RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

Département d'Automatique
Laboratoire de Commande des Processus

Mémoire de projet de fin d'études pour l'obtention du diplôme d'ingénieur d'état en Automatique

Graph Neural Networks: A Comparative Study

Réalisé par :

Mohamed Mehdi ATAMNA
Ibrahim LAICHE

Présenté et soutenu publiquement le 07/07/2020

Composition du Jury :

Président	M. Messaoud CHAKIR	MCB	École Nationale Polytechnique
Promoteur	M. Rachid ILLOUL	MCA	École Nationale Polytechnique
Co-promotrice	Mme Asma ATAMNA	Postdoc	Télécom Paris
Examinateur	M. Mohamed TADJINE	Professeur	École Nationale Polytechnique

ENP 2020

RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE
MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

ÉCOLE NATIONALE POLYTECHNIQUE

Département d'Automatique
Laboratoire de Commande des Processus

Mémoire de projet de fin d'études pour l'obtention du diplôme d'ingénieur d'état en Automatique

Graph Neural Networks: A Comparative Study

Réalisé par :

Mohamed Mehdi ATAMNA
Ibrahim LAICHE

Présenté et soutenu publiquement le 07/07/2020

Composition du Jury :

Président	M. Messaoud CHAKIR	MCB	École Nationale Polytechnique
Promoteur	M. Rachid ILLOUL	MCA	École Nationale Polytechnique
Co-promotrice	Mme Asma ATAMNA	Postdoc	Télécom Paris
Examinateur	M. Mohamed TADJINE	Professeur	École Nationale Polytechnique

ENP 2020

Dedication

This work is dedicated to my wonderful parents, my two sisters Asma and Lina and my close friends, all of whom have been an exceptional source of support and gave me strength when I most needed it.

Mohamed Mehdi ATAMNA

I dedicate this work to my beloved parents, for their endless love, support, and encouragement.

To my brother Abdessamed, my two sisters Lamia and Walaa, all my relatives, my friends, and all those who have helped me throughout these five years.

Ibrahim LAICHE

Acknowledgments

We would like to express our sincere gratitude to our two advisors, **Dr Asma Atamna** and **Mr Rachid Illoul** for their continuous support and immense knowledge.

We would also like to thank the professors of the **Control Engineering** departement at École Nationale Polytechnique for supporting and assisting us along our journey.

Our sincere thanks also go to the thesis committee for honoring us by evaluating our work.

ملخص: الرسوم البيانية هي بنية بيانات قوية تستخدم لنمذجة الأجسام وتفاعلاتها .نظرًا لقدرتها على التقاط معلومات غنية عن الكيانات المتفاعلة ، يتم استخدامها لنمذجة مجموعة واسعة من البيانات .على سبيل المثال ، في الروبوتات ، يمكن تمثيل أجسام الروبوت المفصلية بعقد رسم بياني بينما يمكن تمثيل المفاصل التي تربط الأجسام معًا بحواف الرسم البياني .في الآونة الأخيرة ، ظهرت البنى العصبية القادرة على معالجة بيانات إدخال الرسم البياني ، والتي تسمى الشبكات العصبية البيانية (GNNs)، مع نتائج واعدة في العديد من مهام التعلم تحت الإشراف مثل تصنيف الرسم البياني .وقد تم العمل أيضًا على استخدام شبكات GNN هذه في العديد من التطبيقات ذات الصلة بالتحكم مثل الاستدلال ، التعرف على النظم ، و التحكم الاستشرافي وحتى للمهام المرئية مثل التعرف على الحركات البشرية . في هذا العمل ، وبعد مراجعة شاملة للأبحاث الحديثة حول GNNs بما في ذلك التطبيقات أحدثت طفرة في التحكم ، قمنا بدمج بنيتي GNNمرجعيتين مع آلية إهتمام قوية ، واقتراح بنيتين جديدتين والتحقق من فعاليتها على أربع مجموعات مرجعية لتصنيف الرسوم البيانية باستخدام منهجية صارمة وأدوات البرمجيات المتطورة .تحقق البنيات المقترحة لدينا مكاسب رائعة في الأداء تصل إلى 14 ٪ فوق الأداء المرجعي في مجموعات بيانات معينة ، مما يفتح آفاقًا مثيرة للاهتمام لتطبيقات مستقبلية ، خاصة في التعرف على الحركات البشرية ومشكلات تقديروضع الجسم.

الكلمات المفتاحية: التعلم الآلي ، الشبكة العصبية البيانية ، آلية الإهتمام ، التعلم تحت الإشراف ، تصنيف الرسوم البيانية.

Résumé : « Graph Neural Networks : Une Étude Comparative »
Les graphes représentent une puissante structure de données qui est utilisée pour modéliser les objets et leurs interactions. Grâce à leur capacité à capturer de riches informations sur les entités interagissantes, les graphes sont utilisés pour modéliser une variété de données. Par exemple, en robotique, les différents corps constituant un robot articulé peuvent être représentés par les nœuds d'un graphe alors que les articulations les reliant entre eux peuvent être représentés par les arêtes d'un graphe. Récemment, des architectures de réseaux de neurones conçus pour traiter des données structurées en graphes, appelés *Graph Neural Networks* (GNNs), sont apparues et ont obtenu des résultats prometteurs sur plusieurs tâches d'apprentissage supervisé telles que la classification de graphes. D'importants travaux ont également été entrepris afin d'appliquer ces architectures à des problèmes pertinents à l'automatique tels que l'observation, l'identification des systèmes, la commande prédictive et mêmes des tâches visuelles comme la reconnaissance d'action humaine. À travers ce travail, après un passage en revue approfondi de l'état de l'art en matière d'architectures GNN, applications en automatique incluses, nous combinons deux architectures parmi les plus avancées avec un puissant mécanisme d'attention pour proposer deux architectures innovantes que nous validons sur quatre ensembles de données pour la classification de graphes très utilisés en recherche, le tout en suivant une méthodologie très rigoureuse et en utilisant des outils logiciels de pointe. Nos architectures présentent un gain en performance allant jusqu'à 14% par rapport aux architectures de référence sur certains ensembles de données, ouvrant ainsi la voie à des perspectives intéressantes pour des applications futures, notamment sur des problèmes d'estimation de pose et de reconnaissance d'action humaine.

Mots-clés : apprentissage automatique, graph neural network, mécanisme d'attention, apprentissage supervisé, classification de graphes.

Abstract: Graphs are a powerful data structure that is used to model objects and their interactions. Owing to their ability to capture rich information about interacting entities, they are used to model a wide range of data. For example, in robotics, an articulated robot's bodies can be represented with a graph's nodes while the joints linking the bodies together can be represented with a graph's edges. Recently, neural architectures that are able to process graph input data, called *Graph Neural Networks* (GNNs), have emerged with promising results on

many supervised learning tasks such as graph classification. Work has also been done to use these GNNs in many control-related applications such as for inference, system identification, model-predictive control and even for visual tasks like human action recognition. In this work, after an extensive review of the state-of-the-art literature on GNNs, including breakthrough applications in control, we combine two reference GNN architectures with a powerful attention mechanism, proposing two novel architectures and validating them on four benchmark graph classification datasets using rigorous methodology and cutting-edge software tools. Our proposed architectures achieve impressive gains in performance of up to 14% over baselines on certain datasets, opening up interesting perspectives for future work, especially in human action recognition and pose estimation problems.

Keywords: machine learning, graph neural network, attention mechanism, supervised learning, graph classification.

Contents

List of Figures

List of Tables

List of Symbols

$G = (V, E)$	Graph G
V	The set of vertices (nodes)
v_i	Node (vertex) i
E	The set of edges
e_i	Edge i
A	The adjacency matrix
n	The number of vertices (nodes)
$\|V\|$	The cardinality of set V
X	Node feature matrix
d	Number of features of each node
X_v	Node feature vector for node v
I_n	Identity matrix of size n
$\mathcal{N}_v$	The neighborhood of v
$\mathcal{N}_v^*$	The neighborhood of v and the node itself
$deg(v)$	The degree of node v
D	The degree matrix
h_i^t	Node attributes of node v_i at iteration t
$\mathbf{x}$	Inputs

y	Labels
θ	Parameter vector
f_θ	A model parameterized by θ
L_θ	Loss function
ℓ	The loss for a single example
m_{train}	Number of training examples G
$\hat{y}^i$	Predicted class label for example i
y^i	Correct class label for example i
$\mathbf{w}$	Weight vector
$\mathbf{W}$	Weight matrix
σ	Nonlinear activation function
η	The learning rate
$\mathcal{G}$	The set of graphs (graph space)
$\mathcal{Y}$	The set of graph labels (label space)
h_G	Graph embedding (vector representation)
h_i	Node embedding of node v_i
$h_v^{(k)}$	The feature vector of node v at the k^{th} layer
$\hat{A}$	The normalized adjacency matrix
$W^{(k)}$	The k^{th} layer's weight matrix
$\alpha_{i,j}$	Attention coefficients
A_α	The weighted adjacency matrix
L	The number of graph convolutional layers in GINA
γ	Learning rate's decay rate.

List of Abbreviations

GNN	Graph Neural Network
GCN	Graph Convolutional Network
GIN	Graph Isomorphism Network
GAT	Graph Attention Network
GCNA	Graph Convolutional Network with Attention
GINA	Graph Isomorphism Network with Attention
WL	Weisfeiler-Lehman
NP	Nondeterministic polynomial time
ML	Machine Learning
SVM	Support Vector Machine
MSE	Mean Squared Error
ANN	Artificial Neural Network
MLP	Multi-Layer Perceptron
ReLU	Rectified Linear Unit
SGD	Stochastic Gradient Descent
CNN	Convolutional Neural Network

RNN	Recurrent Neural Network
LSTM	Long Short-Term Memory
AGC-LSTM	Attention Enhanced Graph Convolutional LSTM
GSP	Graph Signal Processing
DCNN	Diffusion-Convolutional Neural Networks
MPNN	Message Passing Neural Networks
SGC	Simple Graph Convolution
PTC	Predictive Toxicology Challenge
GPU	Graphics Processing Unit
GC	Graph Convolution
lr	Learning rate
wd	Weight decay
Hid. dim.	Hidden layer's dimension
acc.	Accuracy
Avg.	Average

General Introduction

Graphs are powerful data structures that are used to model objects and their relationships. Thanks to their ability to capture rich information about interacting entities, graphs are used to model a wide range of real-world data: social networks, molecular graph structures, biological protein-protein networks and recommender systems among others. For control engineering, graphs are particularly interesting because many physical systems can be represented using this data structure. For example, an articulated robot's bodies can be represented with a graph's nodes while the joints linking the bodies together can be represented with a graph's edges. This makes it possible to design powerful, graph neural network (GNN) architectures for many relevant tasks such as inference, model-predictive control and system identification [62] as well as human action recognition [66]. Because GNN architectures have such a wide range of applications, research in this area is very active and state-of-the-art research papers are published regularly, meaning that there is a lot of room for impactful research and continuous improvement.

GNNs were developed to effectively handle graph-structured data. They learn latent representations for an input graph's nodes by recursively aggregating neighboring node features for each node, capturing important structural information about a node's neighborhood. The learned representations can then be used for various problems, such as node classification and graph classification, the latter of which is the problem we tackle in this work.

Recently, deep GNN architectures based on graph convolutions have emerged with state-of-the-art results on many graph-related problems. Of particular interest are Graph Convolutional Networks (GCNs) [40] and Graph Isomorphism Networks (GINs) [84], both of which revolutionized the field of graph neural architecture design. The GCN architecture is noteworthy because its authors introduced an efficient implementation of the convolution

operation on graphs while GIN's authors provided important theoretical conditions for a GNN to be invariant to node permutations (which GIN satisfies).

Another interesting state-of-the-art architecture is the Graph Attention Network (GAT) [74], which introduced the usage of attention mechanisms in GNN architectures. Attention mechanisms, originally introduced in natural language processing applications, are important because they allow a GNN to focus on the most important neighbors of a given node in order to compute new representations, rather than giving all neighbors equal importance by default.

Motivated by the recent advances in GNNs and in attention mechanisms for graph-structured data, we wanted to explore the question of whether two reference GNN architectures, namely GCN and GIN, would benefit from using attention on the particular task of graph classification. In an attempt to address this question, we propose in this work two novel GNN architectures, GCNA and GINA, that extend GCN and GIN respectively with the attention mechanism implemented in the GAT [74]. Our two architectures exhibit the important property of invariance to *graph isomorphism*, which guarantees that the same output is produced for two equivalent graphs. To the best of our knowledge, our work is the first to consider using attention with these two particular architectures. Our experiments on four benchmark graph datasets show promising results where, on some datasets, using attention helps improve the performance by up to 14%. More specifically, our contributions in this work are as follows:

- We provide an extensive overview of the current, state-of-the-art literature on graph neural networks, including breakthrough applications in the field of control theory.
- We propose two new GNN architectures, *Graph Convolutional Network with Attention* (GCNA) and *Graph Isomorphism Network with Attention* (GINA), leveraging existing state-of-the-art architectures and augmenting them with an attention mechanism while preserving the property of invariance to node permutations (or graph isomorphism).
- We make use of cutting-edge, powerful software tools for implementing deep learning models and computation in the Python programming language.
- We implement a rigorous and thoroughly detailed experimental procedure, in line with best practices in the field of machine learning.
- We evaluate our proposed GCNA and GINA architectures, and compare them against baselines (GCN and GIN) in a graph classification task on a number of relevant benchmark datasets which are often used in state-of-the-art research papers.

- We provide interesting possible perspectives to explore, should any students decide to build on this work in the future. To this end, we also make our code publicly available.

The remainder of this thesis is organized as follows. In Chapter 1, we introduce important graph-related definitions and notations that we use throughout this work. In Chapter 2, we give an overview of important machine learning notions ranging from supervised learning, classification and regression to overfitting and regularization. We then present some classical supervised learning models before introducing artificial neural networks and related concepts. In Chapter 3, we introduce graph neural networks, which are the artificial neural networks at the heart of this work, and the notion of graph convolution. We also review the most prominent graph neural network architectures in the literature, including the GCN [40] and the GIN [84]. In Chapter 4, we introduce the concept of attention with a focus on attention mechanisms for graph neural networks and, more specifically, the attention mechanism implemented in the GAT [74]. In Chapter 5, we present the architectures we propose, consisting in GCN with attention (GCNA) and GIN with attention (GINA). In Chapter 6, we detail our experimental procedure where we evaluate our architectures on four benchmark graph datasets. We give, in particular, the details of the tested architectures, as well as the training and hyperparameter tuning procedures, and discuss the observed results. Finally, Chapter 6.4.3 concludes this thesis.

CHAPTER 1

Graph Theory: Concepts and Notations

In this chapter, we introduce some graph theory concepts that are relevant to the subsequent chapters as well as their related notations. We also define the different types of graphs that are relevant to our work.

1.1 General Definitions

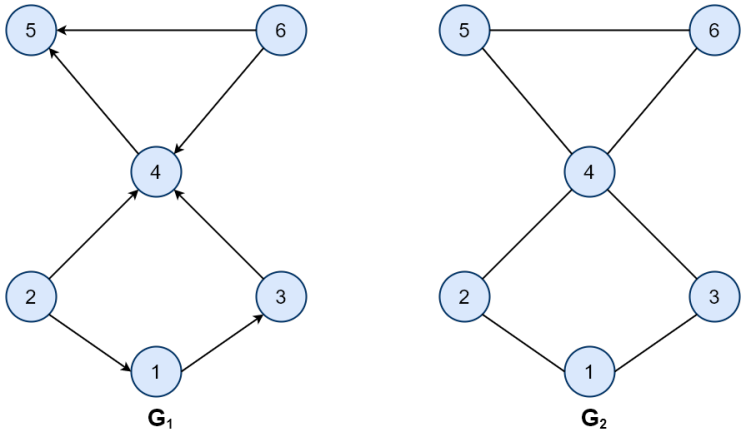
Graphs are a mathematical tool designed to represent a set of objects and the relations between them. The objects are represented by points called vertices and each pair of connected objects is called an edge. Graphs, as well as graph-related concepts, are formally defined in [63] as follows:

Definition 1.1. (Graphs) A graph G is a pair of non-empty sets (V, E), where $V = \{v_1, \ldots, v_n\}$ is the set of *vertices* (also called *nodes* or *points*) and E is the set of *edges* (also called *lines* or *links*), formed by pairs of vertices (v_i, v_j).

A common way of representing a graph G in practice is by its *adjacency matrix* $A \in \mathbb{R}^{n \times n}$, where $n = |V|$ is the number of vertices (or the *order* of the graph), and such that the element on the i^{th} row and j^{th} column $a_{ij} = 1$ if there is an edge between nodes v_i and v_j (i.e., $(v_i, v_j) \in E$) and $a_{ij} = 0$ if there is no edge.

We say that a graph is *undirected* if the edges have no orientation; that is, the edge (v_i, v_j) is identical to the edge (v_j, v_i). In this case, the adjacency matrix A is symmetric. If the

edges are oriented, we say that the graph is *directed*. Edges can also be between a node and itself: $(v_i, v_i) \in E$. In that case, these edges are referred to as *self-loops*. Fig. 1.1 shows a directed graph and an undirected one along with their adjacency matrices.



$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix} \qquad \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Figure 1.1: Example of a directed graph (G_1) and an undirected one (G_2) and their corresponding adjacency matrix.

In addition to the adjacency matrix A, a graph G usually has a *node feature matrix*[1] $X \in \mathbb{R}^{n \times d}$, where d is the number of features for each node. Each row of X represents the feature vector X_i of node v_i. In some cases, a set of edge features—represented by an *edge feature matrix*—is also available to characterize the graph along with A and X.

In this work, we only consider undirected graphs with no self-loops, and we only use node features. If the graph has no node features (i.e., purely structural graph), we take $X = I_n$, where n is the number of nodes and $I_n \in \mathbb{R}^{n \times n}$ is the identity matrix. We now give definitions of various graph-related concepts that we use later on, namely the notions of neighborhood, node degree and degree matrix and walk.

Definition 1.2. (Neighboring nodes and neighborhood) Given a graph $G = (V, E)$, two nodes $v_i, v_j \in V$ are said to be *neighbors*, or *adjacent* nodes, if $(v_i, v_j) \in E$, and we denote with $\mathcal{N}_{v_i} = \{v_j \in V | (v_i, v_j) \in E\}$ the *neighborhood* of v_i. If G is directed, we distinguish

[1]For the sake of brevity, we sometimes refer to the node feature matrix X as "feature matrix" in the rest of this paper.

between *incoming neighbors* of v_i (nodes $v_j \in V$ such that $(v_j, v_i) \in E$) and *outgoing neighbors* of v_i (nodes $v_j \in V$ such that $(v_i, v_j) \in E$).

Definition 1.3. (Node degree and degree matrix) Given a graph $G = (V, E)$, the *degree* of a node $v_i \in V$ is the number of its neighbors. Formally:

$$\deg(v_i) = |\mathcal{N}_{v_i}| = |\{v_j \in V | (v_i, v_j) \in E\}|. \tag{1.1}$$

The *degree matrix* D for G is an $n \times n$ diagonal matrix defined as follows:

$$D_{ij} = \begin{cases} \deg(v_i) & \text{if } i = j \\ 0 & \text{otherwise.} \end{cases} \tag{1.2}$$

Note that if G is directed, we distinguish between *indegree* (number of incoming neighbors), and *outdegree* (number of outgoing neighbors) of a node.

