ARISTOTLE UNIVERSITY OF THESSALONIKI

Graph Neural Networks

by

Iakovos Marios Tsouros

A thesis submitted in partial fulfillment for the
Graduate degree

in the
Faculty of Sciences
School of Physics

Supervising Professor: Dr. Panagiotis Argyrakis

Date

Declaration of Authorship

I, [author's name], declare that this thesis titled, [thesis title] and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Inspiring quote goes here (optional)

Quote's attribution

ARISTOTLE UNIVERSITY OF THESSALONIKI

Abstract

Faculty of Sciences
School of Physics

Graduate Degree

by Iakovos Marios Tsouros

Abstract goes here.

Acknowledgements

Acknowledgements go here.

Contents

List of Figures

List of Tables

Abbreviations

Acronym	**W**hat (it) **S**tands **F**or

Dedication (optional)

Chapter 1

Introduction

1.1 Graphs

In this subsection, the main aspects of graph theory are briefly presented.

1.1.1 Introduction

In the real world, many problems can be described by a diagram connecting a set of points with lines, joining pairs of these points, or even creating loops on a single point. A simple example of that would be a set of points representing people with lines connecting acquintances, or points representing atoms and lines representing chemical bonds, creating a representation of a molecule as a graph attribute. In the examples above, the only information contained is whether two points are associated, with the manner being disregarded. The concept of a graph consists of a mathematical abstraction of the above. [5]

Definition 1.1. Mathematically, in its simplest form, a **graph** is an ordered pair[1] $G = (V, E)$ of:

- **V**, a set of vertices (also known as nodes).
- $E \subseteq \{\{x, y\} | x, y \in V \; x \neq y\}$, which is the set of **edges** which consists of unordered pairs of vertices that connect two nodes.

This type of object is called an **undirected simple graph** to avoid confusion with other types.

[1] An ordered pair (a, b) is a pair of objects in which the order of appearance or insertion is significant; the ordered pair (a, b) is different than (b, a) unless $a = b$. An unordered pair is a set of the form a, b is a set having two elements with no relation between them and $a, b = b, a$.

Definition 1.2. A *graph* G is an ordered pair $(V(G), E(G))$ consisting of a set $V(G)$ of *vertices* (also called *nodes* or *points*) and a set $E(G)$, disjoint from $V(G)$ which consists of *edges* (also called *links* or *lines*) together with an incidence function ψ_G that associates with each edge of G an unordered pair of not necesserily distinct vertices of G. If e is an edge and u and v are vertices such that $\psi_G = u, v$ then e is said to *join* u and v, and the vertices u and v are called the *ends* of e. We denote the numbers of vertices and edges G by $u(G)$ and $e(G)$ which two parameters are called the *order* and *size* of G, respectively [5].

In short, we can define a **graph** as an ordered triple $G = (V, E, \phi_G)$ consisting of:

- V, a set of *vertices*
- E, a set of *edges*
- $\phi_G : E \to \{\{x, y\}|x, y \in V \; and \; x \neq y\}$ an *incidence function* mapping every edge to an unordered pair of vertices - an edge associated with two distinct vertices. The incidence function is a function of the edges.

This type of object is called an *undirected multigraph*, to avoid confusion. Note, that the above definition of the *incidence function* does not allow for *loops* (mappings of an edge on the same vertex).

A *loop* is a an edge that allows a connection of a vertex to itself and a graph can be defined to either allow or disallow the presence of loops. Some authors allow for loops to exist on *multigraphs* [6], while other consider these kind of graphs to exist in a different category, called *pseudographs* [7]. Allowing loops requires modifying the incidence function so they can be supported. The new incidence function can be written as:

$$\phi_G : E \to \{\{x, y\}|x, y \in V\} \tag{1.1}$$

The example presented below should better illustrate clarify the definition (of a pseudograph).

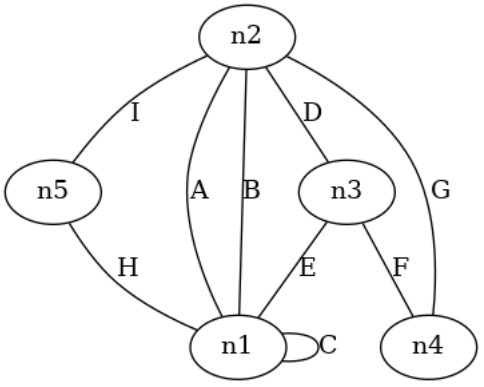


FIGURE 1.1: An undirected pseudograph with labeled nodes and edges.

Example 1.1.

For the graph presented in Figure 1.1 the following can be assumed:

$$G = (V(G), E(G))$$

and

$$V(G) = \{n_1, n_2, n_3, n_4, n_5\}$$
$$E(G) = \{A, B, C, D, E, F, G, H, I\}$$

and the incidence function is defined as:

$$\psi_G(A) = n_1n_2, \quad \psi_G(B) = n_1n_2, \quad \psi_G(C) = n_1n_1, \quad \psi_G(D) = n_2n_3,$$
$$\psi_G(E) = n_1n_3, \quad \psi_G(F) = n_3n_4, \quad \psi_G(G) = n_2n_4, \quad \psi_G(H) = n_1n_5,$$
$$\psi_G(I) = n_2n_5$$

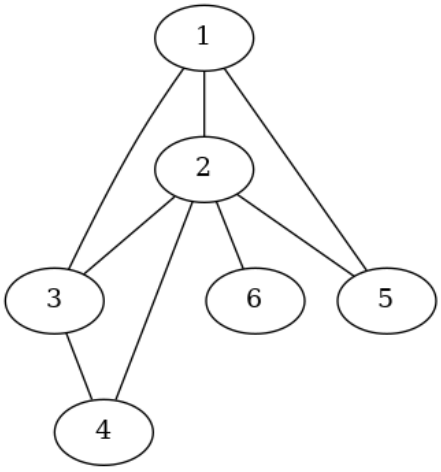
It should now be clear that with the newer definition of ϕ_G, self loops are now possible. Additionaly, even though this was not prohibited by the previous definition, it is worth noting that a node can be connected to another with multiple edges (or multiedges), or that it can have zero connections to other nodes. Generally, V is assumed to be a non-empty set, but E can be empty.

It is now possible to define some characteristic attributes of graphs:

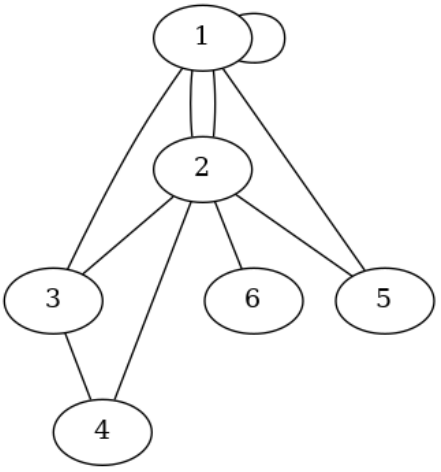
- $|V|$: the **order** of a graph is the number of its vertices.
- $|E|$: the **size** of a graph is the number of its edges.
- The **degree** (or **valency**) of a single node is the number of edges connected to it. The **degree** of a graph is the maximum number of edges connected to a single vertex that belongs to it.
- The edges of create a *homogenous relation*[2] $\sim$ on the vertices of the graph that is called **adjacency relation**; for each edge (x, y), its endpoints *x, y* are said to be **adjacent** to each other, denoted by $x \sim y$. This property will be particulary useful when the adjacency matrix is defined in the following section.

It can be inferred from the above definitions and attributes that for an undirected graph of order *n*, the maximum *degree* of a node is $n - 1$ and and maximum *size* of a graph is $n(n - 1)/2$.

[2] A **homogenous relation** (or **endorelation**) over a set *X* is a set of assignments (binary relations) over *X* and itself; i.e. it is a subset of the cartesian product $X \times X$



(A) Multigraph with no loops and multiple edges.



(B) Mutligraph with loops and multiple edges.

FIGURE 1.2: Two undirected multigraphs.

In this section only *undirected* graphs were considered, which are graphs with edges with no orientation. A whole other class of graph objects with edges which have orientation exists, called *directed graphs*. These kind of graphs objects are out of scope for this thesis and will not be presented.

1.1.2 Adjacency Matrix

Definition 1.3. The *adjacency matrix* is the fundamental mathematical representation of a graph. It is a square matrix, the elements of which represent which pair of nodes are *adjacent* or not. Thus, the adjacency matrix **A** of a graph of order n is the $n \times n$ matrix with elements A_{ij} such that:

$$A_{ij} = \begin{cases} 1 & \text{if there exists at least one edge connecting } i \text{ and } j \\ 0 & \text{if there no edges connecting those edges directly.} \end{cases} \tag{1.2}$$

Considering the simple undirected graph of Figure 1.3a we can construct the following adjacency matrix:

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

TABLE 1.1: Adjacency matrix for Figure 1.3a

For this simple network, which has no loops and only one edge connect two nodes, the diagonal matrix elements are always zero and the matrix is symmetric, as for each edge connecting i and j there is a representation for the j to i connection as well.

In a more complex case, such as the one presented in Figure 1.2b where loops and multiedges are present an adjacency matrix can still be constructed. In this case, a multiedge is represented by setting the value of the corresponding A_{ij} value equal to the multiplicity of the edge. In this case, $A_{12} = A_{21} = 2$.

For loops, the most common representation in the case of undirected graphs is to still set the value of the A_{ii} element equal to 2 (i.e. $A_{11} = 2$ in the example presented). Essentially, an edge of a loop has two ends that connect to the same node, thus the result [8, p. 68]. Additionaly, defining the matrix in such manner, allows for better computations and is consistent with the definition of the representation of an edge connecting two nodes of an undirected graph [9, p. 108].

Thus, the adjacency matrix for the graph of Figure 1.2b is

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

TABLE 1.2: Adjacency matrix for Figure 1.2b

Remark 1.4. Note that the degree of a node can be easily found by summing the values of the column or row of the adjacency matrix that correspond to said node.

1.1.2.1 Adjacency List

An alternative to the adjacency matrix is the *adjacency list.* An adjacency list is a collection of lists, one for each node i. Each list contains the labels of the nodes that i is connected to, and its the most common method for storing networks on computers as it requires less space. It is also possible to represent edge attributes in an adjacency list by appending an extra column which holds these values. An example for the graph presented in Figure 1.2b with multiedges and loops:

Node	Neighbors
1	1,2,2,5,3
2	1,1,3,5,6,4
3	1,2,4
4	3,2
5	1,2
6	2

TABLE 1.3: Adjacency list for Figure 1.2b

Each edge of the network appears twice, thus for a network with m edges the size of the adjacency list would be $2m$, much smaller compared to the $n \times n$ matrix required to build an adjacency matrix. This is particularly useful in networks which are relatively *sparse*[3], but have a high order.

1.1.3 Graph Laplacian

The graph lacplacian is another representation matrix representation of a graph. In its simplest form, for a simple, undirected and unweigheted network of order n, is a $n \times n$ matrix **L** with elements:

$$L_{ij} = \begin{cases} k_i, & \text{if } i = j \\ -1, & \text{if i} \neq \text{j and there exists an edge connecting } i \text{ and } j \\ 0, & \text{everywhere else} \end{cases}$$

where k_i is the degree of the node. Another way to write the graph Lacplacian is as

$$\mathbf{L} = \mathbf{D} - \mathbf{A}$$

where **D** is a diagonal matrix containing the degrees of the nodes.

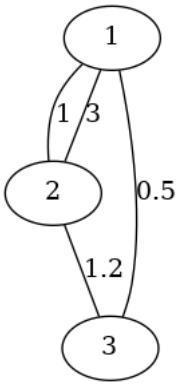
[3] *Sparse* networks are networks with a much lower number of edges than those possible.

In a similar manner the Laplacian matrix can be constructed for weighted networks, by replacing the the degree k_i of a node with the relevant matrix elements.

The Laplacian matrix has many applications in the study of dynamical systems, random walks and graph visualization. It has also found applications in graph neural networks, as its spectral decomposition allows the construction of low dimensional embeddings with applications in graph neural networks, such as ChebNet [10] which will be discussed in depth later.

1.1.4 Edge weights

So far, while presenting edges, we have considered graphs where connections between nodes represented binary relations between them; they either existed or they did not. In many situations when studying graphs, it is useful to represent edges as connections which carry some kind of attribute or value, commonly called *weight*. This weight could be any real number that fits a particular example, such as the distance between two airfields on an airline network, the kinship of connections on a social network (negative values can represent animosity and vice versa) or any type of relational attribute that can be quantified and characterizes the connection between nodes that belong in the same network[9, p. 109]. A simple example is presented in the figure below.



