoneType,
                                    OptTensor)

from torch.nn import Parameter, Linear
from torch_sparse import SparseTensor, set_diag
from torch_geometric.nn.conv import MessagePassing
from torch_geometric.utils import remove_self_loops, add_self_loops, softmax

import time

import networkx as nx
import numpy as np
import torch
import torch.optim as optim

from torch_geometric.datasets import TUDataset
from torch_geometric.datasets import Planetoid
from torch_geometric.data import DataLoader

import torch_geometric.nn as pyg_nn

import matplotlib.pyplot as plt

from torch.nn import Linear
from torch_geometric.nn import SAGEConv


class GraphSageConv(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim,
                 ):
        super(GraphSageConv, self).__init__()
        self.conv1 = SAGEConv(input_dim, hidden_dim, normalize=True)
        self.conv2 = SAGEConv(hidden_dim, hidden_dim, normalize=True)
        self.conv3 = SAGEConv(hidden_dim, 2, normalize=True)
        self.classifier = Linear(2, output_dim)

    # Define the forward pass
    def forward(self, x, edge_index):
        h = self.conv1(x, edge_index)
        h = h.tanh()
        h = self.conv2(h, edge_index)
        h = h.tanh()
        h = self.conv3(h, edge_index)
        h = h.tanh() # Final GNN embedding space: h is the matrix of the node embeddings computed by our encoder which is the composition of 3 GraphSage layers!

        if self.training:
                x1 = h
                h = x1

        # Apply a final linear classifier.
        out = self.classifier(h)

        return out, h
```

A.3 Training

```python
model = GraphSageConv(dataset.num_features, 4, dataset.num_classes)
criterion = torch.nn.CrossEntropyLoss() # Define loss criterion.
optimizer = torch.optim.Adam(model.parameters(), lr=0.01) # Define optimizer.

def train(data):
    optimizer.zero_grad() # Clear gradients.
    out, h = model(data.x, data.edge_index) # Perform a single forward pass.
    loss = criterion(out[data.train_mask], data.y[data.train_mask]) # Compute the loss solely based on the training nodes.
    loss.backward() # Derive gradients.
    optimizer.step() # Update parameters based on gradients.


    accuracy = {}
    # Calculate training accuracy on our four examples
    predicted_classes = torch.argmax(out[data.train_mask], axis=1)
    target_classes = data.y[data.train_mask]
    accuracy['train'] = torch.mean(
        torch.where(predicted_classes == target_classes, 1, 0).float())

    # Calculate validation accuracy on the whole graph
    predicted_classes = torch.argmax(out, axis=1)
    target_classes = data.y
    accuracy['val'] = torch.mean(
        torch.where(predicted_classes == target_classes, 1, 0).float())

    return loss, h, accuracy

accuracies = []
epochs = []
for epoch in range(500):
    loss, h, accuracy = train(data)
    accuracies.append(accuracy["val"])
    epochs.append(epoch)
    # Visualize the node embeddings every 10 epochs
    if epoch % 10 == 0:
        visualize(h, color=data.y, epoch=epoch, loss=loss, accuracy=accuracy)
        time.sleep(0.3)