Definition 1.4. (Walks) Given a graph $G = (V, E)$, a *walk* is an alternating sequence of vertices and edges $v_1, e_2, v_2, e_3, v_3, e_4, \ldots, e_k, v_k$ where each edge $e_i = (v_{i-1}, v_i) \in E$.

In the following section, we introduce the important notion of *graph isomorphism* as well as the *Weisfeiler-Lehman* (WL) [81] isomorphism test.

1.2 Graph Isomorphism and Weisfeiler-Lehman Test

Informally speaking, we say that two graphs with the same number of vertices and edges are isomorphic if their vertices are connected in the same way. A formal definition of graph isomorphism, according to the authors of [82], is given by:

Definition 1.5. (Graph Isomorphism) Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, an *isomorphism* of graphs G_1 and G_2 is a bijection f between the set of vertices of V_1 and V_2: $f : V_1 \to V_2$, such that any two vertices v, u of G_1 are adjacent ($(v, u) \in E_1$) in G_1 if and only if $f(v)$ and $f(u)$ are adjacent ($(f(v), f(u)) \in E_2$) in G_2. We say that G_1 and G_2 are *isomorphic*.

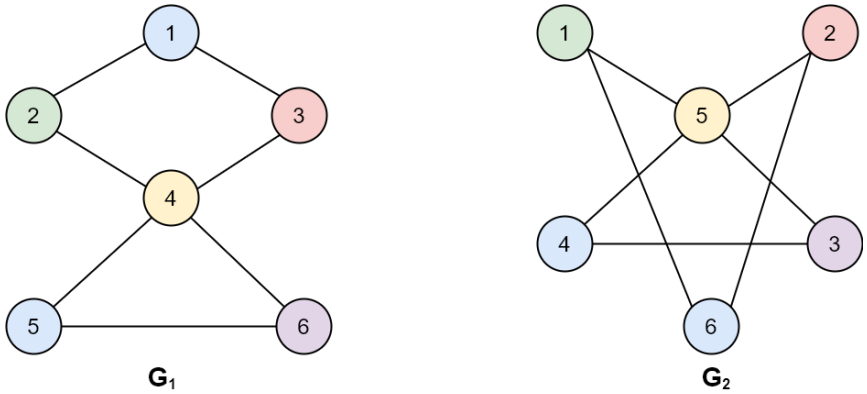


Figure 1.2: The two graphs shown here are isomorphic, meaning that they are the same up to a permutation of vertices (nodes).

Definition 1.5 states that isomorphic graphs have the same structure independently of the vertex indexing. That is, they are identical up to a permutation of vertices. In Fig. 1.2, the two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic even though their drawings are different. In fact, we obtain the second graph by applying the mapping $f : V_1 \to V_2$ in Fig. 1.3.

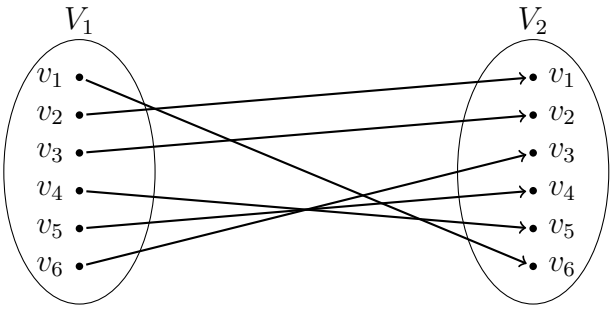


Figure 1.3: The node permutation $f : V_1 \to V_2$ that transforms G_1 into G_2 (see Fig. 1.2).

Graph isomorphism is important in graph neural architecture design. Indeed, a graph neural network should ideally be invariant to node permutations in that the same output should be obtained for two isomorphic graphs.

In general, finding whether two graphs are isomorphic is a challenging problem, and it is an especially important problem in computational complexity theory. This is mainly because it is not known if an algorithm exists to solve this problem in polynomial time, and it is also unknown whether the problem is NP-Complete [65]. One algorithm that is often used to test whether two graphs are isomorphic is the Weisfeiler-Lehman (WL) test [81] whose

main idea is to iteratively compute a canonical representation—a *coloring*[2]—for the nodes of a given graph. If two graphs do not have the same coloring, then they are definitely not isomorphic. However, two non-isomorphic graphs can share the same coloring. This means that the WL test alone cannot prove that two graphs are isomorphic. A simplified version of the (1-dimensional) WL test is presented in Alg. 1, where a hash function is used to map variable-sized inputs to a fixed-sized output and where node attributes $h_i^{(t)}$ are usually scalar integers (initialized to 1 if there is no attribute).

Data: Initial node coloring $(h_1^{(0)}, \ldots, h_n^{(0)})$ for graph $G = (V, E)$
Result: Final node coloring $(h_1^{(T)}, \ldots, h_n^{(T)})$
$t \leftarrow 0$;
repeat
  for $v_i \in V$ **do**
    $h_i^{(t+1)} \leftarrow \text{hash}\left(\{h_j^{(t)}\}_{v_j \in \mathcal{N}_{v_i}}\right)$;
  end
  $t \leftarrow t + 1$;
until *stable node coloring is reached*;

Algorithm 1: The 1-dimensional Weisfeiler-Lehman [81] test.

Artificial neural networks for processing graph data that are permutation-invariant usually have convolution operators which are based on the WL algorithm [40, 89], more details on which algorithm can be found in [16].

In the next chapter, we provide an overview of important concepts in machine learning and artificial neural networks. This allows us to lay the groundwork for a subsequent discussion of specialized neural network architectures designed to process graph-structured data.

[2]Graph coloring consists in attributing labels (or "colors") to the vertices of a given graph such that adjacent vertices cannot have the same color.

CHAPTER 2

Machine Learning and Artificial Neural Networks: an Overview

Machine learning (ML) is the study of computer algorithms that can learn from data. Broadly speaking, the goal of a machine learning algorithm is to build a mathematical model which can leverage a set of input data (i.e., a collection of examples) to learn to make accurate predictions about previously unseen data (supervised learning) or find useful patterns in the input data itself (unsupervised learning).

Since the problem we tackle falls into the category of supervised learning problems, we focus in this chapter on concepts relating to that particular category of problems. Furthermore, because supervised learning is an expansive field, a thorough overview is out of the scope of this work. As such, this chapter is only intended as a brief introduction to some of the most fundamental concepts. We start by defining what supervised learning is before detailing two classical machine learning algorithms. Then, we dive into artificial neural networks by describing the most fundamental type, detailing how these networks are set up and trained in the context of classification—one of two types of supervised learning problems. Next, we discuss some of the common problems which arise when training neural networks as well as practical solutions to these problems. Finally, we briefly discuss two specialized, more complex and very important types of neural networks which are currently used to power some impressive machine learning applications.

2.1 Supervised Learning

2.1.1 Definition

In the book *Dive into Deep Learning* [87], the authors define a supervised learning problem as the task of predicting *labels* (i.e., outcomes or targets) given *inputs* (i.e., features or covariates). Inputs are usually denoted by $\mathbf{x}$ while labels are denoted by y. Each (input, label) pair is called an example, typically denoted by $(\mathbf{x}^{(i)}, y^{(i)})$. A collection of m examples forms a *dataset*, $\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{m}$.

Definition 2.1. Let $\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{m}$ be a dataset. Supervised learning aims to produce a model f_θ, where θ represents the model's parameters, that maps the inputs $\mathbf{x}^{(i)}$ to a prediction $f_\theta(\mathbf{x}^{(i)})$ that is as close to $y^{(i)}$ as possible.

The term *supervised learning* comes from the view that the target $y^{(i)}$ we want our model to correctly predict is provided by us, in our role as supervisors, because we are "showing" the system what to do. Concretely, we want to obtain a model such that the prediction $f_\theta(\mathbf{x}^{(i)})$ matches the label $y^{(i)}$ as closely as possible. When we say that a model f_θ "learns", this means that its parameters θ are iteratively optimized so that its predictions become more accurate.

2.1.2 Training and Test Sets

It is important to introduce the notion of *training* and *test* sets. We want to train and optimize our model on a subset of the whole dataset called the training set. Why? Because if we use the whole dataset for training, although we may manage to optimize our model on this whole dataset extremely well, there is no guarantee as to how it will perform when exposed to unseen data examples.[1] Reserving a portion of the dataset for testing (i.e., the test set) gives us a tool to check our learned model's ability to generalize to new, unseen data. In practice, 80% of a dataset is usually reserved for training while the remaining 20% are used for testing, although these percentages can vary, of course.

Ultimately, the goal of supervised learning is to learn a model that captures important patterns and dependencies present in training data while, at the same time, generalizing well enough to unseen data. Perfectly satisfying both objectives is, however, impossible: a model may perfectly capture the patterns in the training set, but it runs the risk of being overly sensitive to noise present in that same set, thus performing poorly on test

[1] Note that the success of a supervised learning model is based on the assumption that the labeled data used to train the model comes from the same distribution as the general unseen data, i.e., it is a representative sample of the unseen general data.

data. Striking a good balance between these two objectives is crucial in machine learning practice.

This ability to generalize to unseen data ties in with the notions of *overfitting* and *underfitting*, which we introduce in Section 2.6.

2.1.3 Loss Function

A loss function (also called a criterion) is a formal measure of how good (or bad) a model f_θ is at making predictions (i.e., how close a prediction $f_\theta(\mathbf{x}^{(i)})$ matches the actual label $y^{(i)}$). When training, we want to obtain a model such that this loss function (which we denote with L_θ) is minimized over all m_{train} training examples:

$$L_\theta = \frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \ell(y^{(i)}, f_\theta(x^{(i)})), \tag{2.1}$$

where ℓ, the loss for a single example, depends on the task at hand: usually the mean squared error for regression and cross-entropy for classification problems, both of which are defined in the next few sections.

2.1.4 Types of Supervised Learning Problems

Supervised learning problems can be split into two distinct categories, depending on the nature of the outputs (targets):

- **Regression:** Outputs are real numbers. For example: building a model to predict house prices. The input $\mathbf{x}$ may be any relevant feature(s) of a house (size, number of bedrooms, etc.) while the output y is the price of said house.
- **Classification:** The model predicts which class (among a discrete set of options) the input belongs to. For example: building a model to predict the class an input image belongs to. Classification options might include objects such as a car, house, phone, etc.

Despite being only one of many paradigms within machine learning, supervised learning is by far the most commercially successful of all, powering everything from text and speech translation to face recognition technology and everything in between, across a wide spectrum of industries.

2.1.5 Performance Evaluation

In this subsection, we present the *accuracy*, a classical and widely used performance measure on classification tasks, as well as a central evaluation procedure in ML, the so-called *cross-validation*.

Accuracy

As stated in Subsection 2.1.2, the performance of a machine learning model is evaluated on the test set in order to have an estimate of model's performance on future unseen data. Depending on the task at hand, the performance measure—or performance indicator—varies. Let us consider the case of classification. A very common performance measure is the *accuracy*, which measures the ratio of correctly classified examples over a given dataset. Formally, let us consider the set $\{\hat{y}^{(1)}, \ldots, \hat{y}^{(m)}\}$ of class labels predicted by a model and the set $\{y^{(1)}, \ldots, y^{(m)}\}$ of correct labels. The accuracy is defined as follows:

$$\text{Accuracy} = \frac{\sum_{i=1}^{m} \mathbf{1}_{y^{(i)}}(\hat{y}^{(i)})}{m}, \tag{2.2}$$

where $\mathbf{1}_a(b) = 1$ if $a = b$ and 0 otherwise.

Accuracy is best suited to estimate the performance on *well-balanced* datasets, i.e., datasets where the labels are distributed roughly uniformly among classes. On imbalanced datasets where some classes are over- (or under-) represented, other performance measures are used that are out of the scope of this work.

Cross-Validation

Cross-validation is a set of model evaluation techniques that are used when the test set is on the smaller side or, more generally, to obtain a more robust performance estimation. Let us consider a particularly popular variant of cross-validation, the so-called *k-fold cross-validation*. The idea in k-fold cross-validation is to split the dataset into k complementary subsets—or *folds*—as opposed to one training set and one test set, then for each fold $i \in \{1, \ldots, k\}$, use the remaining $k - 1$ folds to train the model and the current fold i as a test set to evaluate the model. That way, we have k data points (performance estimates) that we can average to get the overall performance of the model.

In practice, k-fold cross-validation is commonly used to compare and select the fitter model for a particular prediction task (model selection), as well as to select the best parameter values for a model (see hyperparameter tuning in Section 2.8).

2.2 Some Classical Machine Learning Algorithms

Before introducing artificial neural networks, which are by far the most dominant machine learning approach, we first describe two of the simplest and most popular supervised learning algorithms: linear regression and support vector machines (SVMs).

2.2.1 Linear Regression

Linear regression, as its name implies, solves a regression problem. The goal is, for a vector $\mathbf{x} \in \mathbb{R}^d$ with d features, to predict a scalar value $\hat{y} = f_\theta(\mathbf{x})$ that matches the label $y \in \mathbb{R}$ as closely as possible. Linear regression makes the assumption that the relationship between the independent features $\{x_1, \ldots, x_d\}$ and the dependent variable y is linear. This means that our prediction $\hat{y}$ is expressed as:

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b, \tag{2.3}$$

where $\{w_1, \ldots, w_d\}$ are the parameters of our model (also called *weights*) and $b \in \mathbb{R}$ is a bias term. Collecting all features and weights into two vectors $\mathbf{x}, \mathbf{w} \in \mathbb{R}^d$, we end up with a more compact notation:

$$\hat{y} = \mathbf{w}^\top \mathbf{x} + b. \tag{2.4}$$

The goal is to find the best parameters $\mathbf{w}$ and b such that the loss function is minimized. One of the most widely used loss functions in linear regression is the mean squared error (MSE), defined as follows:

$$\text{MSE} = \frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2, \tag{2.5}$$

m_{train} being the number of training examples.

2.2.2 Support Vector Machines

Support vector machines (SVMs) are most commonly used to solve classification problems. Their goal is fairly straightforward: to find the optimal hyperplane which linearly separates (i.e., classifies) data points into two classes. Although SVMs can be used for multiclass classification problems and they can also leverage an approach called "the kernel trick" to create nonlinear hyperplanes [9], we focus here on the simplest case:

- The classification problem is binary: $y^{(i)} \in \{-1, 1\}$.
- Features are two-dimensional: $\mathbf{x}^{(i)} \in \mathbb{R}^2$.

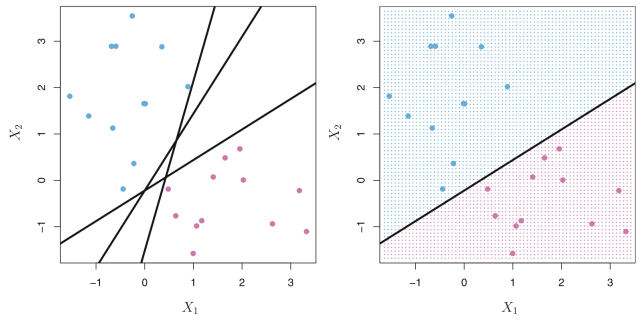


Figure 2.1: Left: There are two classes of observations, shown in blue and in purple. Three separating hyperplanes, out of many possible, are shown in black. Right: A separating hyperplane is shown in black. The blue and purple grid indicates the decision rule made by a classifier based on this separating hyperplane: a test observation that falls in the blue portion of the grid will be assigned to the blue class, and a test observation that falls into the purple portion of the grid will be assigned to the purple class. Figure from [36].

- Training examples are linearly separable—i.e., they lie in two distinct groups which can be clearly separated by a hyperplane (a line in the case of two-dimensional features) with no overlap between the two classes.

As illustrated in Fig. 2.1, there is an infinity of possible hyperplanes that separate the data of the two classes. The SVM algorithm finds the optimal hyperplane $h(\mathbf{x}) = 0$, where $h(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$, $\mathbf{w} \in \mathbb{R}^2$, $b \in \mathbb{R}$, such that the margin between the support vectors—i.e., the closest training examples to the separating hyperplane in each class—and this hyperplane is maximized.

Given the hyperplanes $\mathbf{w}^\top \mathbf{x}+b = 1$ and $\mathbf{w}^\top \mathbf{x}+b = -1$ that define the limit of classes 1 and -1 respectively (see Fig. 2.2), the margin to be maximized is given by $\frac{2}{\|\mathbf{w}\|}$. In addition, the optimal separating hyperplane should satisfy the following two constraints:

$$\mathbf{w}^\top \mathbf{x}^{(i)} + b \geq 1 \text{ when } y^{(i)} = 1, \tag{2.6}$$

$$\mathbf{w}^\top \mathbf{x}^{(i)} + b \leq -1 \text{ when } y^{(i)} = -1, \tag{2.7}$$

for each training example. This can be formulated as the equivalent constrained optimization problem:

$$\begin{aligned} \min_{\mathbf{w}} \quad & \frac{1}{2}\|\mathbf{w}\| \\ \text{subject to} \quad & y^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b\right) \geq 1, \end{aligned} \tag{2.8}$$

where the constraint in (2.8) is the combination of Eq. (2.6) and (2.7).

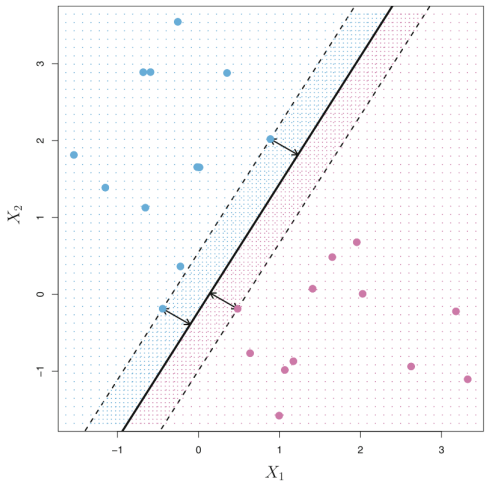


Figure 2.2: The maximal margin hyperplane is shown as a solid line. The margin is the distance from the solid line to either of the dashed lines. The two blue points and the purple point that lie on the dashed lines are the support vectors, and the distance from those points to the hyperplane is indicated by arrows. Figure from [36].

The optimization problem in (2.8) is solved in practice by the Lagrangian multipliers method. Once the optimal $\mathbf{w}$ and b—and therefore the maximal margin hyperplane—are found, classifying new examples $\mathbf{x}^{(i)}$ is straightforward:

$$h(\mathbf{x}^{(i)}) = \mathbf{w}^\top \mathbf{x}^{(i)} + b > 0 \Rightarrow y^{(i)} = 1, \tag{2.9}$$

$$h(\mathbf{x}^{(i)}) = \mathbf{w}^\top \mathbf{x}^{(i)} + b < 0 \Rightarrow y^{(i)} = -1. \tag{2.10}$$

For a more in-depth discussion of SVMs, we refer the reader to the book *An Introduction to Statistical Learning* [36].