(A) Multigraph with no loops and multiple edges.

$$A = \begin{pmatrix} 0 & 4 & 0.5 \\ 4 & 0 & 1.2 \\ 0.5 & 1.2 & 0 \end{pmatrix}$$

(B) Corresponding adjacency matrix.

FIGURE 1.3: Simple example of an unordered graph with weighted edges

Generally, edges and nodes can hold any type of variable as values, such as vectors, the usefulness of which will become apparent when computer algorithms for graph representations and graph neural networks are discussed in later sections and chapters.

1.1.5 Distance between nodes and shortest paths

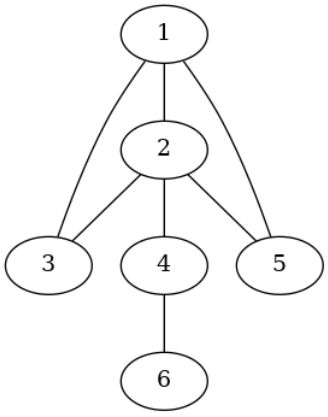
On a graph, any route that traverses nodes along the edges connecting them creates a sequence which is called a *walk*. Walks are not prohibited from traversing previously visited nodes and edges, but walks that do not intersect themselves are called *paths*.

Remark 1.5. *Adjacency Matrix Powers* Before continuing, it is worth mentioning that the powers of the adjacency matrix A^c directly provide the number of walks of length c among two nodes. If there is a connection between two nodes i and j, then $A_{ij} = 1$ else it is 0. Moving to the second power and taking for example an intermediate node k which might lie between i and j, the product $A_{ik}kj$ would be 1 if there is a node and 0 if there is not [9, p. 131]. Generalizing to walks of length c which traverse nodes i to j, their total number is:

$$N_{ij}^{(c)} = [A^c]_{ij}$$

The *shortest path* between two nodes is the shortest walk between those two nodes, the walk which traverses the least amount of nodes. In terms of edges, the least number that must be traversed is called *shortest distance* or often just "distance". Mathematically, the shortest distance between two nodes i and j is the walk with the smallest value c where $[A^c]_{ij} > 0$

Example 1.2. *For instance consider the following graph:*



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

FIGURE 1.4: A graph with a maximum path of 3 (nodes 1 to 6).

In this example, it can be visiually determined that the minimum walk between nodes 1 and 6 is of length 3. Indeed, raising the adjacency matrix to a power of three yields for these nodes:

$$\dots$$

$$N_{1,6}^{(2)} = \sum^{n} A_{1k}A_{k6} = [A^2]_{1,6} = 0$$

$$N_{1,6}^{(3)} = \sum^{n} A_{1k}A_{kl}A_{l6} = [A^3]_{1,6} = 1$$

1.1.6 Node and edge properties

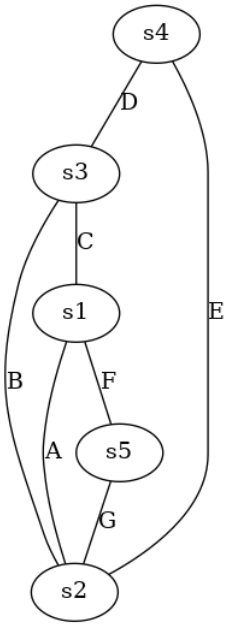
As mentioned in Section 1.1.4, edges can hold more information than just the binary relations between nodes. In fact, this concept can be generalized to nodes and even whole graphs. When studying graphs and networks, especially when using computational methods for applications like complex dynamics, it is the most natural way to phrase data on a graph. The data can be any real number or even categorical data, such as colors.

The matrices which hold the information mentioned before are

- **V: vertex (or node) attributes** (i.e. a label or number of neighbors)
- **E: edge (or link) attributes** (i.e. a label or edge weight)
- **U: global (or master node) attributes** (i.e. number of nodes or number of paths of length 2)

Global attributes usually get their values as an aggregation of the attributes of the nodes and edges, and methods applied on them. For instance, a molecule might have chemical elements as node attributes, types of bonds as edge attributes and the toxicity of the molecule as graph attributes. An example of a classroom should better illustrate the concept.

Example 1.3. *In this example, a classroom of 5 classmates is presented. Each student has a number of* ***node*** *attributes, their age, heigh and average grade (from F to A).* ***Edge*** *attributes between students hold information about their physical proximity when seated for class, and their kinship (as a real number between* $0 - 1$*). Finally,* ***global*** *attributes consist of information about the classroom, such as total number of students and class's failure rate.*



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

FIGURE 1.5: Example graph of a small classroom with labeled edges and nodes

Node	Age	Height	Grade
s1	11.5	135	C
s2	12	140	B
s3	12	142	A
s4	11.5	132	A
s5	12	143	B

(A)

Edge	Distance	Kinship
A	1.5	0.5
B	1.2	0.5
C	0.5	0.8
D	2	0.35
E	2	0.3
F	0.5	0.9

(B)

TABLE 1.4: A) Node properties B) Edge properties

	No. Nodes	Failure %
G	5	0

TABLE 1.5: Graph Properties

1.2 Network Dynamics

Frequently, it is useful to consider cases where the status of the networks studied changes over time. This could mean that the topology of the network changes (i.e. nodes and edges are added or removed), the internal state of the network changes (the properties discussed in the previous section) or both.

The main concern of this thesis is with networks with a fixed topology , but with elements (nodes and edges) whose properties constitute dynamical quantities which can change over time. Following some dynamical rule, nodes can interact with their neighbors or change their own properties, with edges dictating which interactions are possible. In fact, the study of network dynamics combines graph theory with non-linear dynamics [11].

For many real world situations, a proper model of their processes consists of dynamical systems acting on networks. This can range from opinion spreading, epidemics, flow of electricity on grids, spread of packages on internet networks and many more systems whose dynamics are evolving on a network. In fact, many network processes can be understood

1.2.1 Simple Contagion Dynamics (SIS)

In this model, the simple contagion dynamics called susceptible-infected-suspectible (SIS) are considered, a stochastic model. Here, the state of a node has a chance to transition from one state to another based on either a function of the states of its neighbors or a static probability. Specifically, the set of all possible node states are $\mathcal{S} = \{S, I\} = \{0, 1\}$ with S $\rightarrow$ suspectible and $\rightarrow$ infected. The dynamics are stochastic so the set of all possible node outcomes is $\mathcal{R} = [0, 1]^2$. The infection function f_{inf} is defined as the probability of a susceptible node becoming infected, based on a the number of infected numbers n_{inf}:

$$Pr(S \rightarrow I | n_{inf}) = f_{inf} n_{inf} = 1 - (1 - \gamma)^{n_{inf}} \tag{1.3}$$

where γ is the disease's transimission probability $-$ a constant parameter. The recovery probability is constant and defined by:

$$Pr(I \rightarrow S) = \beta \tag{1.4}$$

Therefore, it is possible to define the node outcome function of the SIS dynamics as a function of a node state its neighbors' x_i, x_{N_i}:

$$f(x_i, x_{N_i}) = \begin{cases} (1 - f_{inf}(n_{inf_i}), f_{inf}(n_{inf_i})), & \text{if } x_i = 0, \\ (\beta, 1 - \beta), & \text{if } x_1 = 1, \end{cases} \tag{1.5}$$

where $n_{inf_i} = \sum_{v_j \in N_i} \delta(x_j, 1)$ the number of infected neighbors of node v_i. Note that the output of the Equation 1.5 is a two dimensional probability vector where the "column" is the probability that it becomes or remains susceptible and the second is the probability that it becomes or remains infected.

1.2.2 Majority Rule — Opinion Spreading

The majority rule is one of the simplest opinion spreading dynamics models, where each node adopts the state of the of the majority of its neighbors. It is a deterministic model. The set of all possible node states is $\mathcal{S} = \{0, 1\}$ and outcomes $\mathcal{R} = \{0, 1\}$.

The node outcome function of this type of dynamics takes the form of the majority function from boolean logic, which evaluates to false if more than half of its arguments are false and true otherwise:

$$f(x_i, x_{N_i}) = \left\lfloor \frac{1}{2} + \frac{(\sum_{j \in N_i} x_j) - 1/2}{N_i} \right\rfloor \tag{1.6}$$

where $\lfloor \cdots \rfloor$ denote the floor operation.

1.3 Models of network formation

When studying networks it is important to be able to reproduce the complexity of a real system. To this end, algorithms which attempt to recreate some form of realistic networks have been proposed. In this thesis, the two methods used are the Barabási-Albert model by Albert and Barabási [12] and the Erdős–Rényi model by Erdős et al. [13].

1.3.1 The Barabási-Albert Model

This model usually starts with a small number of connected nodes, m_0. Any new node added to the network connects to m (where $m \leq m_0$) nodes and the probability that the newly added node connects to node i is:

$$p_i = \frac{k_i}{\sum_j k_j} \tag{1.7}$$

where j is the total number of nodes already in the network and k is the degree. As a result, nodes which are heavily linked tend to quickly accumulate more links. This type of network is obviously always fully connected.

1.3.2 Erdős-ényi Model

Two different but closely related types of this model exist.

The $G(n, M)$ Model

Chapter 2

Neural Networks

2.1 Historical Background

Artificial Neural Networks (ANN), or sometimes simply called **neural networks** is a class of computational models that mimic the way biological neural networks work, such as the human brain. Interest on the subject sparked after the seminal paper *"A Logical Calculus of the Ideas Immanent in Nervous Activity"* [14] by Warren McCulloch and Walter Pitts, where they proposed a computationally functional model of neural networks. Their suggestions showed that in principle, any function a digital computer can compute, a neural net should too. The models they described had weights and thresholds, but they lacked a training method.

The suggestions of McCulloch and Pitts lead Frank Rosenblatt to create the *Perceptron* in 1958 [15], a binary classifier algorithm based on supervised learning[1]. Although initially promising, single layer perceptrons were not able to train on multiple classes of patterns and were eventually proven incapable of learning a XOR function [2] in the book *Perceptrons* [16], as the way they worked was by "separating" data linearly. This lead to a stagnation in machine learning research dubbed "AI winter", until the proposal of **backpropagation**[3] by Paul John Werbos in 1975 [17].

A renewed interest in the field lead to the development of the Cresceptron [18] in 1992, a method of training large networks with pooling layers (**max-pooling**) and down-sampling. GPU[4] usage made possible the training of larger networks, while new types of networks

[1] Supervised learning is a machine learning training technique that optimizes a model based on examples input-output pairs.

[2] XOR (Exclusive or, $x \oplus y$) is a logical operation that is true only if its arguments differ.

[3] Backpropagation is a method of fine tuning a neural network based on the error rate obtained from previous runs of the program. It will be discussed in detail later in this thesis.

[4] GPU - Graphics Processing Units is a specialized electronic circuit, a central part of modern computers which excels in efficient computation of algorithms which process large blocks of data parallelly, thus exceling in machine learning applications.

emerged such as the **Recurrent Neural Networks (RNNs)**. **Convolutional Neural Networks** have recently proven to be far superior for image classification tasks.

In recent years, neural **TODO: ADD INFO HERE**

2.1.1 Basics

2.1.1.1 Summary

One can think of ANNs as a directed graph, with a collection of nodes which are densely connected (called **artificial neurons**), transimiting signals to each other. These nodes are usually organized in sets of layers, with signals moving in one direction (i.e. *feed forward networks*) through weighted connections. Signals received on a single neuron are real numbers, and the output of a single neuron is the output of an aggregation of a non-linear function of the sum of its inputs. This function is called an *activation function* and its results are propagated to all of the nodes outgoing connections. Thus, each neuron can be thought as a simple processing unit. The weighted connections between nodes might have an excitatory or inhibitory effect, based on these weights which can be positive, negative or very close to zero, having no effect [19, Chap. 1]. While training, example data is passed through the input layer and gets radically transformed through the layers until it reaches the output layer. The weights and other trainable parameters are then adjusted until the training data consistently reaches satisfactory results.

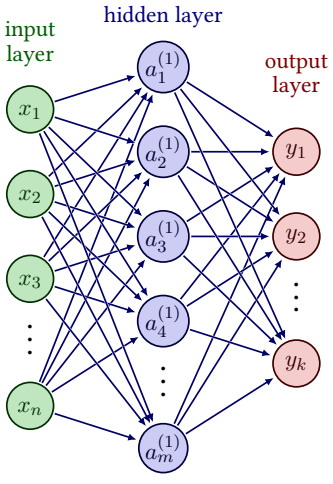


FIGURE 2.1: A simple neural network demonstrating the layered structure and and flow of data from input to output.