plot=plt.figure(1)
plt.plot(epochs, accuracies, color="blue")
plt.ylabel('validation accuracy')
plt.xlabel('Epoch')
plt.show()
```

Bibliography

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[2] Yann LeCun and Yoshua Bengio. “Convolutional Networks for Images, Speech, and Time Series”. In: *The Handbook of Brain Theory and Neural Networks*. Cambridge, MA, USA: MIT Press, 1998, pp. 255–258. ISBN: 0262511029.

[3] Ross Girshick et al. “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation”. In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 580–587. DOI: 10.1109/CVPR.2014.81.

[4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems*. Ed. by F. Pereira et al. Vol. 25. Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[5] David Lazer et al. “Life in the network: The coming age of computational social science: Science”. In: 2009.

[6] Wenming Cao et al. “A Comprehensive Survey on Geometric Deep Learning”. In: *IEEE Access* 8 (2020). DOI: 10.1109/ACCESS.2020.2975067.

[7] Haohan Wang and Bhiksha Raj. *On the Origin of Deep Learning*. 2017. DOI: 10.48550/ARXIV.1702.07800. URL: https://arxiv.org/abs/1702.07800.

[8] D.O. Hebb. “The Organization of Behavior: A Neuropsychological Theory”. In: *Journal of the American Medical Association* 143.12 (1950), pp. 1123–1123. DOI: 10.1001/jama.1950.02910470083028. eprint: https://jamanetwork.com/journals/jama/articlepdf/293955/jama_143_12_028.pdf. URL: https://doi.org/10.1001/jama.1950.02910470083028.

[9] Erkki Oja. “Simplified neuron model as a principal component analyzer”. In: *Journal of Mathematical Biology* 15 (1982). DOI: 10.1007/BF00275687. URL: https://doi.org/10.1007/BF00275687.

[10] Warren S. McCulloch and Walter Pitts. “A logical calculus of the ideas immanent in nervous activity”. In: *Bulletin of Mathematical Biology* 52.1 (1990), pp. 99–115. ISSN: 0092-8240. DOI: `https://doi.org/10.1016/S0092-8240(05)80006-0`. URL: `https://www.sciencedirect.com/science/article/pii/S0092824005800060`.

[11] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65 6 (1958), pp. 386–408.

[12] Kiyoshi Kawaguchi. “A multithreaded software model for backpropagation neural network applications”. In: 2000.

[13] Paul Werbos. “Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Science. Thesis (Ph. D.). Appl. Math. Harvard University”. PhD thesis. Jan. 1974.

[14] Kunihiko Fukushima and Sei Miyake. “Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition”. In: *Competition and Cooperation in Neural Nets*. Ed. by Shun-ichi Amari and Michael A. Arbib. Berlin, Heidelberg: Springer Berlin Heidelberg, 1982, pp. 267–285. ISBN: 978-3-642-46466-9.

[15] Y. Le Cun et al. “Handwritten Digit Recognition with a Back-Propagation Network”. In: *Proceedings of the 2nd International Conference on Neural Information Processing Systems*. NIPS’89. Cambridge, MA, USA: MIT Press, 1989, pp. 396–404.

[16] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: `10.1145/3065386`. URL: `https://doi.org/10.1145/3065386`.

[17] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2014. DOI: `10.48550/ARXIV.1409.1556`. URL: `https://arxiv.org/abs/1409.1556`.

[18] Olga Russakovsky et al. “ImageNet Large Scale Visual Recognition Challenge”. In: *International Journal of Computer Vision (IJCV)* 115.3 (2015), pp. 211–252. DOI: `10.1007/s11263-015-0816-y`.

[19] Michael M. Bronstein et al. “Geometric Deep Learning: Going beyond Euclidean data”. In: *IEEE Signal Processing Magazine* 34.4 (2017), pp. 18–42. DOI: `10.1109/MSP.2017.2693418`.

[20] Alessandro Sperduti. “Encoding Labeled Graphs by Labeling RAAM”. In: *Advances in Neural Information Processing Systems*. Ed. by J. Cowan, G. Tesauro, and J. Alspector. Vol. 6. Morgan-Kaufmann, 1993. URL: `https://proceedings.neurips.cc/paper/1993/file/fc49306d97602c8ed1be1dfbf0835ead-Paper.pdf`.

[21] C. Goller and A. Kuchler. “Learning task-dependent distributed representations by backpropagation through structure”. In: *Proceedings of International Conference on Neural Networks (ICNN’96)*. Vol. 1. 1996, 347–352 vol.1. DOI: 10.1109/ICNN.1996.548916.

[22] P. Frasconi, M. Gori, and A. Sperduti. “A general framework for adaptive processing of data structures”. In: *IEEE Transactions on Neural Networks* 9.5 (1998), pp. 768–786. DOI: 10.1109/72.712151.