2.3 Multi-Layer Perceptrons: Simple yet Powerful Artificial Neural Networks

Artificial neural networks (ANNs) are by far the most powerful and successful supervised learning tools in use today. Multi-layer perceptrons (MLPs), also known as *feed-forward neural networks* or *fully connected neural networks*, form the basis upon which more sophisticated—e.g., convolutional and recurrent—neural network architectures are built.

As explained in the book *Deep Learning* [21], the goal of an MLP is to approximate—i.e., learn—some function f^*. For classification problems, this function $y = f^*(\mathbf{x})$ maps an

input $\mathbf{x}$ to a class y. An MLP, parameterized by θ, learns the values of the parameters θ for which $y = f_\theta(\mathbf{x})$ gives the best approximation of the function f^*. The universal approximation theorem [32] posits that MLPs—given enough neurons—can approximate any function. However, it doesn't guarantee the *learnability* of such an approximation—i.e., any desired function can be approximated by an MLP, but actually learning the parameters which would allow an MLP to approximate that function is not guaranteed.

Modeled loosely after real networks of biological neurons in the brain, artificial neural networks are an attempt at translating neuroscience research findings into a set of linear transformations followed by nonlinear functions, mathematically emulating an extremely simplified model of how the brain processes information.

Generally speaking, many tasks which require mapping an input vector $\mathbf{x}$ to an output y can be accomplished with artificial neural networks (given sufficient labeled training data and sufficiently large models). MLPs can learn increasingly complex functions by stacking multiple "layers", hence the term *multi-layer* in the name. Mathematically, this can be expressed as a composition of functions. For example, a two-layer perceptron can be written as $y = f(\mathbf{x}) = f^{(2)}\left(f^{(1)}(\mathbf{x})\right)$, where $f^{(1)}$ and $f^{(2)}$ represent the first and second layers of the network, respectively. Intermediate layers are called *hidden* layers while the final layer ($f^{(2)}$ in this case) is called the *output* layer.

2.3.1 Describing Feedforward Layers

Feedforward layers are the basic building blocks of multi-layer perceptrons. The word *feedforward* comes from the fact that the output of each layer is only fed to the next layer, with no feedback connections present. MLPs are built by successively stacking such layers, a concept best explained through an example. Fig. 2.3 describes a two-layer perceptron, where the outputs of each layer (hidden and output) can be described with the following equations:

$$h = \mathbf{w}^{(1)}\mathbf{x} + b^{(1)}, \tag{2.11}$$

$$o = \mathbf{w}^{(2)}h + b^{(2)}, \tag{2.12}$$

where $\mathbf{x} \in \mathbb{R}^4$ is the input vector, $h \in \mathbb{R}^5$ is the hidden layer's "activation" vector and $o \in \mathbb{R}^3$ is the output. Since the output layer has three neurons, this particular MLP can be used to classify inputs into three different classes. $\mathbf{w}^{(1)} \in \mathbb{R}^{5\times4}$ and $b^{(1)} \in \mathbb{R}^5$ are the first layer's learnable parameters while $\mathbf{w}^{(2)} \in \mathbb{R}^{3\times5}$ and $b^{(2)} \in \mathbb{R}^3$ are the second layer's learnable parameters. It should be clear that each layer simply represents a linear transformation.

This primitive version of our two-layer perceptron example is, however, incomplete—a

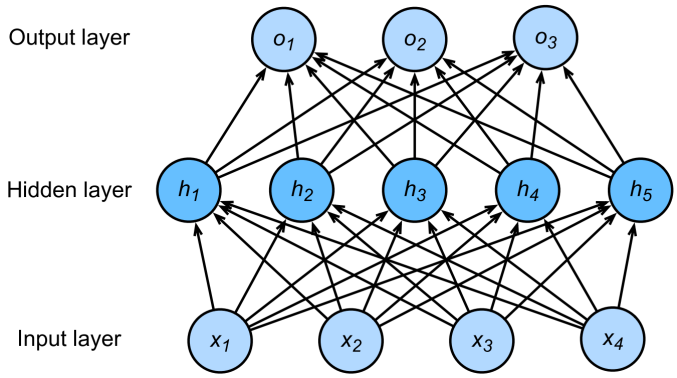


Figure 2.3: A two-layer perceptron. The hidden layer has 5 hidden units (i.e., neurons) while the output layer has 3. Figure from [87].

composition of linear transformations is itself a linear transformation. Indeed:

$$\begin{aligned} o &= \mathbf{w}^{(2)}h + b^{(2)} \\ &= \mathbf{w}^{(2)}(\mathbf{w}^{(1)}\mathbf{x} + b^{(1)}) + b^{(2)} \\ &= (\mathbf{w}^{(2)}\mathbf{w}^{(1)})\mathbf{x} + (\mathbf{w}^{(2)}b^{(1)} + b^{(2)}) \\ &= \mathbf{w}\mathbf{x} + b. \end{aligned} \tag{2.13}$$

Essentially, we would be wasting computational resources trying to learn multiple layers' worth of parameters when a single-layer perceptron would give the same results, not to mention missing the representational power of a key element: nonlinear activation functions.

Nonlinear Activation Functions

To truly unlock the power of multi-layer architectures and make sure that stacking more MLP layers would result in the ability to learn more complex representations, *nonlinear activation functions* (denoted here by σ) have been introduced. These functions are applied element-wise following a linear transformation. The default recommendation (according to [21]) is to use the rectified linear unit (ReLU) function: $\text{ReLU}(z) = \max\{z, 0\}$, illustrated in Fig. 2.4. The ReLU function is attractive because it remains piecewise linear (i.e., composed of two linear parts). As such, it retains some of the properties which make gradient-based optimization methods work well with linear models.

There are other nonlinear activation functions in use. One such function is the LeakyReLU

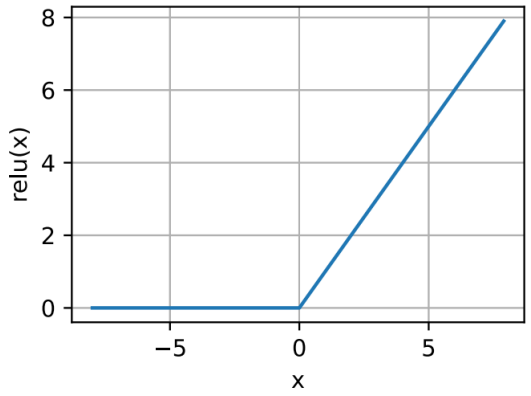


Figure 2.4: The rectified linear unit (ReLU) function. Figure from [87].

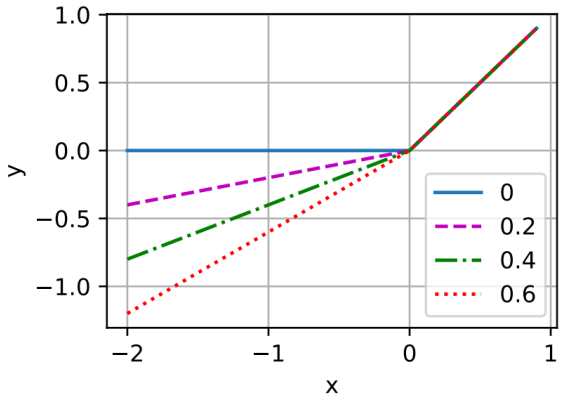


Figure 2.5: The LeakyReLU function for different values of α. Notice that ReLU is a particular case of LeakyReLU ($\alpha = 0$). Figure from [87].

nonlinearity, illustrated in Fig. 2.5. For a given $\alpha \in [0, 1]$, it is defined as:

$$\text{LeakyReLU}(z) = \begin{cases} z & \text{if } z > 0, \\ \alpha z & \text{otherwise.} \end{cases}$$

The main advantage of LeakyReLU over ReLU is that it can prevent neurons from becoming "dead" when using gradient-based optimization methods.[2]

[2]Neurons are said to be "dead" when their outputs do not change during optimization (meaning, in practice, that they become unable to learn anything). LeakyReLU alleviates this problem thanks to its small negative slope, preventing gradients from getting stuck on zero. We discuss why this is important in Section 2.4.

Applying a nonlinear activation function, our two-layer perceptron becomes:

$$h = \sigma(\mathbf{w}^{(1)}\mathbf{x} + b^{(1)}), \tag{2.14}$$

$$o = \mathbf{w}^{(2)}h + b^{(2)}. \tag{2.15}$$

We show now how the output o of our MLP can be used to perform classification.

2.3.2 Case of Binary and Multiclass Classification

Eq. (2.15) gives us the raw output vector of our network. To give this output a more intuitive meaning, let us consider a multiclass classification task. Each element of the vector o can be interpreted as the chance of the input $\mathbf{x}$ belonging to the corresponding class. To compute the probability of each class i, we normalize the outputs between 0 and 1 by passing o to the softmax function:

$$\hat{y}_i = \text{softmax}(o) = \frac{\exp(o_i)}{\sum_j \exp(o_j)}, \tag{2.16}$$

where o_i is the i^{th} element of o and $\hat{y}_i$ is the estimated probability of $\mathbf{x}$ belonging to class i. Now, it is clear that $\sum_i \hat{y}_i = 1$ and the elements $\hat{y}_i$ of $\hat{y}$ can be interpreted as estimated conditional probabilities of each class: $\hat{y}_1$ is the probability of $\mathbf{x}$ belonging to class 1, $\hat{y}_2$ is the probability of $\mathbf{x}$ belonging to class 2 and so on. The predicted class is simply the one with the highest probability.

In cases where the classification problem is binary, the output o is a scalar. Instead of the softmax, we use the sigmoid function:

$$\hat{y} = \text{sigmoid}(o) = \frac{1}{1 + \exp(-o)}. \tag{2.17}$$

As illustrated in Fig. 2.6, the sigmoid function collapses its inputs from a range of $[-\infty, \infty]$ to $[0, 1]$, leading to a probabilistic interpretation of $\hat{y}$, in a similar fashion to the multiclass classification case. For classification, we use 0.5 as a threshold: $0 \leq \hat{y} < 0.5$ means that $\mathbf{x}$ belongs to class 1 while $0.5 \leq \hat{y} \leq 1$ means that $\mathbf{x}$ belongs to class 2.

2.3.3 Cross-Entropy Loss

To measure how good the probabilities predicted by an ANN are, one of the most commonly used loss functions for multiclass classification problems is the *cross-entropy loss* function. Before defining it, however, it is important to point out that the label of an example $\mathbf{x}$ must be contained in a vector $y = [0, \ldots, 1, \ldots, 0]$ which contains a single 1 at the i^{th}

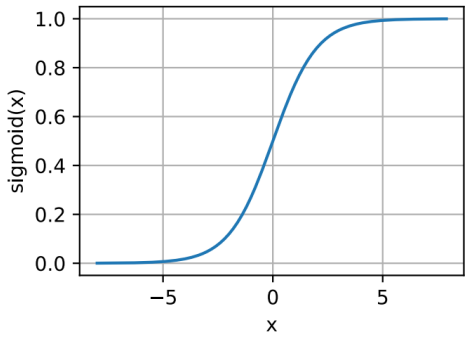


Figure 2.6: The sigmoid function. Figure from [87].

position, indicating that $\mathbf{x}$ belongs to class i. Now, we can express the cross-entropy loss for a single example as follows:

$$\ell(y, \hat{y}) = -\sum_j y_j \log \hat{y}_j. \tag{2.18}$$

Averaged over a full training set $\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{m_{\text{train}}}$, the cross-entropy loss becomes:

$$L = -\frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \sum_j y_j^{(i)} \log \hat{y}_j^{(i)}. \tag{2.19}$$

The cross-entropy loss function computes how probable the actual classes are according to our model, given the features. It relies on the principle of likelihood maximization, a detailed explanation of which can be found in [87].

In the case of binary classification (where the true label $y = 0$ for class 1 and $y = 1$ for class 2), the loss function is called *binary* cross-entropy. For a single example, it is expressed as follows:

$$\ell(y, \hat{y}) = -\big(y \log \hat{y} + (1 - y) \log(1 - \hat{y})\big). \tag{2.20}$$

Averaged over the full training set, it becomes:

$$L = -\frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} y^{(i)} \log \hat{y}^{(i)} + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)}). \tag{2.21}$$

Now that a basic MLP architecture and an appropriate loss function have been detailed, the next section will focus on describing the training procedure for neural networks.

2.4 Training Neural Networks: Stochastic Gradient Descent and Variants

Training a neural network means searching for a parameter set θ that minimizes the loss L over the training set.

Although nonlinearities give neural networks the ability to learn more complex representations, they also make loss functions nonconvex. This means that training ANNs calls for gradient-based optimizers that are more sophisticated than the classical gradient descent algorithm and that convergence to a global minimum is not guaranteed. In practice, loss functions for ANNs present many local minima, and these optimizers usually reach a low enough value for the cost function to, at least, result in acceptable model performance. It should be noted that this nonconvex nature of the problem means that gradient descent-based algorithms are sensitive to the initial values of the model's parameters. For MLPs, the recommendation is to initialize weights $\mathbf{w}$ to small random values and biases b to zero or to small positive values [21].

In this section, we focus on two optimization algorithms in particular: stochastic gradient descent and Adam [39], a more sophisticated and one of the best performing optimizers currently in use.

2.4.1 Stochastic Gradient Descent

Stochastic gradient descent (SGD) is the quintessential optimization algorithm for deep learning. Recall that loss functions are typically computed over a full training dataset (Eq. (2.5) and (2.19)). For very large datasets, this can become very computationally expensive. Consider the following loss function for a training dataset $\{\mathbf{x}^{(i)}, y^{(i)}\}_{i=1}^{m_{\text{train}}}$:

$$L_\theta = \frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \ell(y^{(i)}, f_\theta(\mathbf{x}^{(i)})). \tag{2.22}$$

Classical gradient descent requires us to compute the following gradient expression:

$$\nabla_\theta L_\theta = \frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \nabla_\theta \ell(y^{(i)}, f_\theta(\mathbf{x}^{(i)})), \tag{2.23}$$

where $\nabla_\theta L_\theta$ is the gradient of the loss function with respect to the model's parameters θ [21].

As m_{train} grows to very large numbers, taking a single gradient step becomes prohibitively expensive. As explained in [21], the insight of SGD is that the true gradient may be

approximated using a small subset of m' examples drawn uniformly from the training set, potentially lowering the computational cost considerably. The gradient estimate is given by:

$$\nabla_\theta L_\theta = \frac{1}{m'} \nabla_\theta \sum_{i=1}^{m'} \ell(y^{(i)}, f_\theta(\mathbf{x}^{(i)})). \tag{2.24}$$

Next, SGD updates the parameters θ with:

$$\theta \leftarrow \theta - \eta \nabla_\theta L_\theta, \tag{2.25}$$

where η is the algorithm's *learning rate*, which determines the size of the step taken towards the optimum at each iteration.

In machine learning practice, the common procedure is to first randomly shuffle the training set, then use succesive subsets of m' examples (called *minibatches*) to compute the loss, its gradients and then update θ. A full pass through the training set is called an *epoch*, and the optimization process can be repeated for any number of epochs chosen by the user.

Another interesting concept in training neural networks is the use of dynamic learning rates, meaning that η can be programmed to decay as a function of the number of training epochs. Doing so can allow the optimizer to better "close in" on the minimum of the loss function and may reduce the possibility of overshooting it. This is especially important for SGD, since the trajectory of the parameters throughout optimization is more noisy than for standard gradient descent due to its stochastic nature [87].

2.4.2 Adam

As mentioned in Section 2.4, the Adam optimizer [39] is a more sophisticated variant of the SGD algorithm. Adam efficiently combined various concepts and techniques for effective optimization, which have made it one of the more robust and effective stochastic gradient-based optimization algorithms. Notably, it uses exponential moving averages to estimate the first (the mean m) and the second (the uncentered variance v) moments of the gradients, both of which are initialized at zero:

$$m \leftarrow \beta_1 m + (1 - \beta_1) \nabla_\theta L_\theta, \tag{2.26}$$

$$v \leftarrow \beta_2 v + (1 - \beta_2) (\nabla_\theta L_\theta)^2, \tag{2.27}$$

where β_1 and β_2 are positive parameters to control the exponential decay rates of the moving averages. The initialization with zeros causes moment estimates to be biased toward small values, especially at the initial time steps of optimization, which is why a bias correction step follows:

$$\hat{m} \leftarrow m / (1 - \beta_1^t), \tag{2.28}$$

$$\hat{v} \leftarrow v / (1 - \beta_2^t), \tag{2.29}$$

where t is the current time step. Finally, the model's parameters are updated with:

$$\theta \leftarrow \theta - \eta \hat{m}/(\sqrt{\hat{v}} + \epsilon), \tag{2.30}$$

where η is the learning rate and ϵ is a constant used for numerical stability. Concepts such as dynamic learning rates and the use of minibatches carry over from SGD, naturally.

Before talking about some of the common problems encountered when training neural networks (and the solutions), there is one important concept we need to introduce: how do we actually compute the gradients $\nabla_\theta L_\theta$? The next section discusses this question.

2.5 Backpropagation

Backpropagation is an efficient way of computing a neural network's gradients in practice. Before describing it, however, let us provide some context.

When using a feedforward network to predict an input $\mathbf{x}$'s label $\hat{y}$, information flows through the various layers until an output is produced. This is called *forward propagation*. During training, forward propagation continues until the cost function L_θ is evaluated. This information flow can be visualized through computational graphs, which decompose a neural network's computations into a series of elementary operations. Fig. 2.7 illustrates an example of a computational graph.

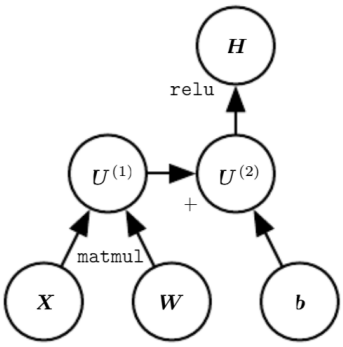


Figure 2.7: A computational graph for the expression $H = \text{ReLU}(XW + b)$. Figure from [21].

Similarly, we can exploit this flow of information in reverse: starting from the cost function, we can go backward through the computational graph and the various functions and variables from which it is built, recursively applying the chain rule of calculus until we finally compute an algebraic expression for the gradients $\nabla_\theta L_\theta$. This algorithm is called

backpropagation [60], and is one of the most important breakthroughs in the history of machine learning, powering every modern application of neural networks of all kinds.

Modern software tools for machine learning computation automatically build computational graphs of neural networks and evaluate gradients in the background, thus eliminating the need to compute gradient expressions by hand. Actually evaluating gradient expressions in computers, however, requires some extra considerations, the details of which can be found in [21].

2.6 Overfitting and Underfitting

By now, we have seen how a (simple) neural network works, how to establish a loss function for classification and how to train and optimize the model's parameters using gradient-based algorithms. When training and testing neural networks, however, we can run into two potential problems: overfitting or underfitting. Before explaining what these two problems mean and what they entail, it is important to take a step back and look at what we are trying to achieve when training a neural network.