2.1.1.2 Building Blocks

As mentioned before, the building blocks of ANNs are its artificial neurons organized into layers. In Figure 2.1 a basic ANN is presented, demonstrating the different layers that are typically present in a feedforward neural network [5] while the building blocks are widely considered [20] to be:

- **Input layer** This layer's purpose is to act as an entrypoint to the neural network and performs no computations. Data moves from this layer to the hidden layer succeeding it.
- **Hidden layer** Networks typically have one or more hidden layers, in which some computation takes place. Data moves from the input layer to a hidden layer where it is transformed and together with its weights moves to the next hidden layer or the output layer.
- **Output layer** Transformed data "exits" the network here, where it can pass through some function to reach the desired output format
- **Edges and Weights** Each node in a layer is connected to a set (usually all) of the nodes of the following layer with a weighted edge. Data from nodes $1...n$ of layer k will be the input of node j of layer l, multiplied by a weight W_{ij}.
- **Activation Functions** An activation function takes as input some form of aggregate (usually the weighted sum) of the signals arriving at a node and produces an output. This function is typically nonlinear and differentiable for reasons which will be discussed later.
- **Learning** The learning process in a ANN involves modifying its weights and other learnable parameters to improve the accuracy of the result on the output layer. Learning usually involves a cost function which is evaluated on a predefined basis and adjustments are made accordingly. One of the most widespread learning techniques is *backpropagation*, where the error is propagated backwards through the network.

Along with the data from previous layers aggregated at a neuron, a bias is typically added which acts in the same way the intercept does in a linear equation. It adjusts the output of the activation function along with the weighted sum of the inputs to the neuron. It is also a trainable constant value provided to each node of a layer. Biases are node level parameters and do not depend on values provided by previous layers.

[5] Feedforward Neural Networks (FNNs) are the simplest type of neural networks, with the information moving only in one direction ("forward" through the layers), without any loops or cycles [20]. Different types of neural networks are discussed later.

2.2 Mathematics useful in ANNs

In this section some mathematical constructs which are widely used in machine learening and ANNs are briefly presented.

2.2.1 Gradient Descent

Gradient based optimizations methods are of great importance to NNs as they are most common method of training these models. They are tasked with minimizing or maximizing some function $f(\boldsymbol{x})$ which is oftenly called **objective function**. In NN training scenarios, where the goal is to minimize it the same function is also commonly called a **cost**, **loss** or **error** function[21].

Gradient descent is a method of minimizing a function $f(x)$ and finding a local minimum, given that its differentiable. The derivative $f'(x)$ is the slope of $f(x)$ at x, so it specifies how a change in x would provide a step towards the local minimum. For instance, for small values of ϵ, $f(x - \epsilon \operatorname{sgn}(f'(x)))$[6] is less than $f(x)$ and traversing the slope to ever decreasing values if possible through following the direction with the opposite sign of the derivate. This method is called **gradient descent** and it was first proposed by Cauchy [22] in 1847.

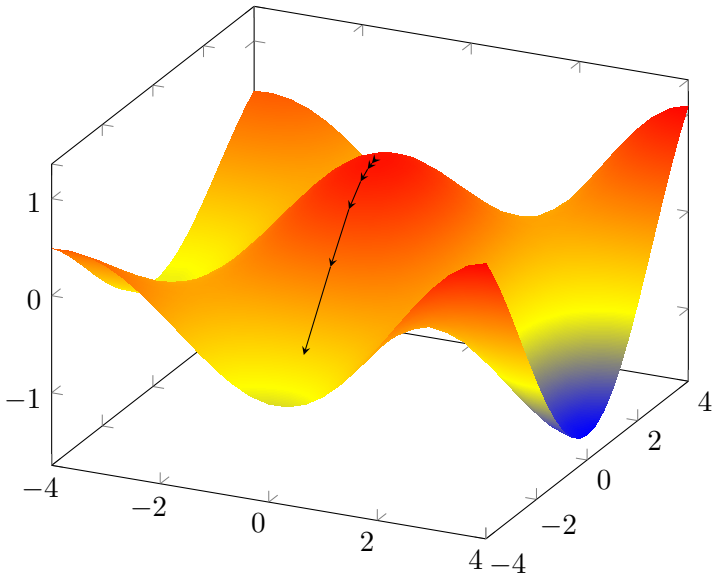


FIGURE 2.2: Gradient descent in three dimensional space

Definition 2.1. Consider a multi-variable function $F(\boldsymbol{x})$ which is **defined** and **differentiable** in a neighborhood of a point a. The value of $F(\boldsymbol{x})$ will decrease the fastest when moving from

[6] sgn is the *signum function*, a piecewise function which returns the sign of its input or 0 if input is 0. (i.e. $\operatorname{sgn}(-1) = -1$ and $\operatorname{sgn}(12) = 1$).

a towards the negative gradient direction of F, which is $-\boldsymbol{\nabla} F(\boldsymbol{a})$. Thus when:

$$\boldsymbol{a}_{n+1} = \boldsymbol{a}_n - \gamma \nabla F(\boldsymbol{a}_n) \tag{2.1}$$

then $F(\boldsymbol{a}_n) \geq F(\boldsymbol{a}_{n+1})$ and therefore the values of $F(\boldsymbol{a})$ move towards the local minimum. A sequence $\boldsymbol{x_0}, \boldsymbol{x_1}, \boldsymbol{x_2}, ...\boldsymbol{x_m}$ that follows the rule set by Equation 2.1 will lead to the monotic sequence $F(\boldsymbol{x}_0) \geq F(\boldsymbol{x}_1) \geq F(\boldsymbol{x}_2), ...$ and the sequence $\boldsymbol{x_n}$ will converge to the local minimum. If some special choices are made for γ[7] the function is guaranteed to reach a local minimum. Additionaly, if the function is *convex* local minima are global minima and gradient descent converges to a global solution.

For a function to be convex, a line segment connecting two of its points must lay on or above its curve. Mathematically for two points x_1 and x_2, this can be expressed as

$$f(\lambda x_1 + (1-\lambda)x_2) \leq \lambda f(x_1) + (1-\lambda) f(x_2) \tag{2.2}$$

where λ is a location on a section line and $0 \leq \lambda \leq 1$.

2.2.2 Hadamard Product

The **Hadamard product**, also known as the **element-wise product** is a matrix operation in which two matrices of the same dimensions are multiplied to produce a matrix of equal dimensions, where its i, j elements are the product of elements i, j of the original two matrices. For two matrices A and B of dimensions $m \times n$ the Hamarand product can be written as:

$$(A \odot B)_{ij} = (A)_{ij}(B)_{ij} \tag{2.3}$$

2.3 Feedforward Networks

Feedforward neural networks (FNNs) are the exemplary of ANNs, and the first to be conceived and created by Rosenblatt in 1958 with the creation of the perceptron [15]. Their goal is to approximate some function f^*; i.e. a classifier uses the function $y = f^*(\boldsymbol{x})$ to map the input $\boldsymbol{x}$ to some category y. The goal of a feedforward network would then be to define a mapping $\boldsymbol{y} = f(\boldsymbol{x};\ \boldsymbol{\theta})$ and train in a way that the values of the parameter vector $\boldsymbol{\theta}$ provide the best approximation of the function f^*.

[7] γ in machine learning is also known as the "learning rate" and its one of the hypermateres of NNs. Special choices made here include a selection of γ via line search which satisfies the Wolfe conditions or the Barzilai-Borwein method [23]

These networks are called feedforward as information flows from the input layer $\boldsymbol{x}$ through intermediate computational layers used to define $\boldsymbol{f}$ and finally to the output $\boldsymbol{y}$. There are no loops providing (called **feedback connections**) information from the output back into the input or other intermediate layers of the network.

Feedforward networks can be thought as a chain of functions, composing the final structure of the network. As an example, consider a network with three of these functions as $f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$. In this case, the exponents denotes the layer, with $f^{(1)}$ being the first layer, $f^{(2)}$ the second etc. The total number of these functions is called the **depth** of the NN and the terminology "deep learning" arose from this layered structure.

During training the goal is to modify the parameters of the network in a way that $f(\boldsymbol{x})$ closely matches $f^*(\boldsymbol{x})$. The training set is composed of pairs of $\boldsymbol{x}$ and labels $y \approx f^*(\boldsymbol{x})$ and the output of the network is evaluated at different training points. Training data defines the exact result expected from each input $\boldsymbol{x}$, a value as close as possible to y. The rest of the layers can have arbitary behaviours as long as they transform the data in a way defined by the training goal. The learning algorithm can "use" them in any way that is useful to it, and the training data has no immediate effect on their behaviour. They are thus called **"hidden layers"** as they do not produce any meaningful result, related to the function [21, p. 160].

2.3.1 Perceptrons, learning algorithms and simple NNs

The perceptron serves as great precursory example to neural networks, as it introduces many concepts that are common in more complex NN paradigms. It is in fact a simple neuron, a computational unit, which takes a binary vector as input and produces a binary output.

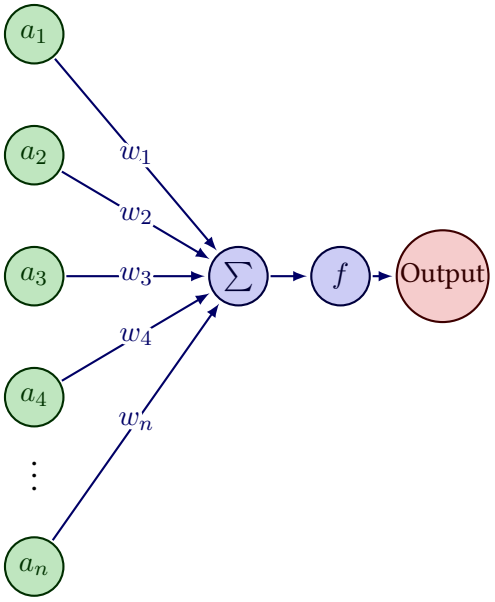


FIGURE 2.3: A simple perceptron

Inputs are multiplied with *weights*, and the output of the neuron is either 0 or 1, determined by whether the weighted sum of its inputs $\sum_i w_i a_i$ is greater than a threshold value $threshold$. All of these numbers are real numbers and a parameter of the neuron itself. The algebraic form of this can be written as:

$$\text{Output} = \begin{cases} 0 & \text{if } \sum_i w_i a_i \leq \text{threshold} \\ 1 & \text{if } \sum_i w_i a_i > \text{threshold} \end{cases}$$

By modifying the values of the threshold, the output can be changed. The above equation can be written in a more compact form by replacing the sums with the dot product of the weights and the input vector, $\sum_i w_i a_i = \boldsymbol{w} \cdot \boldsymbol{a}$. Moving the threshold to the left side of the equation and replacing it by what is referred to as the *bias* yields:

$$\text{Output} = \begin{cases} 0 & if \boldsymbol{w} \cdot \boldsymbol{a} + b \leq 0 \\ 1 & if \boldsymbol{w} \cdot \boldsymbol{a} + b > 0 \end{cases}$$

where b = - threshold. The bias can be seen as a measure of the tendency of the neuron to fire. Larger bias numbers makes the neuron easy to activate and output 1, while smaller ones require larger inputs and positive weights.

Remark 2.2. As mentioned before, perceptrons are able to solve problems which are linearly seperatable with the slope represented by the $\boldsymbol{w} \cdot \boldsymbol{a}$ term and the bias acting as the intercept. This excludes problems which are not, the most famous being the classic XOR gate, as there does not exist one single line capable of seperating the predictions.

Multiple perceptrons can be combined in a network to compute any logical function, including the XOR gate. These kind of NNs are called *multilayer perceptrons*, and they are the basis modern artificial neural networks. As discussed before, these neurons (or nodes) are organized in layers and can have a depth based on the number of these layers. Typically, at least 3 layers are present.

A more practical activation function

So far, the way perceptrons transform data through a function has been described, but while they can produce sensible results as any other computing device, weights and biases had to be manually configured. It is possible to introduce a learning algorithm which does the tuning of these parameters automatically, in response to input-output (training data) pairs.

The perceptrons described used the Heaviside step function [24], often denoted with H. This function has a binary nature, with its output value being 0 or 1 based on a threshold, which presents a problem when fine tuning a perceptron or a NN based on them. Minor changes in a weight or bias value can completely alter the output and trigger a perceptron to flip, possibly changing the output of the whole network. A better choice for an activation function is the *sigmoid function* σ also called the *logistic function* which replaces the perceptron with the sigmoid neuron [25].