[23] M. Gori, G. Monfardini, and F. Scarselli. “A new model for learning in graph domains”. In: *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.* Vol. 2. 2005, 729–734 vol. 2. DOI: 10.1109/IJCNN.2005.1555942.

[24] Franco Scarselli et al. “The Graph Neural Network Model”. In: *IEEE Transactions on Neural Networks* 20.1 (2009), pp. 61–80. DOI: 10.1109/TNN.2008.2005605.

[25] Joan Bruna et al. *Spectral Networks and Locally Connected Networks on Graphs.* 2013. DOI: 10.48550/ARXIV.1312.6203. URL: https://arxiv.org/abs/1312.6203.

[26] Petar Veličković et al. *Graph Attention Networks.* 2017. DOI: 10.48550/ARXIV.1710.10903. URL: https://arxiv.org/abs/1710.10903.

[27] David Duvenaud et al. *Convolutional Networks on Graphs for Learning Molecular Fingerprints.* 2015. DOI: 10.48550/ARXIV.1509.09292. URL: https://arxiv.org/abs/1509.09292.

[28] Zhiwu Huang et al. “Deep Learning on Lie Groups for Skeleton-Based Action Recognition”. In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2017, pp. 1243–1252. DOI: 10.1109/CVPR.2017.137.

[29] Federico Monti, Michael M. Bronstein, and Xavier Bresson. *Geometric Matrix Completion with Recurrent Multi-Graph Neural Networks.* 2017. DOI: 10.48550/ARXIV.1704.06803. URL: https://arxiv.org/abs/1704.06803.

[30] Michael M. Bronstein et al. *Geometric Deep Learning: Grids, Groups, Graphs, Geodesics, and Gauges.* 2021. DOI: 10.48550/ARXIV.2104.13478. URL: https://arxiv.org/abs/2104.13478.

[31] Alex Krizhevsky. “Learning Multiple Layers of Features from Tiny Images”. In: 2009.

[32] Shashi Sathyanarayana. “A Gentle Introduction to Backpropagation”. In: *Numeric Insight, Inc Whitepaper* (July 2014).

[33] Thomas N. Kipf and Max Welling. *Semi-Supervised Classification with Graph Convolutional Networks.* 2016. DOI: 10.48550/ARXIV.1609.02907. URL: https://arxiv.org/abs/1609.02907.

[34] William L. Hamilton, Rex Ying, and Jure Leskovec. *Inductive Representation Learning on Large Graphs*. 2017. DOI: `10.48550/ARXIV.1706.02216`. URL: `https://arxiv.org/abs/1706.02216`.

[35] Wayne W. Zachary. "An Information Flow Model for Conflict and Fission in Small Groups". In: *Journal of Anthropological Research* 33.4 (1977), pp. 452–473. ISSN: 00917710. URL: `http://www.jstor.org/stable/3629752` (visited on 07/04/2022).

[36] Ulrik Brandes et al. "On Modularity Clustering". In: *IEEE Transactions on Knowledge and Data Engineering* 20.2 (2008), pp. 172–188. DOI: `10.1109/TKDE.2007.190689`.

[37] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: `10.48550/ARXIV.1412.6980`. URL: `https://arxiv.org/abs/1412.6980`.

Ringraziamenti

Questa tesi rappresenta la conclusione di un percorso formativo durato tre anni, che mi ha permesso di costruire nuovi rapporti, sia dentro e fuori l'ambiente accademico.

Innanzitutto, vorrei ringraziare la Prof.ssa Rita Fioresi, che è stata di grande aiuto in quest'ultimo anno. Infatti, oltre ad avermi introdotto nel mondo del geometric deep learning ed ad aver permesso la realizzazione di questa tesi, in molte occasioni i suoi consigli si sono rivelati fondamentali nell'orientamento dei miei futuri studi. Inoltre, è stata di sostanziale importanza nella scelta e nell'ammissione del master, e per questo le sono grato. Ringrazio inoltre Ferdinando Zanchetta per le lezioni introduttorie al geometric deep learning che mi hanno aiutato a velocizzare molto il mio studio e l'attuazione di questa tesi.

Successivamente vorrei ringraziare la mia famiglia, che mi ha sempre sostenuto durante gli studi e mi ha sempre spinto a sfruttare tutte le possibilità che mi venivano offerte.

È d'obbligo a questo punto nominare il gruppo studio Qualcosa, senza il quale non avrei avuto la possibilità di confrontarmi su argomenti di elevata importanza accademica e notevole complessità. In particolare ringrazio di cuore, in ordine, i membri del gruppo: Sere, che in molte, molte occasioni è riuscita ad evitare dei litigi interni al gruppo e a rasserenare la situazione, e infine Dani, il quale non è che abbia portato molti contributi allo studio, ma comunque è stata un presenza importante (BNF).

Infine, vorrei ringraziare due persone distanti dal mondo accademico della fisica, ma che comunque si sono rivelate essenziali durante il mio percorso. Il primo ringraziamento speciale va all'Ali, che mi ha sempre spinto a dare il meglio di me, sia all'università sia fuori. Ultimo ringraziamento è ovviamente per Chucky.