Fundamentally, our goal with supervised learning is to discover patterns in the distribution from which our training set was drawn, allowing a learned model to, hopefully, classify unseen data correctly. However, as datasets represent only a small sample of real-world data, there is a risk that whatever model we end up with will have discovered patterns that are not adequately representative of the underlying data distribution and thus, performance on unseen (i.e., test) data will inevitably suffer. This phenomenon—learning a model which fits a training set very well but fails to generalize to new data—is called *overfitting*. On the other hand, failing to perform well even on the training set itself is called *underfitting*. It is typically indicative of a model that is too simple to be able to capture meaningful patterns in the data for the task at hand.

A model's complexity (i.e., its ability to fit a wide variety of functions), as stated in [21], determines whether it is likely to overfit or underfit. We want a model that is complex enough to capture the essential features of the data and still generalize well, but not too complex that it essentially "memorizes" the properties of the training data and, by doing so, fails to generalize. Fig. 2.8 illustrates this compromise.

2.7 Regularization

Solving the problem of underfitting is usually a matter of increasing the model's complexity. Reducing overfitting, however, is more complicated. It can be done implicitly by reducing the model's complexity, or explicitly using a set of techniques called *regularization*.

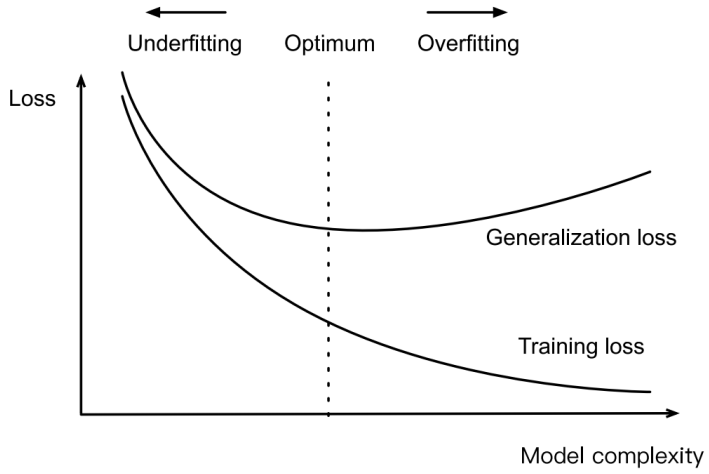


Figure 2.8: Typical curves illustrating the influence of model complexity on underfitting and overfitting. Figure from [87].

The authors of [21] define regularization as "any modification made to a learning algorithm that is intended to reduce its generalization error but not its training error". Although many regularization strategies exist, we focus here on the ones we use in our own experiments: weight decay and dropout.

2.7.1 Weight Decay

Weight decay, commonly known as L_2 regularization, is perhaps the most widely used regularization technique in machine learning [87]. L_2 regularization drives a model's weights closer to the origin by adding a penalty term (the sum of the squared Euclidean norms of the weights) to the loss function.

For the example of linear regression described in Subsection 2.2.1, recall that the unregularized loss function (MSE) had the expression:

$$\frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{2.31}$$

Adding L_2 regularization, the loss function becomes:

$$\frac{1}{m_{\text{train}}} \sum_{i=1}^{m_{\text{train}}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2 + \frac{\lambda}{2}||\mathbf{w}||^2, \tag{2.32}$$

where $\lambda > 0$ is a hyperparameter which controls how much emphasis we place on regularization strength compared to minimizing the prediction error. Intuitively, L_2 regularization

can be thought of as minimizing the weights of the model which contribute the least to minimizing the objective function (i.e., the prediction error in the case of MSE). By doing so, we reduce the model's bias towards the training data. A more thorough analysis of the effects of L_2 regularization can be found in [21].

2.7.2 Dropout

Dropout [68] is a computationally efficient and powerful method of regularizing deep neural networks (i.e., networks with a high number of layers).

Although deep neural networks are very powerful, they are also prone to overfitting. The authors of the dropout paper argue that one of the main characteristics of overfitting in neural networks is co-adaptation, where the activations of neurons in a given layer can become too reliant on a specific pattern of activations in the layer before. The authors theorize that neurons co-adapt to fix mistakes of other neurons in the previous layers during training. This co-adaptation, in turn, leads to overfitting because it partially results from sampling noise present in training data but not in test data, even if it is drawn from the same distribution.

Dropout helps break up co-adaptation by temporarily dropping out neurons—i.e., disabling them, along with all their incoming and outgoing connections—in a given layer with a probability p, thereby forcing all remaining active neurons in that layer to take responsibility for producing the desired output given an input sequence during training. Concretely, dropped neurons see their corresponding weights set to zero during forward propagation, while their gradients vanish during backpropagation. The random nature of dropout means that, after repeated forward and backward propagation operations over a full training set, we can expect most neurons (in layers where dropout is applied) to perform well individually on a wide variety of contexts. Fig. 2.9 visually illustrates what dropout does to an MLP.

Dropout can be performed on any hidden layer in a neural network, where a hyperparameter $p \in [0, 1]$ controls the probability of dropping each neuron in that layer. Experiments show that it improves the performance of neural networks on supervised tasks in computer vision, speech recognition, document classification and more [68].

2.8 Hyperparameter Tuning

The design of artificial neural architectures involves having to decide on the values of various parameters, some of which determine the ANN's architecture, such as the number of hidden layers and units or the type of nonlinear activation function. Other parameters

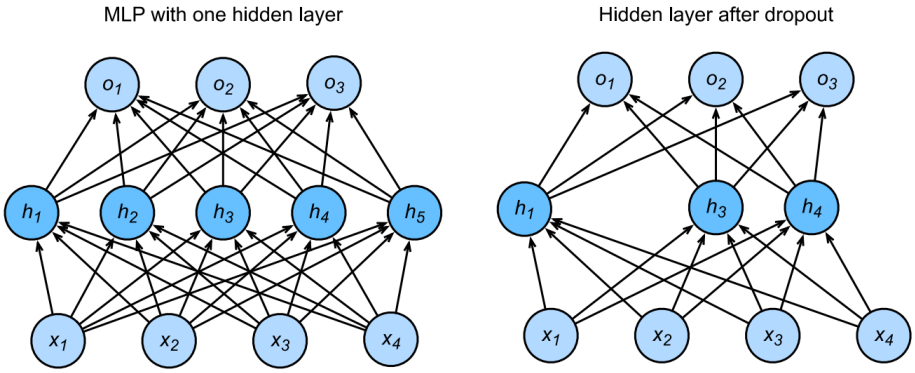


Figure 2.9: MLP before and after applying dropout. Here, the neurons h_2 and h_5 are dropped. Figure from [87].

are related to the optimization algorithm (such as the learning rate η) or to regularization (such as the dropout rate p). Such parameters are called *hyperparameters* and are crucial to the performance of an ANN.

The process of choosing the optimal set of hyperparameters is very important as it controls the overall behaviour of the model both on training and test sets. This process is often called *hyperparameter tuning* (or *hyperparameter optimization*). There are many ways to go about finding the optimal set of parameters, one of which is *grid search* [13], where we choose possible values for each hyperparameter, test each possible configuration—i.e., values combination—then choose the best one. Another method is *random search* [7], which is very similar to grid search, but instead of defining discrete values of parameters, we define an interval for each parameter and choose random values from those intervals. This method usually outperforms grid search as shown in Fig. 2.10.

While these methods yield better results, they are both computationally very expensive, especially when the training procedure takes a relatively long time. There are, however, more sophisticated approaches used by practitioners to perform hyperparameter tuning, such as Bayesian Optimization, which implements a more intelligent exploration of the search space and optimizes a surrogate of the objective function to reduce the computational cost.

2.9 Other Prominent Neural Network Architectures

So far, we have only talked about multi-layer perceptrons. Although powerful in their own right and appropriate for dealing with data which can be represented with real-valued vectors, they can prove unwieldy for handling other types of data such as images, sequential information or structured data. As a result, specialized types of neural network

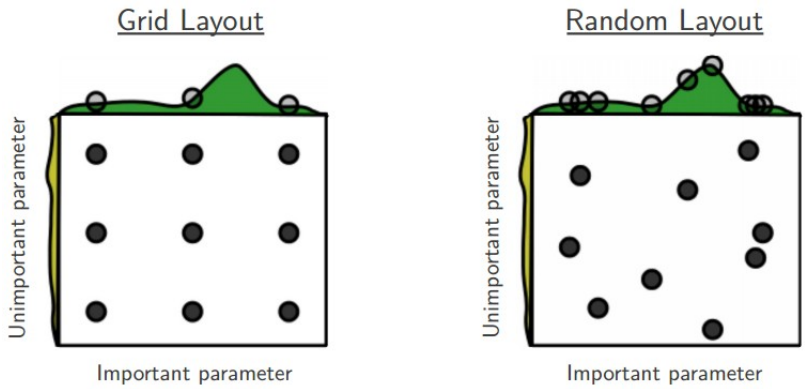


Figure 2.10: Grid and random search of nine trials to find the optimum of a function $f(x, y) = g(x) + h(y) \approx g(x)$. $g(x)$ is represented by the green curve above each graph while $h(x)$ is represented by the yellow one on the left. In this particular example, grid search only explored three different points of $g(x)$ on the nine trials while random search explored a different value on each trial. Figure from [7].

architectures have been developed. We briefly introduce here two of the most important: convolutional neural networks and recurrent neural networks.

2.9.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) [43] have been developed for processing data which has a grid-like topology. In practical applications, inputs are usually a multidimensional array of data (sometimes referred to as *tensors*) such as 2D images. Unlike traditional MLPs (which employ general matrix multiplication operations), they employ an operation called *convolution*, which is a special kind of linear transformation. Although these networks are called convolutional, in practice, neural network software libraries employ a related operation called *cross-correlation* to compute outputs.

For image data processing, the convolution operation is particularly interesting because it allows the extraction of relevant features in images by taking into account the neighboring pixels (i.e., context) for each pixel.

The learnable parameters in a CNN are arranged in multidimensional arrays called *kernels* or *filters*. Fig. 2.11 illustrates a 2D cross-correlation operation. Notice how the kernel in this particular example has 4 parameters arranged in a 2×2 matrix. The filter slides over the input tensor from left to right and from top to bottom, and at each position we compute an element-wise multiplication between the filter's parameters and the input subarray contained in that position. Finally, a *sum-pooling* operation sums the results of

these multiplications at each position to produce an output tensor, although other types of pooling operations are possible (mean-pooling, max-pooling, etc.).

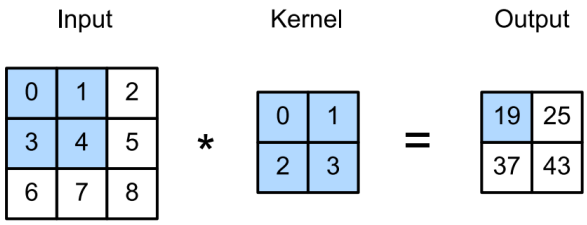


Figure 2.11: Two-dimensional convolution operation. The shaded portions are the first output element and the input and kernel array elements used in its computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$. Figure from [87].

It is interesting to note that the cross-correlation operation can decrease the spatial dimensions of the input, as in Fig. 2.11. It is possible to control the size of the output either by choosing the size of the kernel, its stride (whether it slides over the input tensor one position at a time or more, skipping intermediate locations), by padding the input tensor with zeros all around (therefore artificially increasing its spatial dimensions) or through a combination of these options. It is also possible to use multiple kernels per convolutional layer. Fig. 2.12 illustrates the VGG-16 [67] architecture, a popular CNN for image classification.

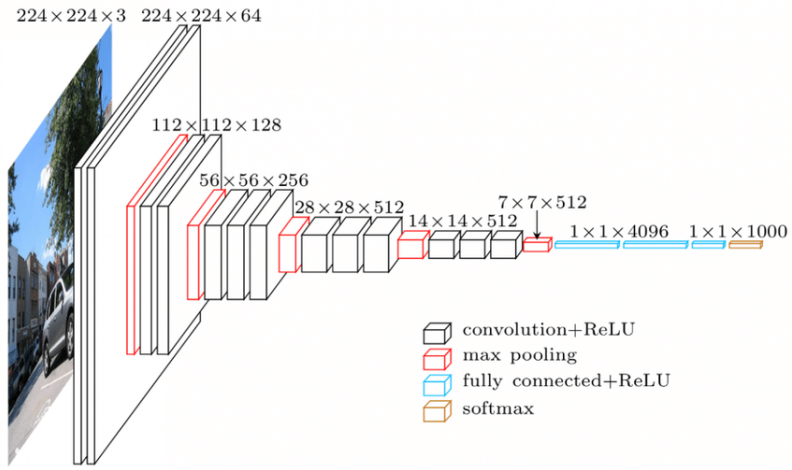


Figure 2.12: VGG-16 [67], a popular CNN architecture for image classification (1000 classes).

In practice, when training CNNs, we observe that kernels tend to learn to extract particular features in images such as contours and other properties. For example, for a CNN trained

to identify pictures of cats, we may find (through visualization) that some kernels learn to identify a cat's ears, while others may learn to identify the general body shape of a cat, etc.

Perhaps unsurprisingly, the powerful computer hardware available nowadays has allowed CNNs to become very deep, with some architectures exceeding 100 layers [28]. Naturally, this makes training such architectures and getting them to converge quite complicated. One interesting solution researchers have come up with is batch normalization [35], which uses the statistics of the minibatches during training to normalize a given layer's outputs before feeding them into the following layer. This makes models less sensitive to parameter initialization and makes tuning hyperparameters less complicated. Another interesting thing to note is that batch normalization layers can be used with other types of neural networks (such as MLPs).

Note that we have provided a very shallow introduction to CNNs. For a detailed explanation of the various concepts not discussed here, we refer the reader to the excellent books [87] and [21], which devote entire chapters to this type of neural network.

2.9.2 Recurrent Neural Networks

Another important type of neural network we need to briefly mention are recurrent neural networks (RNNs). They use feedback connections, where the model's outputs are fed back into itself, hence the term *recurrent* in the name.

While convolutional neural networks are very effective at processing spatial information, RNNs are ideally suited to handle sequential information. To compute current outputs, they use state variables which store past information (classical RNNs) or both past and future information (bidirectional RNNs), along with recurrent connections [87]. This allows RNNs to learn new sequence representations that take into account the temporal dependencies in the data. Interestingly, they are able to process data sequences of variable length without needing to increase the model size, although, for long input sequences, it may be difficult to access information from a long time ago. Fig. 2.13 illustrates a general RNN architecture, where the hidden states store the current output and feed it back for subsequent computation.

Arguably, the most famous type of RNN are long short-term memory (LSTM) [31] networks, successful applications of which include machine translation [70] and image captioning [75]. For a detailed description of this type of architecture, we refer the reader to [87].

In the next chapter, we discuss a particular neural network architecture, designed specifically for handling graph-structured data.

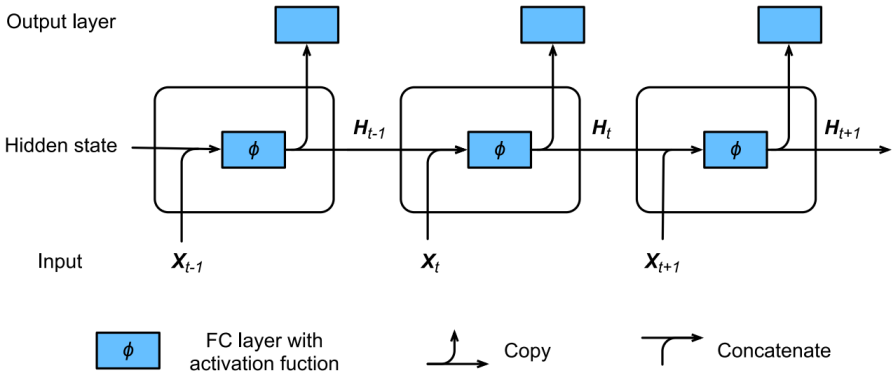


Figure 2.13: An RNN with a hidden state. Figure from [87].

CHAPTER 3

Graph Neural Networks

The advent of deep learning approaches, brought about by increasingly powerful computer hardware and their impressive results in computer vision and natural language processing tasks, has led to the emergence of various deep neural architectures for processing graph-structured data called *graph neural networks* (GNNs). Generally speaking, the goal of these GNN approaches is to learn a mapping that encodes structural information about input graphs—usually represented by their adjacency and node feature matrices—into a low-dimensional vector space, such that the structure of the original graphs is reflected in the geometric relationships in this vector space (also called *embedding* space). Depending on the task at hand (e.g. node or graph classification), new representations—or *embeddings*—are learned for either the nodes or the entire graph. These embeddings are then used instead of the original adjacency and node feature matrices, e.g. to perform classification in the case of supervised learning.

An important breakthrough in graph neural network architecture design came about in 2017, with Thomas N. Kipf & Max Welling introducing the Graph Convolutional Network (GCN) architecture [40]. GCNs are motivated by convolutional neural networks (CNNs), which are able to construct highly expressive representations from spatial features of the input data. However, as CNNs can only operate on data lying on Euclidian domains like images (2D or 3D grids) and text (1D sequences), work has been done to extend the fundamental operations of CNNs (convolution and pooling) to operate effectively on non-Euclidian data such as graphs [3, 15].

The fundamental contribution of Kipf & Welling lies in providing an efficient extension of

the convolution operation to graphs. Their formulation is motivated from a first-order approximation of spectral graph convolutions [27], with further mathematical simplifications leading to a computationally efficient, fully vectorized graph convolution operation that is also scalable to large input graphs—i.e., graphs with a large number of nodes and edges.

Since modern GNN architectures use a neighborhood aggregation scheme where, at any given layer in the network, the representation vector of a node is computed by using a recursive aggregation of that node's neighbors' representation vectors, we can use a general framework to describe these models and compare them effectively. The rest of this chapter is organised as follows. First, we provide motivations for studying GNNs in the context of control engineering by exploring two important research papers which give us a look at a few interesting applications. After that, we detail the general GNN framework described in [84]. Then, in the context of this framework, we describe and motivate the use of Graph Convolutional Networks [40] and Graph Isomorphism Networks (GINs) [84], both of which are—along with attention mechanisms—the state-of-the-art architectures most relevant to our work. Finally, we present other important, state-of-the-art research papers on graph neural networks.

3.1 Graph Neural Networks in Control Engineering

Neural networks, in general, find applications in control theory in various ways, including modeling robotic systems in policy optimization tasks in reinforcement learning [72], human pose estimation [86, 69] and action recognition [66, 59, 57].

In the case of graph neural networks more specifically, a few applications have emerged recently with very promising results:

- Learning a model of the dynamics of a robotic system (system identification).
- Model-predictive control (MPC).
- Human skeleton-based action recognition.

Although we focus on these applications, this list is non-exhaustive; many research works have used GNNs in other control and robotics-related problems [79, 80, 58, 45, 47].

In systems of articulated robots and in human skeleton image sequences, the nature of the data lends itself particularly well to a graph representation: articulated robotic systems have *structure*, which means that exclusively representing them by vectors of features (e.g., positions, linear and angular velocities) would be excluding important information about their physical make-up. Using a graph representation, where an adjacency matrix contains

structural information (with nodes representing bodies and edges representing joints) along with features would seem like a natural way to overcome this limitation. The same intuition can be applied to the human skeleton: graphs perfectly capture the kinematic dependency between the joints and bones in the human body.