The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1+e^{-x}} = \frac{1}{1+exp(-\sum_j w_j x_j - b)} \tag{2.4}$$

It is easy to see the differences between these two functions when plotted:

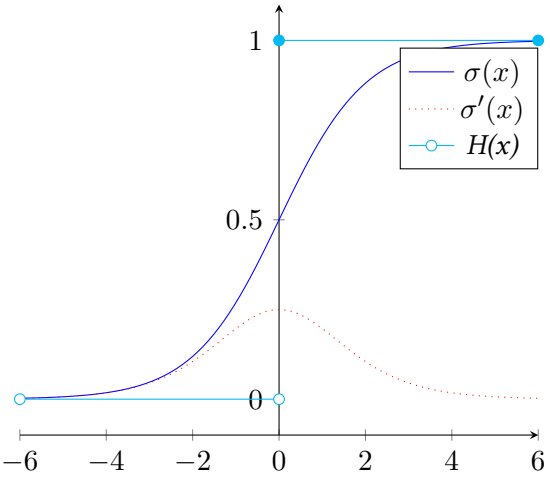


FIGURE 2.4: Plot of the sigmoid function, its first derivative and the Heaviside step function.

For very negative and very positive values of the inputs, the sigmoid approximates the behaviour of the step function, while for inputs around a neighborhood of 0 it differs by offering a smooth transition between 0 and 1 as an output. It also is a differentiable function. This properties allows an approximation of the output when the weights and biases change slightly as:

$$\Delta_{\text{output}} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b$$

This provides an estimation of how small changes in the biases and weights will affect the output of the sigmoid neuron, making choices for their values easy to calculate.

As stated before, all of the computations that take place on a neuron can be represented in a compact matrix form, as shown in Figure 2.5.

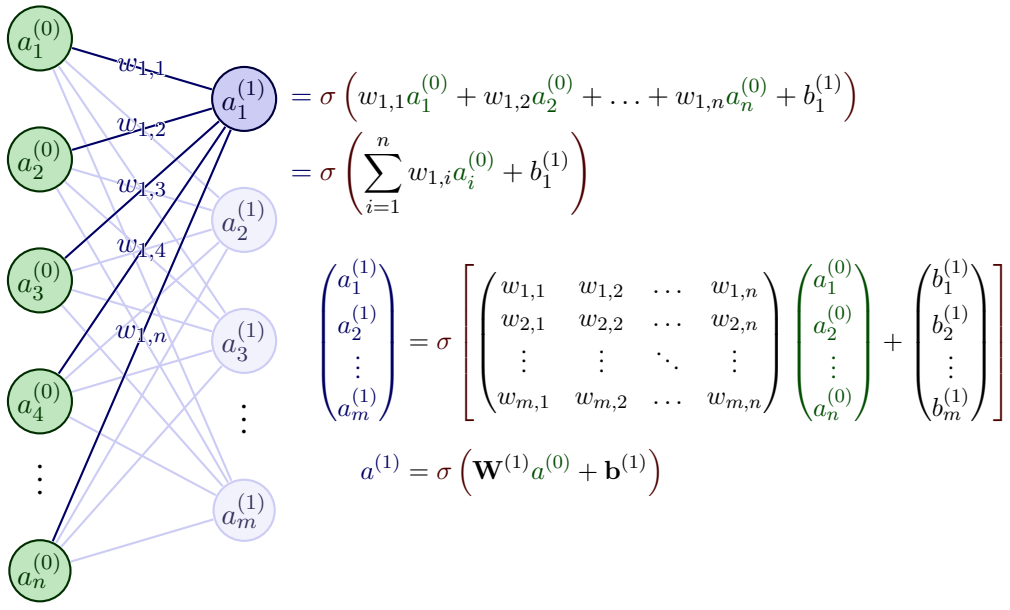


FIGURE 2.5: Input to a single neuron in a feedforward network. Here σ represents the activation function (a sigmoid function in this case) and the exponents represent layers. The activation of a layer can be conveniently represented in matrix form. Biases are added to the input of the node.

Learning

Before defining how training works for a neural network, it is imperative to describe an algorithm of quantifying far from the target output the network is with its current biases and weights. This algorithm is the *cost function* described in Section 2.2.1.

If for all inputs in a vector $\boldsymbol{x}$ the desired output is known and is the output of some function $f^*(\boldsymbol{x})$ or $y(\boldsymbol{x})$ and the cost function can be defined as:

$$C(\boldsymbol{w}, \boldsymbol{b}) = \frac{1}{2n}\sum_{x} ||y(\boldsymbol{x}) - \boldsymbol{a}||^2 \tag{2.5}$$

In this equation, $\boldsymbol{w}$ and $\boldsymbol{b}$ are the weights and biases of the network respectively and $\boldsymbol{a}$ is a vector of the outputs when $\boldsymbol{x}$ is the input.

This cost function is also known as *quadratic* or *mean squared error (MSE)* and it is one of the most commonly used loss functions. The goal of training is to minimize this function, and to achieve this gradient descent and variations of this method are used.

Applying gradient descent to Equation 2.5 yields:

$$\nabla C = \frac{1}{n}\sum_{x} \nabla C_x \tag{2.6}$$

where x are the training samples.

Stochastic Gradient Descent (SGD)

One of the most common variations is *Stochastic Gradient Descent (SGD)*. As training sets get larger, they become computationally expensive especially considering that in Equation 2.5, the total cost function $C = \frac{1}{n}\sum_x C_x$, averaging the costs of each training sample. The gradient of this function requires the computation of each ∇C_x for each sample and then averaging them as $\nabla C = \frac{1}{n}\sum_x \nabla C_x$ which can be prohibitive timewise.

SGD offers a solution to this problem, by drastically simplifying the above process; instead of finding the exact value of ∇C , it estimates its value by randomly picking one sample [26]. This stochastic approximation adds noise, but it has been proven to almost ensure convergence under mild conditions [27].

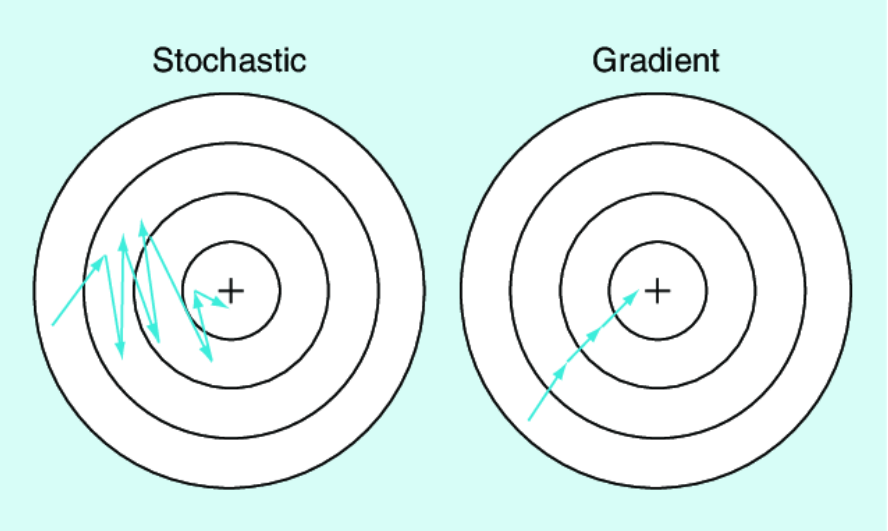


FIGURE 2.6: Noisy convergence of SGD compared to non-stochastic. Image source [1].

A smoother convergence by choosing a small number m of samples which is called a **mini-batch** and is the method most commonly used as it offers a compromise between speed and smoothness of convergence. In this case, Equation 2.6 becomes:

$$\nabla C \approx \frac{1}{m}\sum_{i=1}^{m} \nabla C_{X_j} \tag{2.7}$$

Training rules

We can then define a training rule for the network in terms of weights and biases and using the gradient of the cost function. Applying SGD on Equation 2.5 provides the values of weights w_k and biases b_l that minimizes it. An update rule for them can now be defined in terms of the

cost function:

$$w_k \to w_k^{'} = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial w_k}$$
$$b_l \to b_l^{'} = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{x_j}}{\partial b_l} \tag{2.8}$$

where η is a small positive value, the **learning rate**.

Having described the above, the process of training the NN can be summarized as:

1. The SGD is calculated for a random set number of training samples, the mini-batch.
2. Weights and biases are updated based on Equation **??**.
3. The previous two steps are repeated until the training samples are exhausted.
4. When all samples are exhausted, the training **epoch** is complete, and the process starts again from the first step.
5. Training is usually said to be complete, when the value of the cost function is below a set threshold. The NN is now considered trained.

2.3.2 Backpropagation

Backpropagation is a method of computing the gradient of the loss function with respect to all the weights and biases of the network, based on single training examples, in contrast to more naive methods which compute the gradient on each weight or bias individually.

Even though these days the term *backpropagation* is sometimes used colloquially to refer to the whole process of the learning algorithm, strictly it is just an algorithm to compute the gradient, while SGD can be used for the training itself [21].

Historically, this method has been rediscovered multiple times, as early as 1960 [20] with its first implementation as software in 1970 [21, p. 229], albeit as a method for automatic differentation with no mention to neural networks. In 1974, Werbos proposed its usage in training neural networks[28], but it was not until 1986 when Rumelhart, Hinton, and Williams [29] published their seminal paper "Learning representations by back-propagating errors" which experimentally showed that backpropagation can be used to train deep neural networks faster than any existing technique up to that point.

Ultimately the goal of backpropagation is to compute the partial derivatives $\frac{\partial C_X}{\partial w_{jk}^l}$ and $\frac{\partial C_X}{\partial b_j^l}$ — the partial derivates of the cost function with respect to weights and biases in the network.

Matrix Notation

Before introducing the equations of backpropagation, it is important to present a matrix notation for quantities in a network which are used widely and unambigiuous. Consider the equations presented in Figure 2.5. For a network with L layers, the connection of the k^{th} neuron in the $(l-1)^{th}$ layer with j^{th} node in l^{th} layer can be written as w^l_{jk}. In a similar fashion, the bias of the j^{th} node in the l^{th} layer can be represented as b^l_j and the *activation*[8] of the same node is a^l_j. With this representations declared it is possible to define the matrix representation of a layer l as:

$$\boldsymbol{a^l} = \sigma(\boldsymbol{w^l a^{l-1}} + \boldsymbol{b^l}) \tag{2.9}$$

where σ is the activation function. At this point the intermediate quantity $\boldsymbol{z^l} = \boldsymbol{w^l a^{l-1}} + \boldsymbol{b^l} \rightarrow \boldsymbol{a^l} = \sigma(\boldsymbol{z^l})$ commonly called the *weighted input* is worth declaring.

Fundamentals of Backpropagation

Backpropagation has two special requirements for the cost function. The first is that it must be able to be written as an average $C = \frac{1}{n}\sum_X C_X$ of the cost functions of individual training samples x. This ensures that after computing the partial derivatives $\frac{\partial C_X}{\partial w}$ and $\frac{\partial C_X}{partial b}$ over the training set, it is possible to calculate $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ by averaging over the set.

The second requirement is that the cost function C can be written as a function of the outputs of the NN. For example, the MSE cost function described in Equation 2.5 for a single training example can be written as

$$C = \frac{1}{2}||\boldsymbol{y} - \boldsymbol{a^L}|| = \frac{1}{2}\sum_j (y_j - a^L_j)^2 \tag{2.10}$$

which in fact can be regarded a function of the outputs $\boldsymbol{a^L}$ alone, as the training examples $y(x)$ have fixed values. Note that L denotes the number of layers in the networks.

Equations of backpropagation

Before declaring the fundamental equations of backpropagation, an important intermediate quantity called the **error** δ^l_j must be declared. In a neuron j, during activation, a small quantity Δz^l_j is added as input and the neuron outputs $\sigma(z^l_j + \Delta z^l_j)$. This error is propagated through

[8] The activation of a neuron simply refers to the output of the activation function, taking as input the weighted sum of the inputs and the bias $a' = \sigma(wa + b)$

the network and finally the overall cost changes by $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$. The error can the be defined as:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l} \tag{2.11}$$

The fundamental equations which govern the backpropagation process are described below.

1. The first equation describes the error in the **output layer**:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L) \tag{2.12}$$

   or equivalently in matrix form:

$$\delta^L = \boldsymbol{\nabla_a C} \odot \sigma'(\boldsymbol{z^L}) \tag{2.13}$$

2. An equation for the error in terms of the error in the **next layer**:

$$\delta^l = ((\boldsymbol{w^{l+1}})^T \boldsymbol{\delta^{l+1}}) \odot \sigma'(\boldsymbol{z^l}) \tag{2.14}$$

   This equation is convenient as it propagates the error backwards in the network. Knowning the error in the $(l+1)^{th}$ layer and by transposing the weight matrix the error can then move backwards through the activation function in layer l. This process can be repeated all the way back.

3. An equation describing the rate of change of the cost with respect to **biases** in the network:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \xrightarrow{2.14,\ 2.13} \frac{\partial C}{\partial b} = \delta \tag{2.15}$$

4. An equation describing the rate of change of the cost with respect to **weights** in the network:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{2.16}$$

Finally, Equation 2.15 and Equation 2.16 have been derived and can now be used in SGD or other learning algorithms.

The importance of backpropagation as a method arises from the fact that it is possible to compute all the partial derivatives of the cost function with just one forward and one backward pass through the network. Before its emergence, training a network required computing these derivatives for each weight and bias individually, which was prohibitively time consuming for larger networks.

Learning algorithm using backpropagation

The backpropagation algorithm can be represented in pseudocode as shown below.

Algorithm 1: Backpropagation Algorithm

Data: $\boldsymbol{x}$ the activation of the input layer $\boldsymbol{a^1}$
Result: The gradient of C in terms of $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$

begin
  $layers \leftarrow [2, 3, ...L]$
  `// Feedforward pass`
  for *l in layers* **do**
    $z^l \leftarrow w^l a^{l-1} + b^l$
    $a^l \leftarrow \sigma(z^l)$
  end
  `// Output: Error vector`
  $\boldsymbol{\delta^l \leftarrow \nabla_a C \odot \sigma'(z^L)}$
  `// Backpropagate the error`
  $layers\_reverse \leftarrow [L-1, L-2, ...2]$
  for *l in layers_reverse* **do**
    $\boldsymbol{\delta^l \leftarrow (((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)}$
  end
  `// Output`
  $\frac{\partial C}{\partial w^l_{jk}} \leftarrow a^{l-1}_k \delta^l_j$
  $\frac{\partial C}{\partial b^l_j} \leftarrow \delta^l_j$
end

Consequently, combining backpropagation and a learning algorithm such as a SGD, it is possible to train a network. For instance, in a SGD with m mini-batches case, the training rules, as described in Equation 2.8, can be represented with the following pseudo-code:

Algorithm 2: Training Rules using SGD with backpropagation

Data: Sets of training examples
Result: Updated weights and biases based on gradient descent of the cost function
begin
  $a^{x,l} \leftarrow$ set the activation for each sample
  `// Pass the activation to the backpropagation from Algorithm 1 and get the error as output`
  $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$
  `// Gradient Descent and updates for weights and biases`
  $layers\_reverse \leftarrow [L-1, L-2, ...2]$
  for *l in layers_reverse* **do**
    $w^{l'} \leftarrow w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$
    $b^{l'} \leftarrow b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$
  end
end

2.3.3 Common ANN Architectures

So far, the models discussed were simple perceptrons and multi-layer perceptrons (MLPs), networks which are fully connected — each neuron in a layer is connected to all the neurons of the next layer (i.e. see Figure 2.1). This architecture does not take into account that data used as input might carry some spatial structure, as in the case of image recognition and computer vision. In this section, convolutional neural networks will be introduced briefly, which take into account the spatially local input patterns of some datasets, while at the same time reducing the number of free parameters allowing for deeper networks. While the origin of these types of networks can be traced to 1970, they were first established as a concept by LeCun in the paper "Gradient-based learning applied to document recognition" from Lecun et al. [30].

2.3.3.1 Convolutional Neural Networks (CNNs)

As mentioned before, CNNs take advantage of the spatial structure that some types of data have. For instance an image has a grid-topology, which is ignored in the case of simple or deep MLPs and as a consequence it inhibits learning by treating pixels which are spatially far, in the same manner. CNNs architecture solve this problem by taking into account the spatial characteristics of their input data. The basic concepts of CNNs are *local receptive fields, shared weights* and *pooling*. In this context, the input layer is considered to be 2-dimensional structure typically a square.

Local receptive fields

Each neuron of the hidden layer is connected to a small region of the input layer, typically a small square, which is called the *local receptive field* of the hidden neuron. Each of these connections has a trainable weight, and the neuron has a bias. The region is slid across the input layer, passing values to the hidden layer.

Shared weights and biases

All of the hidden neurons share the same weights and biases; with the activation of the i, kth neuron begin:

$$\sigma\left(b + \sum_{l=0}^{r_x} \sum_{m=0}^{r_y} w_{l,m} a_{j+l,k+m}\right) \tag{2.17}$$

where σ represents the neural activation function, $a_{x,y}$ are the activations of the input at those coordinates and r_x, r_y are the dimensions of the receptive field.

Sharing weights and biases allows for features to be detected from input data, in conjuction with their spatial coordinates. For this reason, the mapping of the input layer to the hidden

layer is called a *feature map*, and the shared weights and biases are often called a *kernel* or *filter*. Typically, CNNs have several of these feature maps, one for each feature they are configured (or learn) to detect.

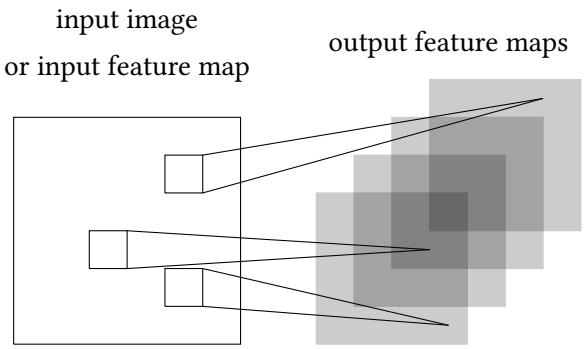


FIGURE 2.7: Illustration of a convolutional layer with multiple feature maps.

Pooling Layers Pooling layers are typically used immediately after the convolutional (hidden) layers. It reduces the dimensions of clusters of neuros into a signle neuron in the next layer. The two most common pooling procedures are max-pooling which outputs the maximum activation from the pooling region and the average pooling which outputs the average value of the activations.

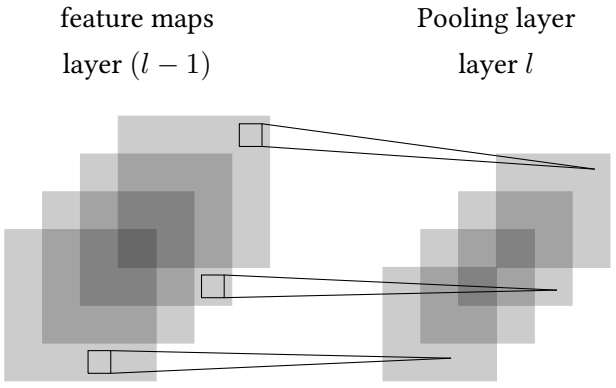


FIGURE 2.8: Illustration of the pooling layer following the feature maps

CNNs are importang for understanding Graph neural networks as they introduce the concept of spatially aware data transformations, taking into account the topology of the input data. This, albeit with some changes, will be a critical concept when studying GNNs in the next chapter.

2.3.3.2 Recurrent Neural Networks (RNNs)

So far, only networks with a static quality were discussed — information flowed from the input layer towards the output. Recurrent neural networks are types of networks were information can flow on dynamically formed temporal connections, exhibiting changing behaviour. As an example, in these types of networks, a neuron's activation might not only depend on activations of previous hidden layers, but it can also pocess an internal memory which affects, in part, its activation based on previous activations. The defining characteristic of RNNs is their change over time dynamically, while they can pocess loops feeding activations back to the neurons after some possible transformations, stored states or incorporate temporal delays. They are most useful in applications where data or processes analyzed change over item, such as in speech or natural language recognition.

Chapter 3

Graph Neural Networks (GNNs)

3.1 Introduction

Real world entities are often defined by their connections to other objects. These objects together with their connections define *graphs* as described in Section 1.1. In the past decade, research has been conducted in neural networks [31] which can operate in graph data; vertices, edges and global attributes of graphs. Graphs can express systems organized in a large number and interacting, and complex systems, which can be found in various areas of scientific study. As a defining characteristic, GNNs are deep learning methods operating on the graph domain [32].

Historically, interest in neural networks for graph structured data appeared with applications of RNNs at the end of the 20th century [33]. The revolution that CNNs brought to the scene of deep neural networks resulted in a rekindled interest for GNNs at the start of the last decade, as they have the ability to extract localized spatial features and construct expressive representations.

While the architecture of CNNs allowed for breakthroughs on data which is organized in euclidean space, such as images (2D grid) or one dimensional sequences (text or speech), it is hard to express graphs on an euclidean domain, as generally, graphs are non-Euclidean. An effort to extend structured deep learning models to non-Euclidean domains, such as in graphs or manifolds, is underway and it is called *geometric deep learning*[34].

Another research approach is focused on *graph representation learning* [35, 36] which aims to translate and represent graphs, nodes and edges as low-dimensional vectors thereby preserving network topology structure, vertex content and other graph level information. This approach is based on the idea of *representation learning* which was sparked by the work done by *Mikolov et al.* in 2013 in word embedding [37, 38], widely regarded as the first graph embedding method

based on representation learning. In this thesis, graph representation learning techniques were used.

Recently, the expressive power of GNNs has found practical use in several scientific application such as drug discovery [39, 40], molecular properties prediction [41], physics aware simulations [42], applications in (internet) network intrusion detection [43], spammer detection in cyberspace [44], anomaly detection in social networks and email networks [45], fake news detection [46], traffic forecasting [47] and many more.

GNNs take into account the features of the underlying graphs and can make informed decisions about them, either in classification or prediction tasks, matching or surpassing the capabilities of older methods such as graph kernels and random walk methods [48].

Some examples of data that is suitable for representation in a graph structure and commonly studied in literature are presented below:

- **Molecules**. Molecules constitue an excellent example of an entity that can be represented with a graph, as they consist of atoms (nodes) that are connected with bonds (edges) and they can have several molecular properties (global properties). It is a very common abstraction in literature, even before the advent of GNNs, to represent them in this structure[49].
- **Social Networks**. Social networks are networks between social actors who interact with each other. These types of networks are important when studying patterns of collective behaviour or even epidemics and other social phenomena.. In this representation, it is common to represent actors as nodes and their relationship as edges.

Tasks

The tasks that a GNN can be used to accomplish can be divided in three main categories, graph (or master) level, node level and edge level.

- **Graph-level task**. These types of tasks operate on the global (or master) node, and can be used to classify or predict some of their properties. For example, when representing a molecule as a graph, they can be used to indicate whether it is toxic, or in social networks when applying some opinion dynamics models it can indicate if a network has reached concesus.
- **Node-level task**. Node level tasks predict or classify some properties or the role of nodes in a graph. As an example, in a model used in studying a simple SIS dynamics, node level tasks would predict the state of the node.

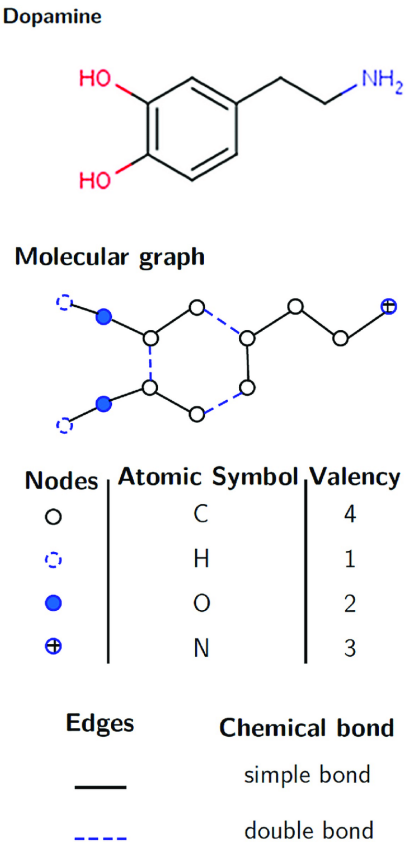


FIGURE 3.1: Dopamine molecule represented as a graph. Different types of nodes represent different atoms and different edges represent the chemical bonds present in the molecule. Image source Stefi et al. [2].

- **Edge-level task**. Edge level tasks predict or classify edge properties or even predict the existence of an edge between two nodes (*link prediction*). In the latter case, it is common to consider a graph fully connected and then prune edges to arrive to a sparse graph.

3.2 From CNNs to Graph Neural Networks

3.2.1 Challenges

Structure As discussed previously, CNNs are extremely efficient at extracting spatial features and creating effective representations from the input data. This is useful when input data is topologically static, as in the case of images, but graphs are flexible mathematical models, where the input data **lacks consistent strucure** makes creating useful representations difficult. For instance, consider the task of predicting whether a molecule is toxic; molecules may have different numbere of atoms, which may be of different types, each connected with a different number of nodes of different types (ionic, covalent and of different strength).

Node-order equivariance Images have a two-dimensional structure where each pixel is defined by its absolute position within the image, possibly even defining features based on itself and its neighborhood. In graphs, nodes have no inherent ordering and thus any possible representation should not depend on the order of the nodes — permutations on node ordering should result in representation that reflect the same outputs.