This intuition provides a strong argument for using GNNs in the applications mentioned above and the experimental results outlined in the next research papers do indeed validate this argument.

3.1.1 Graph Neural Networks for Inference and Control

Sanchez-Gonzalez et al. consider in [62] a supervised learning problem in which they model different physical robotic systems as directed graphs (where nodes represent physical bodies and edges represent joints linking the bodies together, as illustrated in Fig. 3.1) and use system state data to learn a GNN dyamic model of these systems. The goal is that the learned model can then be used to make very accurate forward predictions about system states under random control signals.

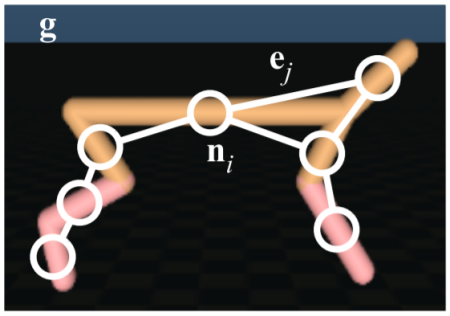


Figure 3.1: Graph representation of a physical system's bodies and joints. Figure from [62].

The basic building block (which they call Graph Network block or GN block) of the architecture they use sequentially applies 3 functions f_e, f_n and f_g to update edge, node and global feature vectors respectively (see Fig. 3.2 for an illustration). The individual functions can be implemented using standard neural networks, and the authors use MLPs in their implementation. The authors generate training data by applying simulated random control signals to their real physical systems then recording the state transitions.

Modeling the system this way, the authors achieve very accurate predictions of future states that generalize well to systems with continuously varying static parameters and to unseen (i.e. test) data. Furthermore, they use a variation of the GNN architecture in a setting where some of the system's properties are not observable (mass, joint stiffness, etc.) and find that it could still make accurate predictions, proving that the model performs *implicit*

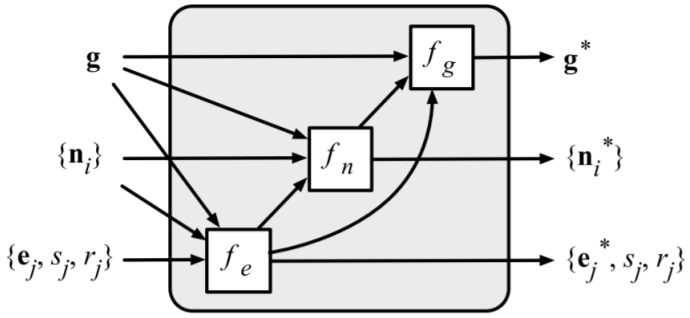


Figure 3.2: A Graph Network (GN) block. Figure from [62].

system identification—that is, these unobserved properties are not estimated explicitly, but are expressed in latent representations which are made available to other mechanisms.

They further show that model-predictive control for reference tracking using this model architecture works really well: the architecture being differentiable, they backpropagate gradients of a cost function J with respect to the control variables u and iteratively optimize the control values using gradient descent. They show that control using the learned model worked in many cases as well as control using the real physics model.

3.1.2 AGC-LSTM Model for Skeleton-Based Action Recognition

Si et al. propose in [66] a novel and general network architecture for skeleton-based action recognition called Attention Enhanced Graph Convolutional LSTM Network (AGC-LSTM), which is the first attempt at a combined Graph Convolutional Network (with attention) and LSTM [31] architecture for this computer vision task.

Human action recognition is an important task in computer vision. The goal is to correctly identify what action a human agent is doing through the adequate understanding of movement characteristics in an input sequence of images. Naturally, this requires a machine learning tool which has the intrinsic ability to operate on sequential data and this is where the LSTM (Long Short-Term Memory) part of this model comes into play. LSTMs [31] are a fundamental discovery in machine learning models; they are able to process sequential data and are at the heart of many breakthrough results in fields such as natural language processing, with successful applications in modern voice assistants and image captioning tools among others.

Effectively extracting discriminative spatial and temporal features is a challenging problem, and there have been numerous attempts at finding effective frameworks based on both RGB (red, green, blue) color video and skeleton data, with skeleton-based approaches

being inherently more robust since they do not suffer from limitations such as background clutter and illumination changes.

The proposed AGC-LSTM is able to effectively capture discriminative spatiotemporal features. More specially, the attention mechanism is employed to enhance the features of key nodes, which assists in improving spatio-temporal expressions. The proposed model achieves state-of-the-art results on two benchmark datasets for skeleton-based action recognition.

Having explained why GNNs are relevant in control and robotics applications, we present, in the next section, a general definition that encompasses many modern GNN architectures.

3.2 A Unifying Framework for Graph Neural Network Architectures

Let $G = (V, E)$ denote a graph with adjacency matrix $A \in \mathbb{N}^{n \times n}$ and a real-valued node feature matrix X whose n rows correspond to node feature vectors X_v, $v \in V$. For graph classification tasks, we are given a set of graphs $\{G_1, \ldots, G_M\} \subseteq \mathcal{G}$ and their labels $\{y_1, \ldots, y_M\} \subseteq \mathcal{Y}$, where $\mathcal{G}$ and $\mathcal{Y}$ represent the graph and label spaces, respectively. Our goal is to learn (i) a representation vector (or embedding) h_{G_i} for each graph $G_i \in \mathcal{G}$, as well as (ii) the function that maps each graph embedding h_{G_i} to the corresponding graph label y_i.

Graph neural networks use graph structure information, contained in the adjacency matrix A, and node feature vectors X_v to learn node embeddings h_v by repeatedly *aggregating* information from the neighborhood of each node. After k iterations of aggregation, where each aggregation operation corresponds to a layer in the considered GNN architecture, the resulting representation of each node is effectively capturing the structural information within the k^{th}-order neighborhood of that particular node. Following the formalism used in [84, 2], the k^{th} layer of a GNN can be characterized by the following equations:

$$a_v^{(k)} = \text{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}_v\right\}\right), \tag{3.1}$$

$$h_v^{(k)} = \text{COMBINE}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right), \tag{3.2}$$

where $h_v^{(k)}$ is the feature vector of node v at the k^{th} iteration/layer and $\mathcal{N}_v$ is the set of nodes adjacent to node v (its neighbors). AGGREGATE is the function which gathers these neighboring nodes' features while COMBINE uses a combination of the resulting aggregate and the current representation of node v to compute its new one. The input to the network being a matrix of node feature vectors X_v means that we initialize $h_v^{(0)} = X_v$.

Depending on the application, node embeddings can be used in a final readout step to compute a graph embedding—i.e. a new representation vector for the whole graph. For node classification, the node representation $h_v^{(K)}$ of the final iteration/layer K is used to predict the node's class. For graph classification, the following READOUT function aggregates node features from the final iteration to obtain the entire graph's representation h_G:

$$h_G = \text{READOUT}\left(\left\{h_v^{(K)} \mid v \in V\right\}\right). \tag{3.3}$$

READOUT can be a simple permutation-invariant function such as summation or a more sophisticated graph-level pooling function.

The choice of the AGGREGATE$^{(k)}(\cdot)$, COMBINE$^{(k)}(\cdot)$ and READOUT$(\cdot)$ functions is what dictates the global architecture of the GNN. The framework and its three main equations having been introduced, we now have a reference point from which we can describe GCNs and GINs clearly.

3.3 Graph Convolutional Networks (GCNs)

As discussed earlier, the GCN architecture's main contribution was to provide a computationally efficient and theoretically motivated implementation of convolution in the graph domain. Because of this, it has now become an important baseline against which progress in GNN architecture design is measured and a good starting point for our own work.

In [40], where GCNs were first introduced, the authors consider the semi-supervised node classification problem in a graph and present a scalable approach based on convolutional neural networks which operate on graphs.

The authors' model relies on a layer-wise propagation rule which can be motivated by spectral graph convolutions, with approximations through first-order Chebyshev polynomials allowing the model to be computationally feasible. Further simplifications reduce the number of trainable parameters (and by extension overfitting). Then, a neural network can be built by stacking multiple such layers and applying a point-wise non-linearity. To avoid exploding/vanishing gradients, the authors introduce a renormalization trick and by conditioning the model on both the feature matrix X and the adjacency matrix A gradient information from the supervised loss will be distributed and the model will be able to use information about the graph's structure. This architecture is called *convolutional* because it represents a node as a function of its surrounding neighborhood. In other words, taking the product of the node feature and adjacency matrices essentially amounts to summing neighboring node information for each node (along with information about the node itself, as long as we enforce self-loops). By applying successive convolution operators, we are effectively convolving the kth-order neighborhood of the nodes.

The full architecture the authors use (for node classification, not graph classification!) is described in the paper by:

$$Z = f(X, A) = \text{softmax}\left(\hat{A}\ \text{ReLU}\left(\hat{A}XW^{(0)}\right)W^{(1)}\right), \tag{3.4}$$

which describes a two-layer GCN where $\hat{A}$ is the normalized adjacency matrix of the input graph (with added self-loops), $W^{(k)}$ is the k^{th} layer's trainable weight matrix and X is a matrix of node features. Here, $W^{(0)} \in \mathbb{R}^{C\times H}$ is an input-to-hidden weight matrix for a hidden layer with H feature maps and $W^{(1)} \in \mathbb{R}^{H\times F}$ is a hidden-to-output weight matrix. C is the dimension of the input features X_v for $v \in V$, H is the dimension of the hidden layer (this is a hyperparameter whose value we can choose, as we will see later in training) and F is our output dimension (equal to the number of classes in our classification problem). Then, the softmax function is applied. Consequently, we end up with a matrix Z, where each row (corresponding to a node in the input graph) contains the estimated conditional probabilities of each class.

Looking at it from the perspective of the framework described above, the AGGREGATE and COMBINE steps defined in Eq. (3.1) and (3.2) are integrated as follows:

$$h_v^{(k)} = f^{(k-1)}\left(W^{(k-1)}\cdot\text{MEAN}\left\{h_u^{(k-1)}, \forall u \in \mathcal{N}_v \cup \{v\}\right\}\right), \tag{3.5}$$

where $\text{MEAN}\left\{h_u^{(k-1)}, \forall u \in \mathcal{N}_v \cup \{v\}\right\}$ represents the multiplication by the normalized adjacency matrix $\hat{A}$ in Eq. (3.4) (element-wise *mean* pooling), $k \in \{1, 2\}$, $h_v^{(0)} = X_v$, $v \in V$ for a graph $G = (V, E)$, $f^{(0)} = \text{ReLU}$, $f^{(1)} = \text{softmax}$ and Z in Eq. (3.4) is the matrix whose rows are the vectors $h_v^{(2)}$, $v \in V$.

The authors used the GCN for node classification on the Citeseer, Cora and Pubmed citation network datasets and on the NELL knowledge graph where only a fraction of nodes are labeled. It significantly outperformed recent related methods and the propagation model they use offers both improved efficiency (fewer parameters and operations) and better predictive performance compared to a naïve 1^{st}-order model or higher-order graph convolutional models using Chebyshev polynomials.

3.4 Graph Isomorphism Networks (GINs)

Although recent GNNs have achieved stat-of-the-art performance in many tasks such as node classification, link prediction and graph classification, their designs are mainly based on empirical intuition, heuristics and experimental trial-and-error, with little theoretical work undertaken to understand their properties and limitations.

Xu et al.'s work in [84] represents another important milestone in GNN architecture design. They provide—and mathematically prove—conditions for a GNN model to be as powerful as the Weisfeiler-Lehman (WL) [81] graph isomorphism test, then proceed to introduce their own model, the Graph Isomorphism Newtork (GIN), which fulfills these requirements. Beyond relying on intuition, their architecture is guaranteed to be maximally expressive: it is proven to map two different (i.e., non-isomorphic) graphs to two different graph embeddings. Because of this, their state-of-the-art GIN is one of the most important GNNs currently available and makes for an excellent high-performance backbone on which we can build and improve.

First, the authors provide a theoretical framework for analyzing the expressive power of GNN architectures. Then, they introduce a novel architecture which is provably as expressive as the WL graph isomorphism test.

Intuitively, the authors theorize that a maximally powerful GNN must be able to map two nodes to different embeddings, except if they have identical subtree structures (i.e., neighborhood structures) with identical features on the corresponding nodes.

The authors introduce (and prove) a series of lemmas and theorems which—when satisfied by a given GNN architecture—allow it to be as powerful as the WL test and thus achieve maximum discriminative power. More explicitly, any GNN can be maximally powerful if, for the following node embedding function:

$$h_v^{(k)} = \phi\left(h_v^{(k-1)}, f\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}_v\right\}\right)\right), \tag{3.6}$$

the functions f, ϕ and the graph-level readout—which operates on the multiset of node features $\left\{h_v^{(K)}\right\}$—are injective. We refer the reader to [84] for proofs and further details.

Having established conditions for a maximally powerful class of GNNs, the authors then developed a simple architecture, *Graph Isomorphism Network (GIN)*, using multi-layer perceptrons (MLPs) to model and learn injective aggregation functions. MLPs are able to do so because of the universal approximation theorem [32], which is a property that allows the GIN to approximate a theoretical GNN, leading to the following equation:

$$h_v^{(k)} = \mathrm{MLP}^{(k)}\left(\left(1 + \epsilon^{(k)}\right) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}_v} h_u^{(k-1)}\right). \tag{3.7}$$

which merges the AGGREGATE (3.1) and COMBINE (3.2) steps into a single equation.

For graph-level readout, the authors use information from the learned representations at all levels of the GIN: they sum the node features from the same iterations/layers (sum aggregation is injective) and then concatenate these resulting vectors together into a final

graph-level representation h_G (this guarantees that information from all layers is used for the final prediction):

$$h_G = \text{CONCAT}\left(\sum_{v \in V} h_v^{(k)} \middle| k = 0, 1, ..., K\right). \tag{3.8}$$

Next, a discussion of GNNs that do not satisfy the conditions ensues (GCN [40], GraphSAGE [25]), with explanations of why 1-layer perceptrons are insufficient (they may not be powerful enough to be able to distinguish different multiset functions—i.e., functions which operate on sets with possibly repeating elements) and the limitations of mean and max-pooling (they are not injective, hence their limited representational power).

The authors tested the model and compared it to other variants of GNNs on a variety of bioinformatics and social network datasets. Training set performance gives an idea of a model's representational power while test set performance quantifies generalization ability. The GIN outperformed the less powerful GNN variants on every dataset used and these tests highlighted the advantages of sum-aggregation GNNs over mean-aggregation models.

Now that we have motivated our choice of GCNs and GINs as baselines and taken a detailed look at their achitectures, we provide in the next section an overview of the current state of the art in graph neural network research by looking at a list of important papers.

3.5 Other Important Work on Graph Neural Networks

3.5.1 Representation Learning on Graphs

Hamilton et al. provide in [26] a review of the key advancements in the field of representation learning on graphs, particularly methods to embed individual nodes as well as entire (sub)graphs.

Node embedding methods fall into three main categories: matrix factorization-based methods [6, 1, 61, 46], random walk-based algorithms [22, 56] and graph neural networks.

The authors, in their discussion of the various methods, adopt a unified framework to explicitly structure methodological diversity and unify notations and key concepts.

This framework is structured around four components: a pairwise similarity function to measure similarity between nodes, an encoder function that generates node embeddings (and contains trainable parameters), a decoder function which reconstructs similarity values between nodes, and a loss function.

Intuitively, matrix factorization methods try to learn embeddings for each node such that the inner product between learned embeddings approximates some deterministic measure of node similarity. By contrast, random walk approaches employ a flexible, stochastic measure of node similarity, where nodes have similar embeddings if they tend to co-occur on short random walks over the graph.

However, these two families of methods (shallow embedding methods) train unique embedding vectors for each node independently, which leads to some drawbacks:

- No parameter sharing between nodes: computationally inefficient, loss of a powerful form of regularization.
- Failure to leverage node attributes during encoding.
- Inherently transductive [25]: cannot generate embeddings for unseen nodes.

Neighborhood autoencoder methods [11, 77] solve the second issue by incorporating graph structure using deep neural networks. They rely on autoencoders [30] to compress information about a node's local neighborhood. They still suffer from the other drawbacks.

Neighborhood aggregation/convolutional encoders solve the limitations of shallow embeddings and autoencoders by incorporating graph structure into the encoder, leveraging node attributes, sharing parameters across nodes and they can generate embeddings for nodes that were not present during training. The intuition behind them is that they generate embeddings for a node by aggregating information from its local neighborhood. This family includes graph convolutional networks (GCN [40]), column networks [71] and GraphSAGE [25]. It is common for these methods to incorporate supervision from node classification tasks in order to learn embeddings.

Applications of node embeddings include visualization and pattern discovery, clustering and community detection, node classification and semi-supervised learning and link prediction. What's more, sets of node embeddings can be used to generate embeddings of entire subgraphs. Methods to do this include sum-based and graph-coarsening approaches. Subgraph embeddings can be used for classification or the prediction of various properties.

3.5.2 A Spectral Formulation of Convolutional Neural Networks on Graphs

Defferard et al. introduce in [15] the mathematical and computational foundations of an efficient generalization of CNNs to graph data by extending the convolution and pooling operations to operate effectively on data lying on non-Euclidian domains.

Their work relies on graph signal processing (GSP) tools to end up with an efficient spectral graph formulation of CNNs on graphs. Their work shows that this framework is able to produce localized graph filters which are efficient to evaluate and learn while having the ability to extract relevant information from graph-structured data.

3.5.3 Learning Neural Fingerprints of Molecular Data

Duvenaud et al. introduce in [17] a GNN architecture for learning features ("fingerprint" vectors, which are essentially encodings of different substructures of a molecule as a mathematical object) for molecular data represented with graph structures.

However, these fingerprints are non-differentiable and, in the case of circular fingerprints, are no better at modeling chemical features than randomly initialized neural graph fingerprints. To overcome these problems, the authors use a differentiable neural network architecture whose lower layers are convolutional.

This neural graph method for computing differentiable fingerprints provides several advantages over fixed fingerprints:

- Ability to encode only relevant features (fixed fingerprints on the other hand must be extremely large to encode all possible substructures without overlap) therefore reducing computational cost.
- Better predictive performance thanks to machine-optimized fingerprints through the use of data relevant to the task at hand.
- More meaningful feature representations: similar but distinct molecular fragments can activate the same neural graph fingerprint, as opposed to standard fingerprints which encode each fragment completely distinctly.

By using a differentiable architecture, the authors show it is possible to use standard neural-network training methods to scalably optimize the parameters of these neural molecular fingerprints end-to-end and achieve better predictive performance.

3.5.4 Diffusion-Convolutional Neural Networks (DCNNs)

Atwood & Towsley introduce in [3] a neural network architecture for graph-structured data by extending convolutional neural networks to general graph-structured data. To accomplish this, the authors introduced a "diffusion-convolution" operation which builds a latent representation by scanning a diffusion process across each node in a graph-structured

input. Graph diffusion can be represented as the transition matrix (a matrix that gives the probability of jumping from node i to node j in one step) power series.

This architecture offers several advantages over probabilistic relational models and kernel methods[1]:

- DCNNs are significantly more accurate in node classification tasks compared to alternative methods.
- They provide a flexible representation of graphical data which can be used for a variety of classification tasks.
- Prediction from a DCNN can be expressed as a series of polynomial-time tensor operations which is very efficient.

3.5.5 Patchy-san (PSCN)

Niepert et al. introduce in [51] a framework to apply CNNs to bear on a large class of graph-based learning problems. Similar to CNNs for images, they devise a scheme to construct locally connected neighborhoods from input graphs (that is, leveraging graph structure to map nodes and their neighborhood from a graph representation to a vector space representation) which then serve as the receptive fields of a convolutional architecture, allowing the framework to learn effective graph representations.

This approach, which they call Patchy-san, proved to be highly competitive at tasks such as graph classification against state-of-the-art graph kernel approaches at the time while being computationally efficient, and it would provide the basis for later work on graph neural networks, which represent the current state of the art on graph data-based classification tasks.

3.5.6 Spectral and Locally Connected Networks on Graphs

Bruna et al. discuss in [10] ways to construct deep neural networks on graphs other than regular grids. Specifically, they propose two different constructions: a spatial construction, which extends the properties of subsampling and compactly supported filters to general

[1] Graph kernels work by employing a pairwise similarity measure between graphs in a given set of data. A popular approach to graph classification with graph kernels is to use a graph kernel to compute an $n \times n$ kernel matrix K where K_{ij} represents the similarity between graphs $\mathcal{G}_i$ and $\mathcal{G}_j$, and then to plug the computed kernel matrix into a kernalized learning algorithm such as support vector machines (SVM) to perform classification.
For more information on graph kernel methods, we refer the reader to [8] and [85].

graphs and a spectral construction, which draws on properties of convolutions in the Fourier domain to enable a construction where the number of learnable parameters is independent of the input dimension.

These constructions allow efficient forward propagation and can be applied to datasets with a very large number of features. Their main contribution consists in showing that it is possible to obtain efficient architectures from a weak geometric structure, which they validate on low-dimensional graph datasets.

3.5.7 Message Passing Neural Networks (MPNNs)

Gilmer et al. reformulate in [20] existing neural network approaches to graph data processing into a single common framework they call Message Passing Neural Networks (MPNNs) and explore additional novel variations within this framework.

Their results show that MPNNs with the appropriate message, update, and output functions chosen from existing state-of-the-art GNNs have a useful inductive bias for predicting molecular properties, outperforming several strong baselines and eliminating the need for complicated feature engineering.

3.5.8 FastGCN

Chen et al. propose in [12] a modification to the original GCN architecture [40] for learning graph embeddings called FastGCN, whose objectives are:

- To relax the requirement of simultaneous availability of test data: for example, constantly evolving graphs require an inductive scheme that learns a model from only a training set of vertices and that generalizes well to any augmentation of the graph.
- To mitigate time and memory impacts of the recursive neighborhood expansion across layers.

To achieve these goals, they interpret graph convolutions as integral transforms of embedding functions under probability measures and they use a sampling scheme in the reformulation of the loss and the gradient. The proposed approach not only gets rid of the reliance on the test data but also yields a controllable cost for per-batch computation while retaining comparable prediction accuracy.

3.5.9 Simple Graph Convolution (SGC)

Wu et al. theorize in [83] that graph convolutional networks (GCNs) and their variants may be inheriting unnecessary complexity and redundant computation by virtue of deriving inspiration from recent deep learning approaches, which have gained traction due to the limitations of prior approaches. Consequently, in this paper, the authors worked to reduce this complexity through successively removing nonlinearities and collapsing weight matrices between consecutive layers.

Notably, their experimental evaluation demonstrates that these simplifications do not negatively impact accuracy in many downstream applications such as node classification. Moreover, the resulting model—which they call Simple Graph Convolution (SGC)—scales to larger datasets, is naturally interpretable, and yields up to two orders of magnitude speedup over methods such as FastGCN in a variety of benchmark datasets.

3.5.10 UGraphEmb

Bai et al. introduce in [5] an end-to-end neural network-based framework for graph-level representation learning (called UGraphEmb), in which they adopt the GIN state-of-the-art architecture [84] as a node embedding method (although any architecture can be used under such a framework). To generate their graph-level embedding, they propose an attention mechanism called Multi-Scale Node Attention (MSNA).

Experiments on various benchmark datasets show that the produced graph-level embeddings achieve very competitive performance on three downstream tasks, namely, graph classification, similarity ranking, and graph visualization.

CHAPTER 4

Attention Mechanisms

We introduce here attention mechanisms, an important concept we use to augment our baseline GCN [40] and GIN [84] architectures and make them even more powerful.

An attention mechanism is a component of a neural network's architecture which can help improve a model's performance by allowing it to focus on important parts of the input data to make decisions. In other words, attention is in charge of quantifying the dependence between the input and output elements.

Attention mechanisms have been successfully adopted by models solving different tasks, mainly in computer vision and natural language processing, the latter being the breakthrough application domain. Specifically, these mechanisms have been used for machine translation [4] to allow the model to focus on the relevant parts of the input sentence by assigning weights which reflect the relative importance of different words (in the input sentence) to each translated word (output sentence). To better illustrate an attention mechanism's benefits, let us consider an example of English-to-French sentence translation from [4]. The source sentence is:

> *An admitting privilege is the right of a doctor to admit a patient to a hospital or a medical centre to carry out a diagnosis or a procedure, based on his status as a health care worker at a hospital.*

A first translation is done without "attention" and hence it is not so accurate since it cannot be context-aware:

> *Un privilège d'admission est le droit d'un médecin de reconnaître un patient à l'hôpital ou un centre médical d'un diagnostic ou de prendre un diagnostic en fonction de son état de santé.*

Notice how the translation deviates from the original meaning of the source sentence (underlined text). The second translation, on the other hand, takes into account context (through "attention"), which leads to a much better and natural translation:

> *Un privilège d'admission est le droit d'un médecin d'admettre un patient à un hôpital ou un centre médical pour effectuer un diagnostic ou une procédure, selon son statut de travailleur des soins de santé à l'hôpital.*

Attention mechanisms have also been used to select important regions of an input image for image classification [76]. Finally, attention has also been used in image [90] and natural language [42] question answering.

More recently, there has been more interest in applying attention mechanisms to graph-related tasks (e.g, graph classification [50], node classification [74, 19, 88] and link prediction [23, 78]) where they are used to compute node embeddings that focus on the more relevant neighbors of a given node. Attention mechanisms offer several benefits:

- Since attention is essentially a weighted average, it allows for dealing with variable-sized inputs [74] without predefining the input size (in GNNs, attention can be used on nodes with different numbers of neighbors), allowing the model to focus on task-relevant parts of the graph.
- They allow the model to ignore noisy and task-irrelevant parts of the graph, improving the signal-to-noise (SNR) ratio [37, 76].
- They make a model's results more interpretable [74, 4] by analyzing the learned attention weights.

The authors of [44] formally define graph attention as follows:

Definition 4.1. (Graph attention) Given a target graph object v_0 (e.g., node, edge, graph) and a set $\{v_1, \ldots, v_{|\mathcal{N}_{v_0}|}\}$ of graph objects in v_0's neighborhood $\mathcal{N}_{v_0}$, attention is defined as a function $f_a : \{v_0\} \times \mathcal{N}_{v_0} \to [0, 1]$ that maps each of the objects in $\mathcal{N}_{v_0}$ to a relevance score that indicates the importance of each object in $\mathcal{N}_{v_0}$ to v_0. Furthermore, to make scores easily comparable across different objects, we normalize them across all choices of v_j in $\mathcal{N}_{v_0}$, that is, $\sum_{j=1}^{|\mathcal{N}_{v_0}|} f_a(v_0, v_j) = 1$.

In a neural network, these scores will be learned and, in the particular case of GNNs, they will be used to compute a hopefully better node embedding that's the weighted sum of the features of its neighbors; instead of attributing the same weight to each neighbor.

4.1 Types of Graph Attention Mechanisms

Attention mechanisms that have been applied to graphs can be classified into three main types (as described in [44]). They all share the same purpose, that is, they are used to help the model focus on relevant parts of a graph. However, these types differ in how the attention mechanism is defined or implemented, with the difference usually being in the attention function f_a (see Definition 4.1). We describe them here.

4.1.1 Velickovic et al.'s Attention

The first type is the attention mechanism described in Graph Attention Networks (GATs) [74]. This particular type of attention is called *self-attention*—that is, it quantifies the *inter-dependence* between the input elements. This is different from general attention, which quantifies the dependence between the input and output elements (as mentioned at the beginning of this chapter as a general definition of attention).

In that paper, Velickovic et al. introduced a GNN model based purely on attention (implemented as a two-layer network of what they call *graph attentional layers*) and managed to best the GCN and other competing methods in classification accuracy across various datasets. This suggests that their attention mechanism is very powerful, and that by mixing these attention layers with GCN or GIN layers, we might be able to achieve even better performance than standalone GCNs, GINs or GATs. We use a slightly modified version of this attention mechanism to augment our baseline GCN and GIN architectures, the details of which are discussed in Chapter 5.

A graph attentional layer in the GAT accepts as input a set of node embeddings $\{h_1, h_2, ..., h_n\}$, $h_i \in \mathbb{R}^F$ corresponding to the nodes $\{v_1, v_2, ..., v_n\}$, where n is the number of nodes and F is the dimension of each embedding. It outputs a new set of embeddings (of a potentially different dimension F'), $\{h'_1, h'_2, ..., h'_n\}$, $h'_i \in \mathbb{R}^{F'}$.

As a first step, a linear transformation, parameterized by $\mathbf{W}$, maps node embeddings from $\mathbb{R}^F$ to $\mathbb{R}^{F'}$. Then, an attention mechanism $f_a : \mathbb{R}^{F'} \times \mathbb{R}^{F'} \to \mathbb{R}$ is applied as follows:

$$e_{i,j} = f_a(\mathbf{W}h_i, \mathbf{W}h_j), v_j \in \mathcal{N}_{v_i}, \tag{4.1}$$

$$e_{i,j} = 0, v_j \notin \mathcal{N}_{v_i}, \tag{4.2}$$

with $\mathcal{N}_{v_i}$ being the neighborhood of node v_i. The computed coefficients $e_{i,j}$ represent the relative importance of node v_j to node v_i and from the two equations above, attention is only performed (i.e., non-zero) on a node's neighbors.[1] To make coefficients easily

[1]The equations as written in the original GAT paper [74] (and here) may lead the reader to think that attention scores are only computed for a node's neighbors. However, the authors explicitly say that, in

comparable across different nodes, we normalize them across all choices of nodes $v_j \in \mathcal{N}_{v_i}$ using a *masked* softmax function:

$$\alpha_{i,j} = \text{softmax}(e_{i,j}) = \frac{\exp(e_{i,j})}{\sum_{v_k \in \mathcal{N}_{v_i}} \exp(e_{i,k})}, v_j \in \mathcal{N}_{v_i}, \tag{4.3}$$

$$\alpha_{i,j} = 0, v_j \notin \mathcal{N}_{v_i}. \tag{4.4}$$

The attention mechanism f_a is a single-layer feed-forward neural network, parameterized by a trainable weight vector a, then followed by the LeakyRelu nonlinearity. Eq. (4.3) then becomes:

$$\alpha_{i,j} = \frac{\exp\left(\text{LeakyReLU}(a[\mathbf{W}h_i \mid\mid \mathbf{W}h_j])\right)}{\sum_{v_k \in \mathcal{N}_{v_i}} \exp\left(\text{LeakyReLU}(a[\mathbf{W}h_i \mid\mid \mathbf{W}h_k])\right)}, v_j \in \mathcal{N}_{v_i}, \tag{4.5}$$

where $\mid\mid$ represents the concatenation operation. Once obtained, the normalized attention coefficients are used to compute a linear combination of the features corresponding to them—which is effectively a *weighted* graph convolution operation—to serve as the final output features for every node (after potentially applying a nonlinearity, σ):

$$h'_i = \sigma\left(\sum_j \alpha_{i,j} \mathbf{W}h_j\right). \tag{4.6}$$

This is the particular weighted convolution used by the authors. In practice, however, the computed attention weights can be used along a different convolution operator.

The authors have also found it useful to employ *multi-head attention*. This means that K independent attention mechanisms execute the transformation of Eq. (4.6), and then their features are concatenated, resulting in the following output feature representation:

$$h'_i = \Big\Vert_{k=1}^{K} \sigma\left(\sum_j \alpha_{i,j}^k \mathbf{W}^k h_j\right), \tag{4.7}$$

where $\Vert$ represents concatenation, $\alpha_{i,j}^k$ are normalized attention coefficients computed by the k^{th} attention mechanism (a^k), and $\mathbf{W}^k$ is the corresponding input linear transformation's weight matrix. Note that, in this setting, the final returned output will consist of $K \times F'$ features (rather than F') for each node. If multi-head attention is performed at the prediction level, the authors employ *averaging* instead of concatenation, and delay applying the final nonlinearity until then:

$$h'_i = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_j \alpha_{i,j}^k \mathbf{W}^k h_j\right). \tag{4.8}$$

their experiments, the scores are also computed for the node itself. In Section 5.1 , where we propose our own attention mechanism, we modify the notation to reflect the fact that we also compute attention scores for the nodes themselves.

4.1.2 Similarity-Based Attention

This type of attention is fairly similar to the previous one, the main difference being the use of the cosine-similarity ($\text{sim}(\vec{a}, \vec{b}) = \frac{\vec{a} \cdot \vec{b}}{||\vec{a}|| \cdot ||\vec{b}||}$) to give more attention to objects that share more similar features. This attention model was used in Attention-based Graph Neural Networks (AGNNs) [41], where attention weights are computed as follows:

$$\alpha_{i,j} = \frac{\exp\left(\beta \cdot \text{sim}(\mathbf{W}h_i, \mathbf{W}h_j)\right)}{\sum_{v_k \in \mathcal{N}_{v_i}} \exp\left(\beta \cdot \text{sim}([\mathbf{W}h_i, \mathbf{W}h_k])\right)}, \tag{4.9}$$

where sim is the cosine-similarity function and $\mathbf{W}$ is a trainable weight matrix. This model explicitly learns similar embeddings for objects that are relevant to each other.

4.1.3 Attention-Guided Walk

The third type of graph attention is slightly different in purpose. This type of attention was used in Graph Attention Models (GAMs) [37]. In this network, we have an agent that starts at a random node on the graph and, at each time step, moves to a neighboring node. The agent can only gather information from the nodes it chooses to explore; that means that the agent needs to collect enough information to allow it to make a correct prediction on the label of the graph. The agent will only explore a small portion of the graph which means that global information about the graph is unavailable.

The attention mechanism, in this architecture, acts as a guide for the agent and is defined as a function $f_a : \mathbb{R}^h \to \mathbb{R}^k$ which takes as input the hidden state at time t, $h_t \in \mathbb{R}^h$ (h_t contains information from all the nodes previously explored) and outputs a k-dimensional vector, r_{t+1}, that ranks the k nodes of the graph according to which should be visited in the next step.

CHAPTER 5

Proposed Architectures

Now that all the essential building blocks to our work—namely GCNs, GINs and attention layers—have been explained in detail, we introduce in this chapter our proposed architectures.

As mentioned briefly in Subsection 4.1.1, our main contribution lies in augmenting the GCN [40] and GIN [84] architectures with a modified variant of the attention mechanism proposed in [74]. We call the two resulting architectures *Graph Convolutional Network with Attention* (GCNA) and *Graph Isomorphism Network with Attention* (GINA), respectively. In the next section, we describe our own implementation of the attention mechanism before detailing the GCNA and GINA architectures.

5.1 Proposed Attention Mechanism

We augment the GCN and GIN architectures with the attention mechanism presented in [74], and defined in Subsection 4.1.1, to which we make a couple of simplifications due to computational complexity considerations. More precisely, we use neither the linear transformation (characterized by the weight matrix $\mathbf{W}$ and described in Eq. (4.5)) nor multi-head attention. These changes mean that Equations (4.7) and (4.8) are unused while Eq. (4.5) simplifies to:

$$\alpha_{i,j} = \frac{\exp\left(\text{LeakyReLU}(a[h_i \mid\mid h_j])\right)}{\sum_{v_k \in \mathcal{N}^*_{v_i}} \exp\left(\text{LeakyReLU}(a[h_i \mid\mid h_k])\right)}, v_j \in \mathcal{N}^*_{v_i}, \tag{5.1}$$

where we compute attention scores for a node's neighbors and the node itself (i.e., $\mathcal{N}^*_{v_i} = \mathcal{N}_{v_i} \cup \{v_i\}$) and, for any $h_i \in \mathbb{R}^F$, let us recall that $a \in \mathbb{R}^{2F}$ is the trainable weight vector of the attention mechanism (which is a single-layer feed-forward neural network).

Once we compute attention coefficients $\alpha_{i,j}$ (refer to Subsection 4.1.1 for more details), we use them to perform a *weighted* graph convolution, where the neighbors of a node, v_i, contribute proportionally to their attention coefficients, $\alpha_{i,j}$, in the new node embedding of v_i. Implementing this weighted graph convolution in a GCN or a GIN is fairly straightforward. In fact, all we need to do is create a new, weighted adjacency matrix A_α by multiplying non-zero entries in the adjacency matrix with added self-loops by the corresponding $\alpha_{i,j}$ coefficients, as illustrated in the example below.

Example 5.1. Let A be an adjacency matrix. The weighted adjacency matrix, A_α, used for graph convolution after self-loops are added and attention is applied is as follows.

$$\underset{A}{\begin{bmatrix} 0 & 1 & 0 & \dots & 1 \\ 1 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 1 & \dots & 0 \end{bmatrix}} \rightarrow \underset{A \text{ w/ self-loops}}{\begin{bmatrix} \mathbf{1} & 1 & 0 & \dots & 1 \\ 1 & \mathbf{1} & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 1 & \dots & \mathbf{1} \end{bmatrix}} \xrightarrow{\text{attention}} \underset{A_\alpha}{\begin{bmatrix} \alpha_{1,1} & \alpha_{1,2} & 0 & \dots & \alpha_{1,n} \\ \alpha_{2,1} & \alpha_{2,2} & \alpha_{2,3} & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \alpha_{n,1} & 0 & \alpha_{n,3} & \dots & \alpha_{n,n} \end{bmatrix}}$$

where n is the number of nodes of our input graph.

This not only preserves the structure of the input graph (as null elements in the original adjacency matrix A are preserved in A_α), but also provides us with an efficient, vectorized way to update node embeddings in our GCNA and GINA architectures, the details of which are introduced in the following sections.

5.2 Graph Convolutional Network with Attention (GCNA)

First, let us recall the two (equivalent) fundamental embedding equations of the GCN architecture detailed in Section 3.3:

$$Z = f(X, A) = \text{softmax}\left(\hat{A}\ \text{ReLU}\left(\hat{A}XW^{(0)}\right)W^{(1)}\right), \tag{3.4}$$

$$h_i^{(l)} = f^{(l-1)}\left(W^{(l-1)} \cdot \text{MEAN}\left\{h_j^{(l-1)}, \forall v_j \in \mathcal{N}_{v_i} \cup \{v_i\}\right\}\right), \tag{3.5}$$

where $l \in \{1, 2\}$, $h_i^{(0)} = X_i$, $v_i \in V$ for a graph $G = (V, E)$, $f^{(0)} = \text{ReLU}$, $f^{(1)} = \text{softmax}$ and Z in Eq. (3.4) is the matrix whose rows are the vectors $h_i^{(2)}$, $v_i \in V$.[1]

[1]Note that, unlike in Section 3.3, we use l to index layers instead of k. This is done in order to avoid notational confusion in the next (GCNA) equations.