Scalability Networks can be huge, i.e. a graph of the users of social networks could number billions of nodes and connections.

3.2.2 Creating embeddings

Before moving to convolutions on graphs and other forms of neural networks, a representation of the graph in embedding space must be created, which maps individual nodes to fixed-sized real value vectors. This allows the application of modern machine learning algorithms, designed for sequences of data or grid-topology to be applied on graphs. The goal of this procedure is to encode nodes so the similarity in the embedding space approximates the similarity in original network. Typically, node embedding consists of three basic stages:

1. An encoder ENC is defined which maps nodes i.e. u and v to low-dimensional vectors z_u and z_v.

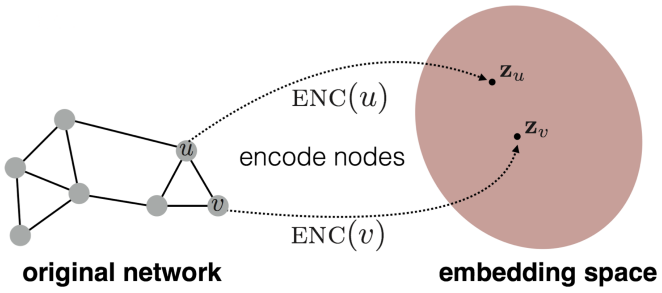


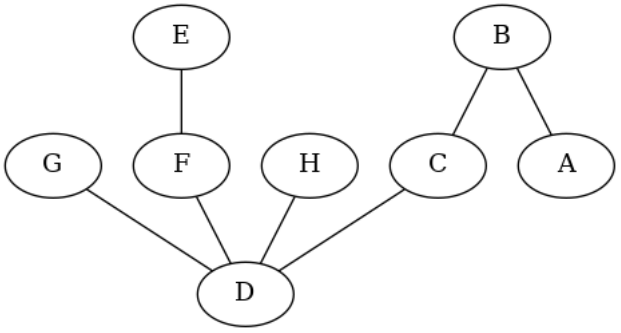
FIGURE 3.2: Embedding of nodes u and v to low dimensional vector space. Image source [3]

2. A node similarity function F_{sim} is defined which specifies how the relationships in the original network map to these in the vector space.

3. The parameters of the encoder are optimized so that the similarities in the original network are approximately the same as the dot product of the node embeddings: $F_{sim}(u, v) \approx \boldsymbol{z_u^\top z_v}$

In the following section several types of embedding methods will be showcased. The notation $h_v^{(k)}$ denotes the representation of node v after k iterations[1] of the algorithms. Nodes have individual features as part of their input, where x_v denotes the x feature of the v^{th} node.

[1]Iterations in this case can be thought as layers, which were discussed in previous sections.



A	1
B	4
C	3
D	7
E	3
F	2
G	1
H	9

FIGURE 3.3: A graph and its corresponding feature vector. The feature vector contains the features from all the nodes, in this case simply a real number.

3.2.3 Initial Implementations

In this section an introduction to the methods used for embedding will be made, at first mentioning the polynomial methods used historically before moving to modern graph neural network techniques.

Polynomials of the graph Laplacian

The graph Laplacian was discussed in Section 1.1.3 while the use of its polynomials as a means of creating embeddings was first suggested by Defferrard et al. [50] in 2016. According to this publication, it is possible to define polynomials of the form:

$$p_w(L) = w_0 I_n + w_1 L + w_2 L^2 + \cdot + w_d L^d = \sum_{i=0}^{d} w_i L^i \tag{3.1}$$

where $w = [w_0, \cdot w_d]$ is a vector of coefficients. Each w, $p_w(L)$ and L are $n \times n$ matrices.

A feature vector's x_v convolution can then be written as:

$$x' = p_w(L)x \tag{3.2}$$

For instance, considering the convolution of a node v with $w_1 = 0$ and all other coefficients equal to 0:

$$\begin{aligned} x'_v &= (Lx)_v = L_v x \\ &= \sum_{u \in G} L_{vu} x_u \\ &= \sum_{u \in G} (D_{vu} - A_{vu}) x_u \\ &= D_v x_v - \sum_{u \in N(v)} x_u \end{aligned} \tag{3.3}$$

It possible to show [51, Lemma 5.2] that the degree d of the polynomial directly influences the behaviour of the convolution as:

$$\text{dist}_G(v, u) > i \rightarrow L^i_{vu} = 0 \tag{3.4}$$

which implies that the convolution of x with a polynomial of degree d is:

$$x'_v = \sum_{i=0}^{d} w_i \sum_{\substack{u \in G \\ \text{dist}G(v,u) \leq i}} L^i_{vu} x_u \tag{3.5}$$

This means that the node v convolves with nodes u which are up to distance d; thus this creates localized filters with a degree governed completely by d. Note that a direct comparison with CNNs leads to the conclusion that the polynomials are in a sense equivalent to filters, and the weights used in previous equations are the (shared) weights of these filters.

Another variant of this polynomial method is called ChebNet [50] and replaces the filters with the Chebyshev polynomials of the first kind.

Remark 3.1. **Order equivariance of polynomial filters**

The polynomial filters of the previous section are order equivariant, a fact which is easy to prove. Considering the permutation matrix [2] P, an algorithm f is node-order equivariant if and only if for all permutations of P:

$$f(Px) = Pf(x)$$

[2] A permutation matrix is a square binary matrix which has exactly one entry of 1 in each row and column and 0 elsewhere.

Following the transformations:

$$x \to Px$$
$$L \to PLP^\intercal$$
$$L^i \to PL^iP^\intercal$$

the polynomial filter equation changes as:

$$\begin{aligned} f(Px) &= \sum_{i=0}^{d} w_i (PL^iP^\intercal)(Px) \\ &= P \sum_{i=0}^{d} w_i L^i x' \\ &= Pf(x) \end{aligned}$$

thus it is node-order equivariant.

3.2.3.1 Algorithmic Computation using polynomials

A neural network using stacked layers of any kind of polynomial filter can be created, much like a CNN. For K different polynomial filter layers, each gets a vector $\boldsymbol{w^{(k)}}$ of learnable weights. The operation starts from an initial vector of features x.

Algorithm 3: Embedding Computation for a GNN using K polynomial filter layers.

Data: A feature vector x
Result: An embedding for K^{th}
begin
  // Vector of initial features.
  $h^{(0)} \leftarrow x$
  $layers \leftarrow [1, 2, \dots K]$
  for *k in layers* **do**
    // The polynomial matrix $p^{(k)}$ is computed and multiplied with last embedding
    $p^{(k)} \leftarrow p_{w^{(k)}}(L)$
    $g^{(k)} = p^{(k)} \times h^{(k-1)}$
    // g goes through a non-linear activation σ
    $h^{(k)} \leftarrow \sigma(g^{(k)})$
  end
end

Note that this computation can be thought as "message passing" between connected nodes [52], in effect each node receiving some information from its neighbors. Some analogies to the CNNs should better explain the concept of message passing; the iterative nature of this

method, i.e. repeating 1-hop localized convolutions K-times, is in effect the receptive field of the convolution with each hop adding the information from another hop up to K hops away. Additionaly, the filter weights are shared accros all the convolutions in the same manner as in CNNs.

3.3 Modern Graph Neural Networks

3.3.1 Embeddings

Message passing forms the basis of all modern GNNs, while in this thesis a model employing an attention layer was used, called Graph Attention Networks first proposed by Velickovic et al. [53]. The attention [4] mechanism used in this paper is briefly discussed in Appendix A and in the case of graphs, it operates by computing the hidden representations of each node by attending over its neighbors, using self-attention.

As mentioned in the previous section, a function which creates the representations of nodes must be permutation invariant. As an example, it is possible to consider a weighted sum as an aggregation of a node with its first degree neighbors. A choice for a permutation invariant function is a scalar scoring function $f(n_i, n_j)$ that assigns weights based on pairs of nodes, scoring their relevance. A softmax function [3] can be applied to focus on the neighbor with the highest relevance to the node. A common scoring function is the inner product and it is also usual to transform the nodes before scoring into key and query vectors to increase the expressivity of the mechanism.

[3] A softmax (also known as softargmax or multi-class logistic) function takes a vector $\boldsymbol{z}$ of real values as input and returns a a vector K of real values that sum to 1. It is a generalization of the logistic function to multiple dimensions and its formula is $\sigma(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$ where $K > 1$.

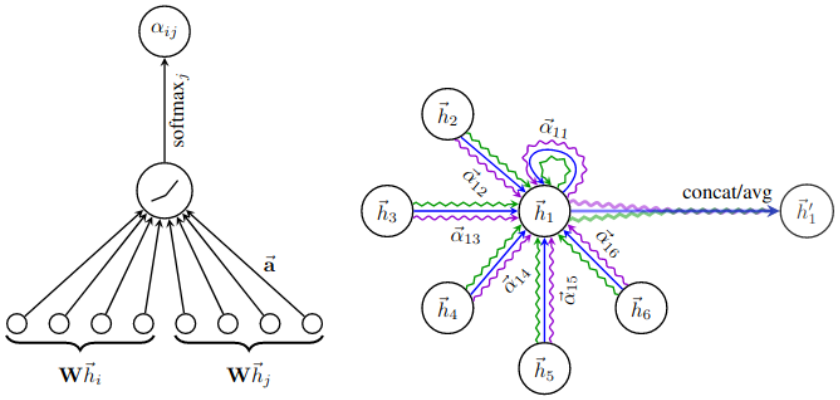


FIGURE 3.4: The attention mechanism used by Velickovic et al. [53]. On the **left** is a demonstration of the attention mechanism $f(\boldsymbol{W}\vec{h}_i, \boldsymbol{W}\vec{h}_j)$ used by the model, with a parametrization of $\vec{\boldsymbol{a}}$ of the weight vector. On the **right** a demonstration of the multi-head module in action, acting on node $\vec{h}_1$ with $K = 3$ with independent attention computations visible. The representation $\vec{h}'_1$ is obtained through the aggregation of features after proper concatenation of the heads. Image source [53].

The attention mechanism used in graph attention networks is extensively discussed in Appendix A.2. In a more compact form, it is possible to express the embedding produced by a GAT in a compact form as:

Starting from $h_v^{(0)} = x_v$ for the initial embedding for node v:

$$h_v^{(k)} = f^{(k)}\Big(\boldsymbol{W}^{(k)}\Big[\sum_{u \in N_u} \alpha_{vu}^{(k-1)} h_u^{(k-1)} + \alpha_{vv}^{(k-1)} h_v^{(k-1)}\Big]\Big), v \in V. \tag{3.6}$$

The attention weights $\alpha^{(k)}$ derivation can be found in the appendix. Obviously, predictions can be made using the final embedding $\hat{y}_v = g(h_v^{(K)})$ where g is generally another neural network learned together with the GAT model. The parameters $f^{(k)}$, $\boldsymbol{W^{(k)}}$and$\boldsymbol{A^{(k)}}$ are shared across all nodes and are potentially learnable [54].

3.3.2 Learning

In the previous section, embeddings were created for nodes, using the GAT spatial computation described. These embedding layer can be attached in an end-to-end fashion in the same manner as a typical NN, to train the network and produce usable outputs. This can be achieved by defining a suitable loss function for each learning task:

Node classification: In a similar manner as in other classification tasks, node classification can take advantage of the categorical cross-entropy loss function:

$$L(y_v, \hat{y_v}) = -\sum_c y_{vc} log \hat{y}_{vc} \tag{3.7}$$

where $\hat{y}_{vc}$ is the predicted probability of node v belogning to class c.

Graph Classification: The aggregate of the node embeddings can be used to create a vector representation of the entire graph which can be used for tasks even beyond classification. It can be passed to another neural network NN_G as:

$$h_G = NN_G(AGG_{v \in G}(h_v))$$

Edge Prediction: GNNs can be used to predict the presence or absence of edges between nodes, by sampling pairs of adjacent and non-adjacent nodes. A loss function closely resembling the logistic regression can be used in this case:

while there are many publications on powerful pooling techniques [55–57].

Chapter 4

Experiments and Results

4.1 Background

In this thesis, GNNs were used to predict the evolution of complex dynamics on networks, specifically contagion dynamics and majority rule dynamics. To this end, the publication "Deep learning of contagion dynamics on complex networks" by Murphy et al. [58] served as a basis both as an insipiration and as a coding paradigm. In the approach used, dynamics of network are learned through deep learning techniques, very few assumptions are made about them. It is also demonstrated, that GNNs which are usually used for structure learning, can also be used to model complex dynamics on simple and complex networks, and that they can provide predictions for previously unseen network structures allowing for learned dynamics to be applied beyond the training data.

TODO: MAYBE MORE HERE

4.2 Fundamental Ideas of this approach

In this approach, it is assumed that an unknown dynamical process denoted as M which takes places on a known network structure $G = (V, E; \mathbf{\Phi}, \mathbf{\Omega})$ where V is the node vector, E is the node set and $\mathbf{\Phi}_i$ and $\mathbf{\Omega}_{ij}$ are node and edge attributes respectively, such as node attributes or edge weights. The dynamics acting on the network generate a time series D, defined as pairs of consecutive spanshots $D = (\boldsymbol{X}, \boldsymbol{Y})$ with $\boldsymbol{X} = (X_1, \cdots, X_T)$ the state of the nodes at time t and $\boldsymbol{Y} = Y_1, \cdots, Y_T$ the outcome of the dynamics with $Y_t = M(X_t, G)$ with $X_t \in S^{|V|}$, $t, Y_t \in R^{|V|}$ where S is the set of possible node states and R is the set possible node outcomes. The elements $x_i(t) = (X_t)_i$ and $y_i(t) = (Y_t)_i$ correspond to the state of node v_i at time t and its outcome after transitioning respectively, thus $S = R$ and it is possible to

write $y_i(t) = x_i(t + \Delta t)$ where Δt is the length of the time steps. If S is a discrete set then $y_i(t)$ is a transition probability vector and its value is sampled from a set of possible values. In fact, $y_i(t)_m$ is the probability that node v_i transitions to state $m \in S$ if it was previously in $x_i(t)$ — i.e. when studying an SIS dynamics model that could be $R = [0, 1]^{|S|}$. In stochastic dynamics' modelling, the transition probabilities are not directly accessible, so the observed outcome state is used to produce a definition for the observed outcome $\tilde{y}_i(t)$:

$$\tilde{y}_i(t)_m = \delta(x_i(t + \Delta t), m), \forall m \in S \tag{4.1}$$

where δ is the Kronecker delta function. If the stochastic dynamics M used in this case is assumed to act on X_t locally and identically at all times, it is possible to compute the value of y_i with a time independent function identical for all nodes

$$y_i = f(x_i, \Phi_i, x_{N_i}, \Phi_{N_i}, \Omega_{iN_i}) \tag{4.2}$$

where N_i is the set of neighbors of node v_i. The result of this process is a notion of locality of the underlying dynamics which also is time-invariant and node-invariant (and edge-invariant) as required and discussed in previous sections.

4.3 Description of goals

The goal of this approach is to build a model $\hat{M}$ with learnable parameters $\mathbf{\Theta}$ which after being trained over a set of the observed dataset D mimics M given G so that:

$$\hat{M}(X'_t, G'; \mathbf{\Theta}) \approx M(X'_t, G') \tag{4.3}$$

The design of the GNN (discussed in more detail later) is based around the notion of locality, imposed by a modified attention mechanism, which allows for the arthitecture to be inductive; training on a wide range of local structures makes it possible to use the model on any other network within that local range. Therefore, the topology of the network is strongly associated with the quality of the trained models. The node outcomes computed by the GNN can be written in a similar manner as Equation 4.2:

$$\hat{y}_i = \hat{f}(x_i, \Phi_i, x_{N_i}, \Phi_{N_i}, \Omega_{iN_i}; \mathbf{\Theta}) \tag{4.4}$$

Loss Function: The loss function incorporates an arithmetic mean, assuming all inputs are equally important and uniformly distributed. This is critical as in practice that does not hold

with finite number of inputs D which is not typically uniform and some parametrization is needed. The loss function used in this model is:

$$L(\mathbf{\Theta}) = \sum_{t \in T'} \sum_{v_i \in V'(t)} \frac{w_i(t)}{Z'} L(y_i(t), \hat{y}_i(t)) \tag{4.5}$$

where $w_i(t)$ is the weight assigned to node v_i at time t, $Z' = \sum_{t \in T'} \sum_{v_i \in V'(t)} w_i(t)$ a normalization factor and $L(y_i(t), \hat{y}_i(t))$ the local losses of each node.

4.4 Architecture of the GNN

4.4.1 Structure

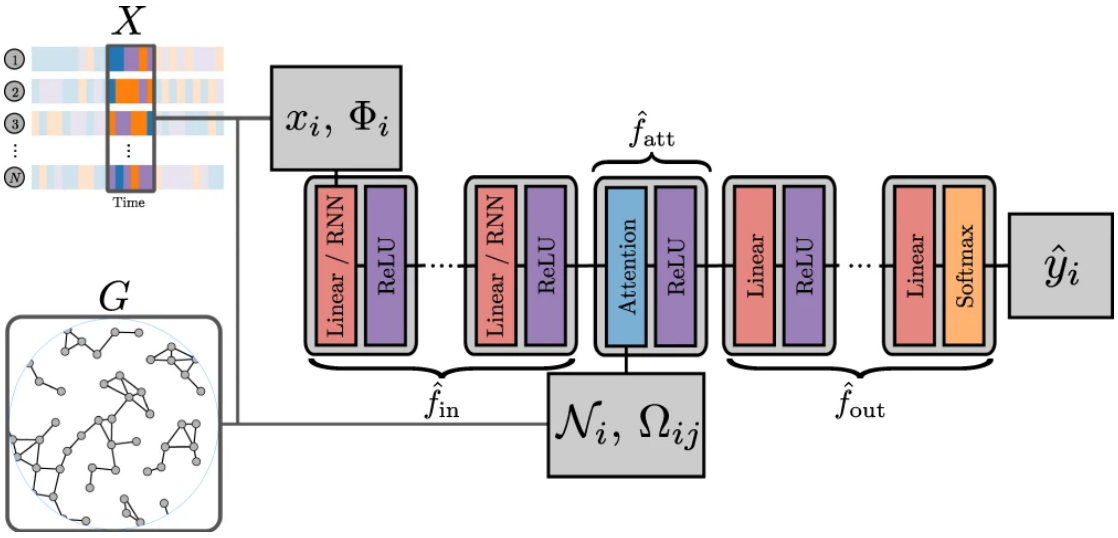


FIGURE 4.1: Schematic showing the layers in the GAT GNN used. Red blocks indicate trainable affine transformations (automorphisms). Purple blocks indicate activation functions. The attention module is in blue and the orange block in the end is the last activation which translates the transformations to the expected format. Image source [58].

The architecture of the network used can be seen in Figure 4.1. The state x_i of each node is transformed through a shared MLP $\hat{f}_{in} : S \to \mathbb{R}^d$ where d is the resulting number of node features:

$$\xi_i = \hat{f}_{in}(x_i) \tag{4.6}$$

When available, the node attributes Φ_i are concatenated to x_i so $\hat{f}_{in} : S \times \mathbb{R}^Q \to \mathbb{R}^d$ and the ξ_i is now a feature vector with representing the state and attributes of node v_i. The features of the first neighbors are then aggregated using an attention mechanism discussed in the next section:

$$v_i = \hat{f}_{att}(\xi_i, \xi_{N_i}) \tag{4.7}$$

If available, edge attributes Ω_{ij} are also included in the attention mechanism. In a similar manner as before, these attributes are first transformed into edge features through a $\psi_{ij} = \hat{f}_{edge}(\Omega_{ij})$ where $\hat{f}_{edge} : \mathbb{R}^P \rightarrow \mathbb{R}^{d_edge}$ is also an MLP. Finally the outcome $\hat{y}_i$ of each node is computed with another MLP $\hat{f}_{out} : \mathbb{R}^d \rightarrow R$:

$$\hat{y}_i = \hat{f}_{out}(v_i) \tag{4.8}$$

4.4.2 Attention Mechanism

The attention mechanism used is a modified version of the one described by Velickovic et al. [53]. It consists of three trainable functions A, B and$C$ that combine the feature vectors ξ_i, xi_j and ψ_{ij} of pair of connected nodes v_i and v_j. The attention coefficient is:

$$\alpha_{ij} = \sigma\big[A(\xi_i) + B(\xi_j) + C(\psi_{ij})\big] \tag{4.9}$$

where σ is the logistic function which allows for an output range in $(0, 1)$, 1 meaning maximal influence of v_j on v_i and 0 non-existent influence. All of the transformations $A, B,$ and C are affine and have trainable weights and biases.

The aggregation formula is:

$$v_i = \hat{f}_{att}(\xi_i, \xi_{N_i}) = \xi_i + \sum_{v_j \in N_i} \alpha_{ij}\xi_j \tag{4.10}$$

and this v_i contais information about itself and its neighbors through a multi-head attention mechanism. In contrast with the proposed aggregation mechanism from the GAT paper, the one used in Equation 4.10 uses a general weighted sum which allows for the architecture to express dynamic process more accurately. The one used by the GAT paper and other architectures is better when creating models for structure learning.

4.4.3 Loss Function

The cross entropy loss function was used in all experiments:

$$L(y_i, \hat{y}_i) = -\sum_m y_{i,m} log \hat{y}_{i,m} \tag{4.11}$$

with y_i, m the m^{th} element of the outcome vector for node v_i a transition probability vector as discussed before.

4.5 Experiments TODO

4.5.1 SIS

4.5.2 Modular

4.5.3 Majority rule

Appendix A

Transformers and self-attention

A.1 Attention and self-attention

In neural networks, *attention*[4] is a mechanism that takes into account several of the inputs of the network and attributes importance to them by means of different weights and biases. Its function mimicks that of cognitive attention of biological brains.

The main objective of attention in neural networks is to learn to recognize which parts of the input are relevant and which are not. For instance one of its first uses was in speech recognition and natural language understanding [59], where large data sequences are encountered.

Another technique described in the paper by Vaswani et al. [4] is **self-attention** where is a mechanism which relates different positions of the same input sequence to create a representation of this sequence.

Multi-head attention is a module (collection of layers) used in NNs using attention which runs attentions mechanisms several times in parallel. The outputs are then concatenated and transformed into the expected output dimension. This allows for different heads attending to different parts of the sequence as required. It is possible to describe a multi-head mechanism as

$$\begin{aligned} \text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) &= [\text{head}_1, \cdots, \text{head}_h]\boldsymbol{W_0} \\ \text{where head}_i &= \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_{\boldsymbol{I}}^{\boldsymbol{Q}}, \boldsymbol{K}\boldsymbol{W}_{\boldsymbol{i}}^{\boldsymbol{K}}, \boldsymbol{V}\boldsymbol{W}_{\boldsymbol{i}}^{\boldsymbol{V}}) \end{aligned}$$

where all $\boldsymbol{W}$ are learnable weight matrices[4].

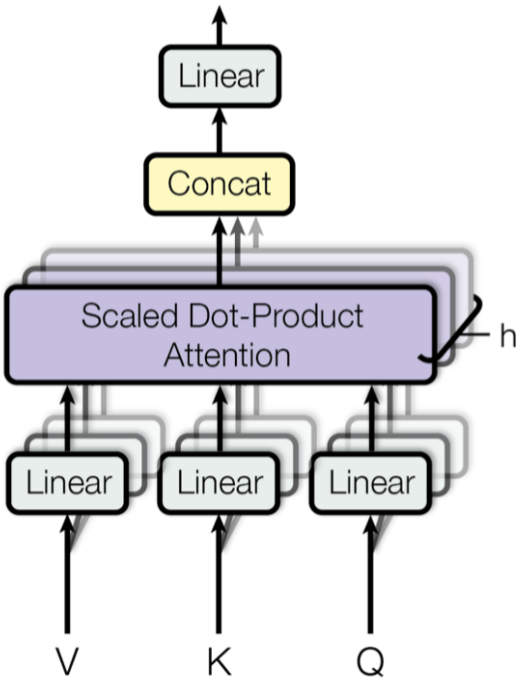


FIGURE A.1: Multi-head attention module as described above. Image source [4]

A.2 Attention in Graph Attention Networks

In their paper Velickovic et al. [53] describe a process of creating a *graph attentional layer*, a process which is summarized in this section.

The layer takes as input a set of node features, $\boldsymbol{h} = \{\vec{h}_1, \vec{h}_2, \cdots, \vec{h}_N\}, \vec{h}_i \in \mathbb{R}^{\mathbb{F}}$ where N is the number of nodes and F are the features per node. The output of the layer is a new set of node features (possibly of differrent cardinality F'), $\boldsymbol{h'} = \{\vec{h'}_1, \vec{h'}_2, \cdots, \vec{h'}_N\}, \vec{h}'_i \in \mathbb{R}^{\mathbb{F}'}$.