Our GCNA architecture is the result of applying our attention mechanism (see Section 5.1) on the two layers, $l \in \{1, 2\}$, of the GCN. For each layer, the first step is to compute the attention coefficients $\alpha_{i,j}^{(l-1)}$ using the attention mechanism, then update node embeddings. The whole end-to-end architecture can be described with:

$$\alpha_{i,j}^{(l-1)} = \frac{\exp\left(\text{LeakyReLU}(a^{(l-1)}[h_i^{(l-1)} \mid\mid h_j^{(l-1)}])\right)}{\sum_{v_k \in \mathcal{N}_{v_i}^*} \exp\left(\text{LeakyReLU}(a^{(l-1)}[h_i^{(l-1)} \mid\mid h_k^{(l-1)}])\right)}, v_j \in \mathcal{N}_{v_i}^*, \tag{5.2}$$

$$h_i^{(l)} = f^{(l-1)}\left(W^{(l-1)} \cdot \text{SUM}\left\{\alpha_{i,j}^{(l-1)} h_j^{(l-1)}, \forall v_j \in \mathcal{N}_{v_i}^*\right\}\right), \tag{5.3}$$

where $l \in \{1, 2\}$, $h_i^{(0)} = X_i$, $v_i \in V$ for a graph $G = (V, E)$, $f^{(0)} = \text{ReLU}$, $f^{(1)} = \text{softmax}$. Note that the MEAN has been replaced with the SUM in Eq. (5.3). Indeed, the weighted arithmetic mean, when the weights sum up to 1 ($\sum_{v_j \in \mathcal{N}_{v_i}^*} \alpha_{i,j}^{(l-1)} = 1$), is equivalent to the weighted sum.

Applied in the listed order, these equations represent our two-layer GCNA architecture. It is interesting to note that Eq. (3.4) can be rewritten in terms of two new, weighted adjacency matrices $A_\alpha^{(0)}$ and $A_\alpha^{(1)}$ whose elements are $\alpha_{i,j}^{(0)}$ and $\alpha_{i,j}^{(1)}$, respectively:[2]

$$Z = f(X, A) = \text{softmax}\left(A_\alpha^{(1)} \text{ ReLU}\left(A_\alpha^{(0)} X W^{(0)}\right) W^{(1)}\right). \tag{5.4}$$

The original GCN does not include a READOUT function (see Eq. (3.3)) since the authors do not use it for graph classification. However, in our implementation of the GCN and GCNA architectures for the graph classification task, we sum the final-layer representations $h_i^{(2)}$ of all the nodes of the input graph to obtain our graph-level readout h_G:

$$h_G = \sum_{v_i \in V} h_i^{(2)}. \tag{5.5}$$

Then, we pass h_G as input to a softmax function for classification:

$$y_G = \text{softmax}(h_G), \tag{5.6}$$

where the position of the highest score in y_G indicates the predicted class of the input graph. For a visual illustration of the GCNA architecture, see Fig. 5.1.

[2]It is unnecessary to use the normalization mentioned in Section 3.3 and in the original GCN paper [40] since normalized $\hat{A}_\alpha = A_\alpha$.

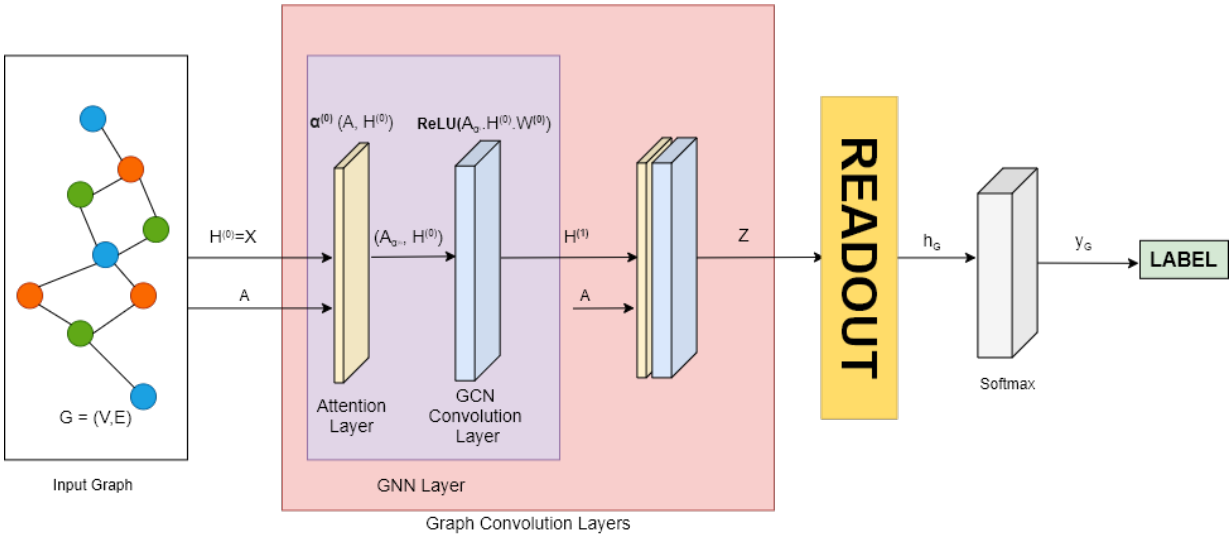


Figure 5.1: GCNA architecture. Note that $H^{(l)}$ is the matrix of the l^{th}-layer node embeddings $h^{(l)}$.

5.3 Graph Isomorphism Network with Attention (GINA)

We propose here our second architecture: GINA. In the same way, let us recall the fundamental GIN embedding equation detailed in Section 3.4:

$$h_i^{(l)} = \text{MLP}^{(l)}\left((1+\epsilon^{(l)})\cdot h_i^{(l-1)} + \sum_{v_j \in \mathcal{N}_{v_i}} h_j^{(l-1)}\right), \tag{3.7}$$

where $l \in \{1, 2, \ldots, L\}$ (L being a hyperparameter indicating the number of layers), $h_i^{(0)} = X_i$, $v_i \in V$ for a graph $G = (V, E)$.

Similarly to our proposed GCNA architecture, GINA is the result of applying our attention mechanism (see Section 5.1) on each layer of the GIN. For each layer, the first step is to compute the attention coefficients $\alpha_{i,j}^{(l-1)}$ using the attention mechanism, then update node embeddings. The whole end-to-end architecture can be described with:

$$\alpha_{i,j}^{(l-1)} = \frac{\exp\left(\text{LeakyReLU}(a^{(l-1)}[h_i^{(l-1)} \,||\, h_j^{(l-1)}])\right)}{\sum_{v_k \in \mathcal{N}_{v_i}^*} \exp\left(\text{LeakyReLU}(a^{(l-1)}[h_i^{(l-1)} \,||\, h_k^{(l-1)}])\right)}, v_j \in \mathcal{N}_{v_i}^*, \tag{5.7}$$

$$h_i^{(l)} = \text{MLP}^{(l)}\left((1+\epsilon^{(l)})\cdot \alpha_{i,i}^{(l-1)} h_i^{(l-1)} + \sum_{v_j \in \mathcal{N}_{v_i}} \alpha_{i,j}^{(l-1)} h_j^{(l-1)}\right), \tag{5.8}$$

where $l \in \{1, 2, \ldots, L\}$ (L being a hyperparameter indicating the number of layers), $h_i^{(0)} = X_i$, $v_i \in V$ for a graph $G = (V, E)$. Note that the convolution in Eq. (3.7) becomes

weighted in Eq. (5.8) thanks to the attention scores. Furthermore, in our experiments, we set $\epsilon = 0$ as we have found in practice that it adds computational complexity with no performance benefit. This means that Eq. (5.8) simplifies to:

$$h_i^{(l)} = \mathrm{MLP}^{(l)} \left(\sum_{v_j \in \mathcal{N}_{v_i}^*} \alpha_{i,j}^{(l-1)} h_j^{(l-1)} \right). \tag{5.9}$$

Applied in the listed order, these equations represent our full GINA architecture. To create our graph-level readout h_G, we simply sum the last-layer node representations $h_i^{(L)}$:

$$h_G = \sum_{v_i \in V} h_i^{(L)}. \tag{5.10}$$

Then, we pass h_G as input to a softmax function for classification:

$$y_G = \mathrm{softmax}(h_G). \tag{5.11}$$

where the position of the highest score in y_G indicates the predicted class of the input graph. For a visual illustration of the GINA architecture, see Fig. 5.2.

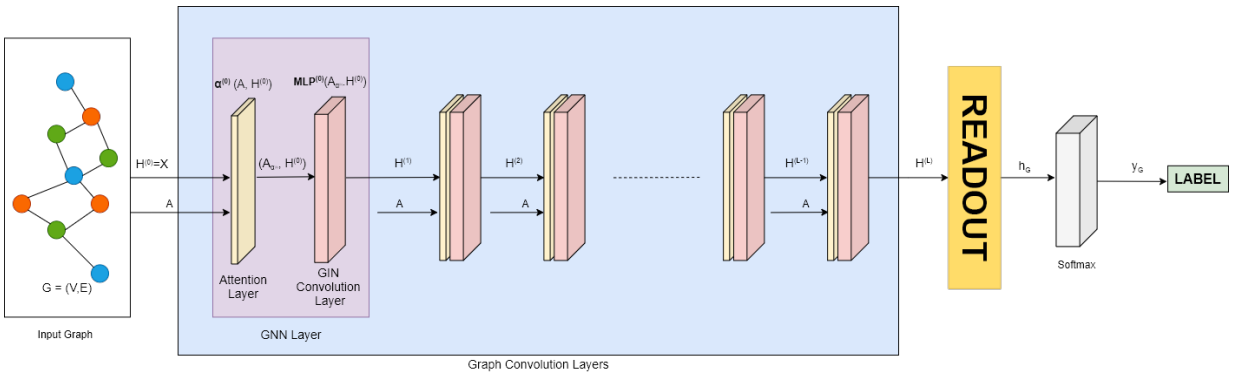


Figure 5.2: GINA architecture. Note that $H^{(l)}$ is the matrix of the l^{th}-layer node embeddings $h^{(l)}$.

In the next chapter, we evaluate these architectures on a number of benchmark datasets, detailing the full process of training and testing as well as discussing the results.

CHAPTER 6

Experimental Procedure

In this chapter, we extensively detail our experimental procedure and discuss our empirical results. We present, in particular: (i) the benchmark datasets we use to train and evaluate our models, GCNA and GINA, (ii) the software tools we use, (iii) the architectures of our models and the baselines against which we compare them, GCN and GIN, (iv) the detailed descriptions of data preprocessing, training, testing, and hyperparameter tuning procedures and (v) the results we achieve for each model on each dataset.

6.1 Datasets

To evaluate our models, we use four real-world graph classification benchmark datasets among the most widely used in GNN literature.[1] We also artificially generated and used a dummy dataset early on to validate the implementation of our models. Some dataset statistics are summarized in Table 6.1 and a brief description of each dataset is given in this section.

6.1.1 Dummy Dataset

To test the correct numerical implementation of our architectures (including baselines, since we also code those from scratch), we created an artificial dataset containing a certain

[1] TU Dortmund University maintains a database of many graph datasets. It can be accessed through this link: `https://chrsmrrs.github.io/datasets/docs/datasets/`

number of random graphs.

For each graph, we randomly generate an adjacency matrix, a node feature matrix and a label. These graphs have a predefined—and equal—number of nodes. Adjacency matrices are created by summing randomly generated triangular binary matrices and their transpose. We then generate a binary label for each node in each graph based on whether it has neighbors (label = 1) or not (label = 0). This gives some sense to our labels and prevents our data from being completely random, which can lead to erratic model behavior and thus make debugging more difficult.

We used this dataset primarily to debug potential mistakes in our code and to assess our models' behavior on a simple, binary node classification task. Therefore, we do not report performance results for this dataset.

6.1.2 ENZYMES

ENZYMES is a dataset consisting of 600 graphs representing the protein tertiary structures (i.e., three-dimensional shapes) of various proteins obtained from the BRENDA [64] enzymes database.

Nodes in these graphs represent the protein secondary structures (the three-dimensional form of local segments of proteins, i.e., helices, sheets and turns) and edges represent whether those structures are neighbors in space. These graphs are divided into 6 classes according to their Enzyme Commission number (EC number), which is a numerical classification scheme based on the chemical reactions that an enzyme catalyzes. This dataset also contains attributes for each node representing physical and chemical information thereof. These attributes include hydrophobicity and the van der Waals volume (the space occupied by the molecule, which is impenetrable to other molecules at ordinary temperatures).

6.1.3 PTC

The PTC dataset was created for the Predictive Toxicology Challenge [29] to compare different approaches for the prediction of rodent carcinogenicity of new compounds (i.e., the ability of a compound to cause cancer to a rodent) based only on information derived from their structure. The PTC dataset is divided into four smaller datasets based on the gender of the rodents (Male/Female) and and their type (Mice/Rats).

In our case, we used the PTC-MR (Male Rats) dataset. This dataset contains 344 graphs representing different chemical compounds labeled according to their carcinogenicity on rodents. The nodes and edges on these graphs represent atoms and the chemical bonds between them.

Property	Graphs	Classes	Avg. n	Max. n	d
MUTAG	188	2	17.93	28	–
PTC	344	2	14.29	64	–
ENZYMES	600	6	32.63	126	18
SYNTHIE	400	4	95	100	15

Table 6.1: Properties of the tested datasets. n is the number of nodes, d is the number of node features, Avg. n is the average number of nodes per graph and Max. n is the maximum number of nodes per graph.

6.1.4 MUTAG

The MUTAG [14] dataset consists of 188 graphs of chemical compounds wherein nodes represent atoms and edges represent the chemical bonds between them. These graphs are divided into two classes according to their mutagenic[2] effect on special bacteria.

6.1.5 Synthie

Synthie [49] is a synthetic (i.e., artificially generated) dataset containing 400 graphs divided into four classes. Nodes have real-valued attribute vectors of dimension $d = 15$. The generation procedure of this dataset is explained in details in [49].

Fig. 6.1 shows class distributions for ENZYMES, PTC, MUTAG and Synthie. With the exception of Synthie, all the datasets are rather well-balanced. This gives us insight on which performance measure to choose.

6.2 Baselines

To assess the potential benefits of using an attention mechanism, we compare our architectures, GCNA and GINA, with their respective versions without attention, namely GCN [40] and GIN [84], as baselines.

The next section details our experimental procedure.

[2]Mutagens are physical or chemical agents that change the genetic material (DNA) of an organism. This increases the number of mutations of the organism, potentially causing cancer.

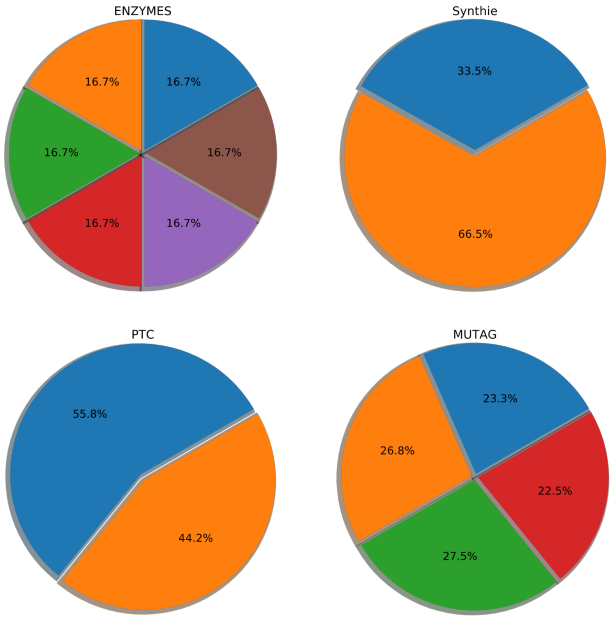


Figure 6.1: Class distribution for ENZYMES, PTC, MUTAG and Synthie datasets.

6.3 Detailed Experimental Setup

6.3.1 Software

Python

The Python programming language has become the standard for most machine learning and artificial intelligence applications. One of the main reasons for that is the great software library ecosystem that it offers, where many libraries are optimized to handle basic machine learning algorithms, data preprocessing and data visualization. The libraries we used in this work are described here:

- **NumPy** [52, 73] is a package dedicated to numerical computation in Python. It includes support for multi-dimensional arrays (vectors, matrices, tensors) and offers a large number of functions that can be used on those arrays.
- **scikit-learn** [55] offers a large number of basic ML algorithms such as clustering, logistic regression and support vector machines. It also includes support for data preprocessing and visualisation through its numerous functions.
- **Matplotlib** [33] is a data visualization library used to create 2D and 3D plots, histograms, pie charts, and other forms of visualization.

- **NetworkX** [24] is a package for the manipulation of graph data. It is particularly useful for preprocessing graph datasets, where raw data—which is stored in text files—needs to be used to efficiently generate adjacency and features matrices that are compatible with NumPy.

To efficiently implement deep learning models along with all the necessary computations, however, we need a high-performance machine learning computation framework, of which many open-source options are available. For this work, we use PyTorch [54].

PyTorch

PyTorch [53] is an open-source deep learning framework on Python supported by Facebook's AI Research group (FAIR). This framework is very flexible, easy to learn, and well documented. Additionally, PyTorch integrates seamlessly with most libraries used in deep learning and data analysis (such as NumPy), streamlining the data preprocessing procedure. Moreover, PyTorch leverages the computational power of graphics processing units (GPUs), which are an all-around much better optimized tool for tensor operations than central processing units (CPUs), making the training process significantly faster.

Another major advantage of PyTorch is its differentiation engine *Autograd*, which provides automatic differentiation (i.e., backpropagation and gradient computation) for all operations on tensors, meaning that we only need to define the forward propagation structure of our model.

Google Colab

Training deep learning models can be very demanding in terms of computational resources. Our models in particular consist of multiple graph convolution layers that have high numbers of trainable parameters, making training on personal laptops with mainstream GPUs very time-consuming. This is why we use the Google Colab platform.

Google Colab[3] is a free programming environment that runs entirely on cloud-based servers and offers users free access to an Nvidia Tesla K80 GPU for sessions of up to 12 hours. Moreover, Google Colab servers are preloaded with a Python environment, making this a convenient tool for our purposes.

[3] `https://colab.research.google.com`

6.3.2 Data Preprocessing

The necessary software tools introduced, we detail here our data preprocessing procedure.

First, we use the NetworkX library [24] to generate NumPy-compatible adjacency and feature matrices from the raw datasets (stored in text files and available at [38]. Since graphs in each dataset vary in size (i.e., they have different numbers of nodes), we equalize adjacency and node feature matrices' dimensions across all graphs in a given dataset in order to leverage PyTorch's efficient built-in data manipulation tools.