A learnalbe linear transformation is then applied to achieve the required expressive power to transform the input features to higher-level dimensional features. This is achieved as a shared linear transformation, parametrized by a weight matrix $\boldsymbol{W} \in \mathbb{R}^{\mathbb{F}' \times \mathbb{F}}$ and applied to each node. A shared self-attention α is then applied to all nodes, with $\alpha : \mathbb{R}^{\mathbb{F}'} \times \mathbb{R}^{\mathbb{F}'} \to \mathbb{R}$ which computes the attentions coefficients:

$$e_{ij} = \alpha(\boldsymbol{W}\vec{h}_i, \vec{h}_j) \tag{A.1}$$

which indicates the importance of node j's to node features of node i. Injecting the graph structure in the mechanism (*masked attention*) results in only computing the e_{ij} for $j \in N_i$ where N_i is the neighborhood of node i. A softmax function is then applied to make the coefficients more easily comparable.

In this implementation, the network is parametrized by a weight vector $\vec{\boldsymbol{a}}^{\intercal} \in \mathbb{R}^{2F'}$ and a LeakyRELU nonlinearity is applied. The fully expanded form of this formula is:

$$\alpha_{ij} = \frac{exp(\text{LeakyRELU}(\vec{\boldsymbol{a}}^{\intercal}[\boldsymbol{W}\vec{h}_i \parallel \boldsymbol{W}\vec{h}_j]))}{\sum_{k \in N_i} exp(\text{LeakyRELU}(\vec{\boldsymbol{a}}^{\intercal}[\boldsymbol{W}\vec{h}_i \parallel \boldsymbol{W}\vec{h}_k]))} \tag{A.2}$$

where $\parallel$ is the concatenation operation.

Finally, after obtaining the normalized attention α_{ij} a linear combination of the features is obtained and served as the final output features of each node, after applying a nonlinearity:

$$\vec{h}'_i = \sigma\Big(\sum_{j \in N_i} \alpha_{ij} \boldsymbol{W}\vec{h}_j\Big) \tag{A.3}$$

A multi-head attention approach is used for stabilizing the process of self-attention. For K independent attention mechanisms executing the process described in Equation A.3, the output feature representation is

$$\vec{h}'_i = \Big\Vert_{k=1}^{K} \sigma\Big(\sum_{j \in N_i} \alpha_{ij}^{k} \boldsymbol{W^k}\vec{h}_j\Big) \tag{A.4}$$

Using the multi-head approach, the output feature vector $\boldsymbol{h'}$ consits of KF' features for each node. If the multi-head attention is performed in the final prediction layer of the network, averaging is usually applied and nonlinearity is not applied until then.

Bibliography

[1] Kristy Carpenter, David Cohen, Juliet Jarrell, and Xudong Huang. Deep learning and virtual drug screening. *Future Medicinal Chemistry*, 10, 10 2018. doi: 10.4155/fmc-2018-0314.

[2] Nouleho Stefi, Dominique Barth, Olivier David, Franck Quessette, Marc-Antoine Weisser, and Dimitri Watel. Improving graphs of cycles approach to structural similarity of molecules. *PLOS ONE*, 14:e0226680, 12 2019. doi: 10.1371/journal.pone.0226680.

[3] Node representation learning. URL `https://snap-stanford.github.io/cs224w-notes/machine-learning-with-networks/node-representation-learning`.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL `https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf`.

[5] U.S.R. Murty Adrian Bondy. *Graph theory*. Graduate texts in mathematics 244. Springer, 3rd corrected printing. edition, 2008. ISBN 1846289696; 9781846289699.

[6] Bela Bollobas and Endre Szemerédi. Girth of sparse graphs. *Journal of Graph Theory*, 39: 194 – 200, 01 2002. doi: 10.1002/jgt.10023.

[7] Ping Zhang Gary Chartrand. *A First Course in Graph Theory*. Dover Books on Mathematics. Dover Publications, 2012. ISBN 0486483681; 9780486483689.

[8] A. Ashikhmin and A. Barg. *Algebraic Coding Theory and Information Theory: DIMACS Workshop, Algebraic Coding Theory and Information Theory, December 15-18, 2003, Rutgers University, Piscataway, New Jersey*. DIMACS series in discrete mathematics and theoretical computer science. American Mathematical Soc. ISBN 9780821871102. URL `https://books.google.gr/books?id=wp7XsCAm9EC`.

[9] Mark Newman. *Networks*. Oxford University Press, 2 edition, 2018. ISBN 0198805098; 9780198805090.

[10] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. 06 2016.

[11] Jürgen Jost. *Dynamical Networks*, pages 35–62. Springer London, London, 2007. ISBN 978-1-84628-780-0. doi: 10.1007/978-1-84628-780-0_3. URL https://doi.org/10.1007/978-1-84628-780-03.

[12] Réka Albert and Albert-László Barabási. Statistical mechanics of complex networks. *Reviews of modern physics*, 74(1):47, 2002.

[13] Paul Erdős, Alfréd Rényi, et al. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[14] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, 1943. doi: 10.1007/bf02478259.

[15] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65 6:386–408, 1958.

[16] M. Minsky and S. Papert. *Perceptrons; an Introduction to Computational Geometry*. MIT Press, 1969. ISBN 9780262630221. URL https://books.google.gr/books?id=Ow1OAQAAIAAJ.

[17] P.J. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences*. Harvard University, 1975. URL https://books.google.gr/books?id=z81XmgEACAAJ.

[18] J. Weng, N. Ahuja, and T.S. Huang. Cresceptron: a self-organizing neural network which grows adaptively. In *[Proceedings 1992] IJCNN International Joint Conference on Neural Networks*, volume 1, pages 576–581 vol.1, 1992. doi: 10.1109/IJCNN.1992.287150.

[19] Kevin N. Gurney. *An introduction to neural networks*. 1997.

[20] Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. ISSN 0893-6080. doi: https://doi.org/10.1016/j.neunet.2014.09.003. URL https://www.sciencedirect.com/science/article/pii/S0893608014002135.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[22] A. Cauchy. Méthode générale pour la résolution des systèmes d'équations simultanées. *C. R. Acad. Sci*, 25:536–538, 1847.

[23] Xiao Qi Yang Liqun Qi, Kok Lay Teo. *Optimization and control with applications*. Applied Optimization. Springer, 1 edition, 2005. ISBN 9780387242545; 0387242546.

[24] Milton Abramowitz and Irene Stegun. *Abramowitz and Stegun*. Bracewell 200, 1972.

[25] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL http://neuralnetworksanddeeplearning.com.

[26] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of COMPSTAT'2010*, pages 177–186, Heidelberg, 2010. Physica-Verlag HD. ISBN 978-3-7908-2604-3.

[27] H. Robbins and D. Siegmund. A convergence theorem for non negative almost supermartingales and some applications**research supported by nih grant 5-r01-gm-16895-03 and onr grant n00014-67-a-0108-0018. In Jagdish S. Rustagi, editor, *Optimizing Methods in Statistics*, pages 233–257. Academic Press, 1971. ISBN 978-0-12-604550-5. doi: https://doi.org/10.1016/B978-0-12-604550-5.50015-8. URL https://www.sciencedirect.com/science/article/pii/B9780126045505500158.

[28] Paul John Werbos. *The Roots of Backpropagation: From Ordered Derivatives to Neural Networks and Political Forecasting*. Wiley-Interscience, USA, 1994. ISBN 0471598976.

[29] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323:533–536, 1986.

[30] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. doi: 10.1109/5.726791.

[31] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1): 61–80, 2009. doi: 10.1109/TNN.2008.2005605.

[32] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81, 2020. ISSN 2666-6510. doi: https://doi.org/10.1016/j.aiopen.2021.01.001. URL https://www.sciencedirect.com/science/article/pii/S2666651021000012.

[33] A. Sperduti and A. Starita. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997. doi: 10.1109/72.572108.

[34] Michael M. Bronstein, Joan Bruna, Yann LeCun, Arthur Szlam, and Pierre Vandergheynst. Geometric deep learning: Going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4):18–42, 2017. doi: 10.1109/MSP.2017.2693418.

[35] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[36] Daokun Zhang, Jie Yin, Xingquan Zhu, and Chengqi Zhang. Network representation learning: A survey, 2017.

[37] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, 26, 2013.

[39] Jiacheng Xiong, Zhaoping Xiong, Kaixian Chen, Hualiang Jiang, and Mingyue Zheng. Graph neural networks for automated de novo drug design. *Drug Discovery Today*, 26(6):1382–1393, 2021. ISSN 1359-6446. doi: https://doi.org/10.1016/j.drudis.2021.02.011. URL https://www.sciencedirect.com/science/article/pii/S1359644621000787.

[40] Pietro Bongini, Monica Bianchini, and Franco Scarselli. Molecular generative graph neural networks for drug discovery. *Neurocomputing*, 450:242–252, 2021. ISSN 0925-2312. doi: https://doi.org/10.1016/j.neucom.2021.04.039. URL https://www.sciencedirect.com/science/article/pii/S0925231221005737.

[41] Oliver Wieder, Stefan Kohlbacher, Mélaine Kuenemann, Arthur Garon, Pierre Ducrot, Thomas Seidel, and Thierry Langer. A compact review of molecular property prediction with graph neural networks. *Drug Discovery Today: Technologies*, 37:1–12, 2020. ISSN 1740-6749. doi: https://doi.org/10.1016/j.ddtec.2020.11.009. URL https://www.sciencedirect.com/science/article/pii/S1740674920300305.

[42] Alvaro Sanchez-Gonzalez, Jonathan Godwin, Tobias Pfaff, Rex Ying, Jure Leskovec, and Peter Battaglia. Learning to simulate complex physics with graph networks. In Hal Daumé III and Aarti Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 8459–8468. PMLR, 13–18 Jul 2020. URL https://proceedings.mlr.press/v119/sanchez-gonzalez20a.html.

[43] Wai Weng Lo, Siamak Layeghy, Mohanad Sarhan, Marcus Gallagher, and Marius Portmann. E-GraphSAGE: A Graph Neural Network based Intrusion Detection System for IoT. *arXiv e-prints*, art. arXiv:2103.16329, March 2021.

[44] Zhiwei Guo, Lianggui Tang, Tan Guo, Keping Yu, Mamoun Alazab, and Andrii Shalaginov. Deep graph neural network-based spammer detection under the perspective

of heterogeneous cyberspace. *Future Generation Computer Systems*, 117:205–218, 2021. ISSN 0167-739X. doi: https://doi.org/10.1016/j.future.2020.11.028. URL https://www.sciencedirect.com/science/article/pii/S0167739X20330612.

[45] Anshika Chaudhary, Himangi Mittal, and Anuja Arora. Anomaly detection using graph neural networks. In *2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*, pages 346–350, 2019. doi: 10.1109/COMITCon.2019.8862186.

[46] Federico Monti, Fabrizio Frasca, Davide Eynard, Damon Mannion, and Michael M. Bronstein. Fake news detection on social media using geometric deep learning. 2019.

[47] Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey, 2021.

[48] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[49] David K Duvenaud, Dougal Maclaurin, Jorge Iparraguirre, Rafael Bombarell, Timothy Hirzel, Alan Aspuru-Guzik, and Ryan P Adams. Convolutional networks on graphs for learning molecular fingerprints. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015. URL https://proceedings.neurips.cc/paper/2015/file/f9be311e65d81a9ad8150a60844bb94c-Paper.pdf.

[50] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 29, 2016.

[51] David K. Hammond, Pierre Vandergheynst, and Rémi Gribonval. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis*, 30(2):129–150, 2011. ISSN 1063-5203. doi: https://doi.org/10.1016/j.acha.2010.04.005. URL https://www.sciencedirect.com/science/article/pii/S1063520310000552.

[52] Justin Gilmer, Samuel S. Schoenholz, Patrick F. Riley, Oriol Vinyals, and George E. Dahl. Neural message passing for quantum chemistry. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 1263–1272. PMLR, 06–11 Aug 2017. URL https://proceedings.mlr.press/v70/gilmer17a.html.

[53] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *stat*, 1050:20, 2017.

[54] Ameya Daigavane, Balaraman Ravindran, and Gaurav Aggarwal. Understanding convolutions on graphs. *Distill*, 2021. doi: 10.23915/distill.00032. https://distill.pub/2021/understanding-gnns.

[55] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *Proceedings of the AAAI conference on artificial intelligence*, volume 32, 2018.

[56] Zhitao Ying, Jiaxuan You, Christopher Morris, Xiang Ren, Will Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling. *Advances in neural information processing systems*, 31, 2018.

[57] Junhyun Lee, Inyeop Lee, and Jaewoo Kang. Self-attention graph pooling. In *International conference on machine learning*, pages 3734–3743. PMLR, 2019.

[58] Charles Murphy, Edward Laurence, and Antoine Allard. Deep learning of contagion dynamics on complex networks. *Nature Communications*, 12(1):1–11, 2021.

[59] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.