Specifically, to achieve this while preserving structural information about input graph data, we use the size of the largest graph in the dataset (say it has n nodes, thus an adjacency matrix $A \in \mathbb{N}^{n \times n}$ and a node feature matrix $X \in \mathbb{R}^{n \times d}$, d being the number of node features) as the common size for all other graphs in the dataset. For these other graphs, we fill each adjacency matrix with zeros until we reach the desired $n \times n$ dimension. We do the same with node feature matrices: we fill them with extra, zero-valued rows until each matrix is of dimension $n \times d$. This process is called *zero-padding*.

6.3.3 Architectures

As mentioned previously, we used the PyTorch framework to implement all the models we investigate; this includes our models, GCNA and GINA, as well as the baselines, GCN and GIN.[4] We started by building the elementary layers for each architecture, that is, graph convolution layers, MLP layers and attention layers. We then combine these layers to create the GIN(A) and GCN(A)[5] models (see equations in Sections 3.3, 3.4, 5.2 and 5.3). Each architecture is determined by a set of hyperparameters, namely:

- The number of graph convolution (GC) layers: this is specific to the GIN(A) architecture. It corresponds to the highest value for k and l in Eq. (3.7) and (5.8), respectively. As for GCN(A), we fix the number of GC layers to two, as done in [40].
- The number of layers in each MLP: this is also specific to GIN(A). It indicates the size of each MLP.
- The number of hidden units: for GIN(A), it refers to the dimension of the hidden layers in each MLP, i.e., the size of its linear mappings. For GCN(A), it refers to the dimension of the linear mappings determined by matrices $W^{(0)}$ and $W^{(1)}$ (see Eq. (3.4) and (5.4)).

All tested values of the aforementioned architecture-related hyperparameters are detailed in Tables A.2 and A.1 in Appendix A.

[4] Our code is available at `https://github.com/theatamna/PFE`

[5] For the sake of brevity, we write GCN(A) or GIN(A) instead of GCN and GCNA or GIN and GINA.

6.3.4 Regularization and Training Hyperparameters

For regularization, we apply dropout and weight decay, both described in Section 2.7. When used, dropout is applied to the output of each convolutional layer, as well as to the attention coefficients described in Eq. (5.1); in other words, each node of the graph is only exposed to a stochastically sampled neighborhood. We also use batch normalization (see Subsection 2.9.1) on every layer on GIN as well as GINA. There are also hyperparameters specific to the training procedure which we recall here:

- Batch size: it defines the number of data samples to use by SGD to update the gradients (model's parameters). We use batches of size 128.
- Number of epochs: it defines the number of times SGD passes through the entire dataset.
- Learning rate (lr): a positive scalar η that defines the step size taken towards the optimum at each iteration.
- Learning rate decay: it determines the amount $1 - \gamma$ by which the learning rate η is decreased each x epochs, i.e., $\eta \leftarrow \gamma.\eta$ every x epochs. We use $\gamma = 0.8$ and $x = 50$.
- Dropout: it randomly deactivates neurons with a given probability p to reduce overfitting. In our case, dropout can be applied to the graph convolution layers of our architectures, or to the attention layers.
- Weight decay (wd): it determines the importance of the L_2 regularization term in the loss function.

While we fix some of these hyperparameters to a given value, we tune the remaining as explained in Section 6.4. All tested hyperparameter configurations can be found in Tables A.2 and A.1 in Appendix A.

6.3.5 Training Procedure

We train our models by performing 5-fold cross-validation for each tested hyperparameter configuration, where we repeat the training 5 times, each using a different fold as test set and the remaining folds as training set.[6] We use the Adam optimizer [39] introduced in Subsection 2.4.2 on the cross-entropy loss defined in Eq. (2.19). To assess the performance of each architecture for a given configuration, we report the average test accuracy over all 5 test folds along with its standard deviation, as well as the maximum accuracy over all 5

[6]Cross-validation is important as it allows each and every example in the considered dataset to be used for testing, thus helping in checking the model's ability to generalize to broader data.

test folds. We opted for the accuracy as our performance measure because it is well-suited for balanced datasets, where classes have approximately the same number of samples. In our case, most of the datasets we use are well balanced as illustrated in Fig. 6.1. For a better interpretation of the test set performance, we also investigate the learning curves (training loss and training accuracy curves) of our models, as illustrated in Section 6.4.

Due to the complexity of the models and the size of the datasets, we had to perform the training procedure on the Google Colab platform. For a given dataset, we first load it from our Google Drive repository and perform the preprocessing step explained in Subsection 6.3.2. The dataset is then split into 5 folds for cross-validation. Since Colab only offers limited runtime, we had to create checkpoints after each fold where we save the training loss and training accuracy for each epoch, as well as the test accuracy for that fold.

Hyperparameter Tuning

Due to the high computational cost of grid and random search, we did not use either of them for our hyperparameter tuning. In fact, performing grid search, for example, for all of our hyperparameters (10 for GIN(A) and 8 for GCN(A)) on the Google Colab platform is impractical. Instead, we opted for an iterative process where we optimize one hyperparameter while keeping the others fixed. That is, at each tuning iteration, we perform 5-fold cross-validation for a set of configurations where only one hyperparameter is varied. Then, we pick the configuration with the best test performance and generate a new set of configurations by varying another hyparameter. We repeat this process until we obtain satisfactory results.

To reduce the number of hyperparameters to tune, we fix some of them across all models and all datasets as mentioned previously. In particular, we fix the number of graph convolution layers in GCN(A) to 2 throughout all experiments.[7] We also set the batch size to 128 and the learning rate decay to 0.8 every 50 epochs.

We start our tuning process from the configuration shown in Table 6.2 that we deem reasonable and from which we generate a set of configurations by varying one hyperparameter. We then perform the tuning procedure as described previously.

6.4 Results

We discuss in this section both training and test results obtained by GCN, GCNA, GIN and GINA on the four datasets presented in Section 6.1 (except the dummy dataset):

[7]The parameter "#GC layers" (number of graph convolution layers) for GCN(A) is omitted in configuration tables for this reason.

ENZYMES, PTC, MUTAG and Synthie. In particular, we discuss two hyperparameter configurations: the initial configuration with which we began training/testing and the best configuration we achieved. The full list of all tested configurations can be found in Appendix A.

6.4.1 Initial Configuration

Table 6.2 lists the hyperparameters of the initial configuration for each model. We use the same number of hidden units at each layer for all architectures. We also train for the same number of epochs with PyTorch's default learning rate for Adam. As for regularization, we use a weight decay of 10^{-3}. Fig. 6.2 shows learning curves (training loss and training accuracy) for one fold on each dataset for all models. Test results are shown in Table 6.3.

Model	lr	#epochs	Hid. dim.	Dropout	wd	#GC layers	#MLP layers
GIN(A)	0.001	300	128	0.0	10^{-3}	5	2
GCN(A)	0.001	300	128	0.0	10^{-3}	–	–

Table 6.2: Initially tested configuration for each model.

Training Results

Looking at the learning curves in Fig. 6.2, we see that GIN(A) models are able to almost perfectly fit all the training sets, achieving 100% training accuracy on the ENZYMES and Synthie datasets and over 90% on the PTC and MUTAG datasets. The corresponding loss function curves decrease much more sharply compared to GCN(A) and reach much lower values at the end of training. As such, GIN(A) models significantly outperform GCN(A) models on training sets (especially on the ENZYMES dataset) as shown by the accuracy curves, which is to be expected as GIN is a more complex architecture than GCN and, as such, has more representational power.

With regards to the attention mechanism, we see that model variants with it achieve better or similar accuracy: GCNA slightly outperforms GCN on all but the PTC dataset (with similar loss curves across all datasets) while GINA achieves lower loss values compared to GIN on three datasets, the fourth being equal. Accuracy-wise, since even the standard GIN learns well on training data, GINA achieves similar results and both architectures are able to fit training data extremely well.

Test Performance

Table 6.3 summarizes the test accuracy achieved by each model with the initial configuration on each dataset. Comparing these results, we immediatly see that GINA achieves

the best results on ENZYMES and Synthie, GIN is the best on MUTAG, while GCN tops the results on PTC. Clearly, the attention mechanism brings important improvements on GINA compared to GIN on three out of four datasets (up to a significant +5.6% on Synthie). On the other hand, attention only improves the performance of GCNA compared to GCN on one dataset (+6.2% on ENZYMES).

Model	GCN		GCNA		GIN		GINA	
	Avg. acc.	Max.	Avg. acc.	Max.	Avg. acc.	Max.	Avg. acc.	Max.
MUTAG	82.0 ± 4.86	89.0	80.8 ± 7.96	94.0	**84.0 ± 3.10**	**86.0**	82.0 ± 5.44	89.0
PTC	**56.4 ± 3.01**	**60.0**	55.0 ± 5.93	65.0	49.2 ± 6.94	57.0	52.0 ± 3.52	57.0
ENZYMES	25.8 ± 3.97	31.0	32.0 ± 4.0	37.0	54.0 ± 2.28	58.0	**56.6 ± 2.87**	**60.0**
Synthie	44.4 ± 4.41	51.0	40.8 ± 3.82	45.0	79.0 ± 4.77	87.0	**84.6 ± 2.42**	**88.0**

Table 6.3: Test results for each model on each dataset for the initial configuration. Reported is the average test accuracy (on 5 folds) with the standard deviation, as well as the maximum test accuracy. Bold font indicates the best result for each dataset.

To be more specific about the impact of the attention mechanism, we notice that, for ENZYMES, it improves performance across the board: GINA performs better than GIN while GCNA outperforms GCN. However, test accuracy for GIN(A) is much lower than training accuracy, suggesting that these models are overfitting; thus, more regularization needs to be used. Synthie test results follow the same pattern as ENZYMES for GIN(A); GCN, however, outperforms GCNA. GCN keeps outperforming GCNA on PTC and MUTAG (even achieving the best overall result on PTC), perhaps suggesting that the attention mechanism does not work as well on GCN as on GIN. It should be noted that GIN and GINA exhibit clear overfitting on PTC and ENZYMES, again suggesting that there is a need for more regularization.

There is a particularly noteworthy aspect when looking at these initial results: adding the attention mechanism significantly improves the performance of GIN on datasets where nodes have features. In the next subsection, where we discuss the best achieved results, we see whether the patterns observed here still hold.

6.4.2 Best Configuration

After trying different configurations (see Appendix A) and adding dropout to attention layers with a rate of 0.6,[8] we achieved better test results on all datasets for all the models, as shown in Table 6.5. These results where obtained using the configurations in Table 6.4.

[8]Dropout on attention layers is only used in this subsection, with a rate of 0.6. The dropout values listed in Table 6.4 are for the outputs of each convolutional layer, as in Subsection 6.4.1.

Dataset	Model	Hyperparameters					
		lr	#epochs	Hid. dim.	Dropout	#GC layers	#MLP layers
MUTAG	GCN	0.001	300	64	0.0	–	–
	GCNA	0.001	300	128	0.0	–	–
	GIN	0.001	300	128	0.0	2	4
	GINA	0.001	300	128	0.0	2	6
PTC	GCN*	5e-05	300	48	0.0	–	–
	GCNA*	5e-05	300	192	0.0	–	–
	GIN*	0.0001	300	80	0.2	4	8
	GINA	0.001	300	128	0.0	3	6
ENZYMES	GCN*	0.0008	500	128	0.0	–	–
	GCNA	0.01	350	128	0.0	–	–
	GIN	0.001	500	128	0.0	4	8
	GINA	0.001	500	128	0.0	4	8
Synthie	GCN*	0.001	300	128	0.25	–	–
	GCNA*	0.001	300	400	0.0	–	–
	GIN	0.001	300	128	0.0	2	4
	GINA	0.001	300	128	0.0	2	4

Table 6.4: Best configuration for each model on each dataset. Dropout rates in the table are applied to graph convolution layers. A dropout rate of 0.6 is additionally applied to attention layers for GCNA and GINA in each configuration. A weight decay of 10^{-3} is also applied everywhere except for configurations marked with '*'.

Training Results

Looking at the learning curves in Fig. 6.3, we notice that loss function curves are more noisy—especially for GIN(A)—compared to the curves in Subsection 6.4.1. This indicates that further tuning is required in order to mitigate the noise during training. A possible solution could be to reduce the dropout rate on attention layers or to use more graph convolution (GC) layers (unlike in the initial configuration, there seems to be more GC layers than MLP layers for GIN(A)). When looking at training accuracy, we see that GCNA equals or outperforms GCN on all datasets while reaching lower loss function values. The reverse is true for GIN(A): GIN achieves equal or higher training accuracies compared to GINA while reaching lower loss function values on ENZYMES and Synthie.

Test Performance

Comparing the test results in Table 6.5, we notice the same trends as in Subsection 6.4.1: GINA achieves the best results on ENZYMES and Synthie, GIN is the best on MUTAG, while GCN tops the results on PTC. After tuning, the attention mechanism brings even more significant improvements on GINA compared to GIN on the two datasets with node features (+3.6% on Synthie and an impressive +14.2% on ENZYMES, while closing the gap to GIN on MUTAG and being extremely close on PTC). On the other hand, we still see that attention does not help GCNA much and that it still overfits compared to GCN (its only advantage is +1.4% on MUTAG).

These test results also stress the importance of hyperparameter tuning: for the best-performing models, we notice performance improvements ranging from +3.2% up to +12.8% on the four datasets compared to the initial configuration in Subsection 6.4.1. We also see that regularization contributes to better generalization, especially for GINA: although its performance on the training set decreased on MUTAG and it takes more epochs to train on ENZYMES, its test performance improves significantly nonetheless. It also generalizes better on Synthie compared to the baseline GIN.

Model	GCN		GCNA		GIN		GINA	
	Avg. acc.	Max.	Avg. acc.	Max.	Avg. acc.	Max.	Avg. acc.	Max.
MUTAG	84.4 ± 6.28	94.0	85.8 ± 3.91	89.0	**87.2 ± 2.8**	**92.0**	86.2 ± 5.30	94.0
PTC	**60.6 ± 5.71**	**66.0**	58.2 ± 3.96	66.0	57.2 ± 4.83	63.0	57.0 ± 4.86	65.0
ENZYMES	38.0 ± 5.40	46.0	35.8 ± 1.72	38.0	55.2 ± 4.06	60.0	**69.4 ± 3.38**	**74.0**
Synthie	53.6 ± 5.00	63.0	46.2 ± 6.55	53.0	87.4 ± 1.35	90.0	**91.0 ± 3.84**	**97.0**

Table 6.5: Best test results after hyperparameter tuning for each model on each dataset. Reported is the average test accuracy (on 5 folds) with the standard deviation, as well as the maximum test accuracy. Bold font indicates the best result for each dataset.

6.4.3 Discussion

Our experiments indicate that the GIN architecture benefits in many cases from the addition of the attention mechanism, particularly on datasets where nodes have features. This is, as we have seen, not usually the case for the GCN architecture. It may be that further hyperparameter tuning is required, or it may be that the GCN architecture is fundamentally limited in such a way that attending over a node's most important neighbors cannot bring tangible performance benefits. Having a closer look at the graphs' structure—node degrees in particular—could also give us more insight into the results, as it could be that attention helps most with graphs with large neighborhoods.

These experiments also demonstrate how sensitive neural network architectures are to the choice of hyperparameters and that learning useful representations requires a careful and extensive tuning procedure.

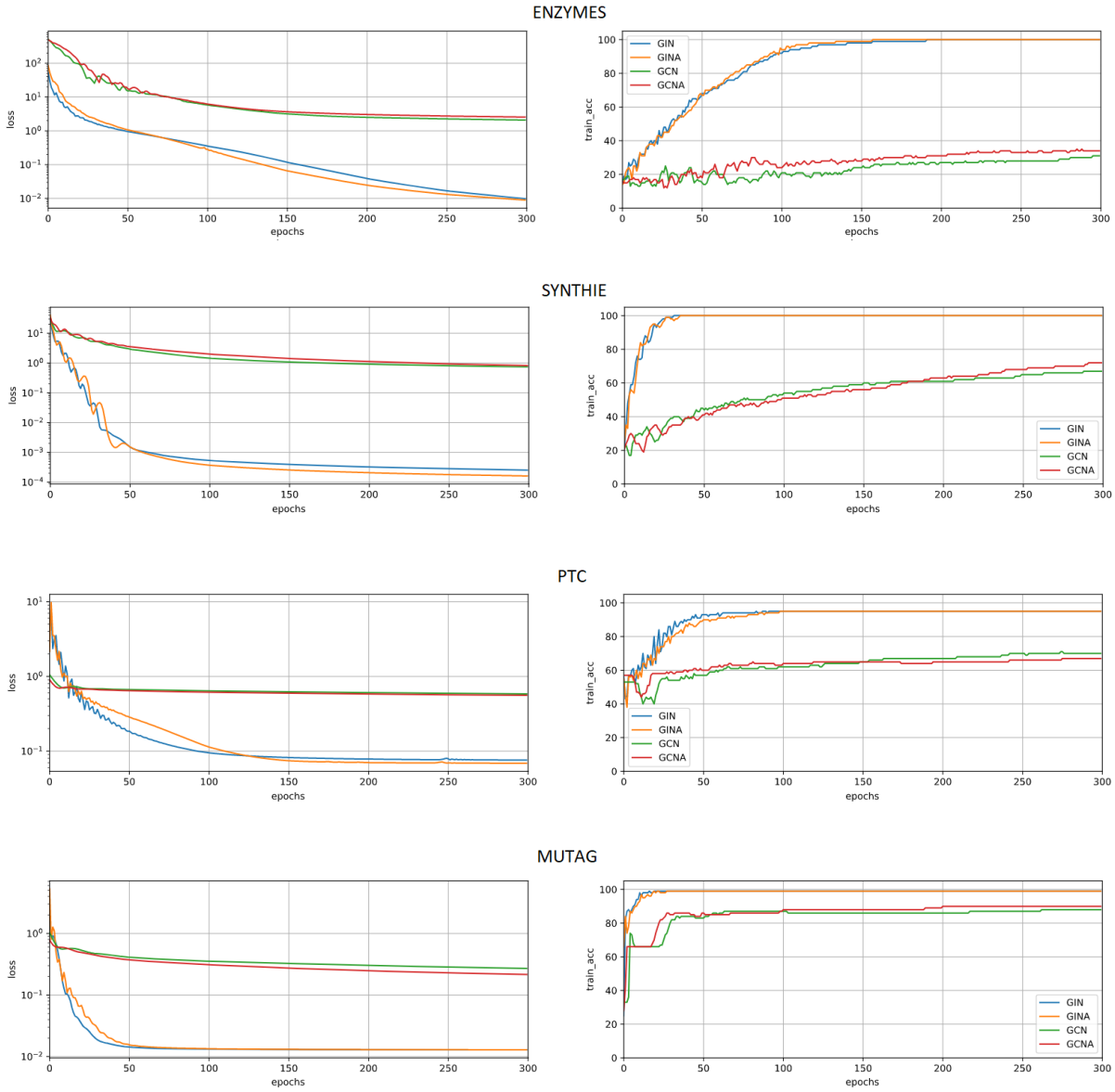


Figure 6.2: Learning curves for all the models on each dataset with the initial configuration. Left: The evolution of training loss for each model. Right: The evolution of training accuracy for each model.

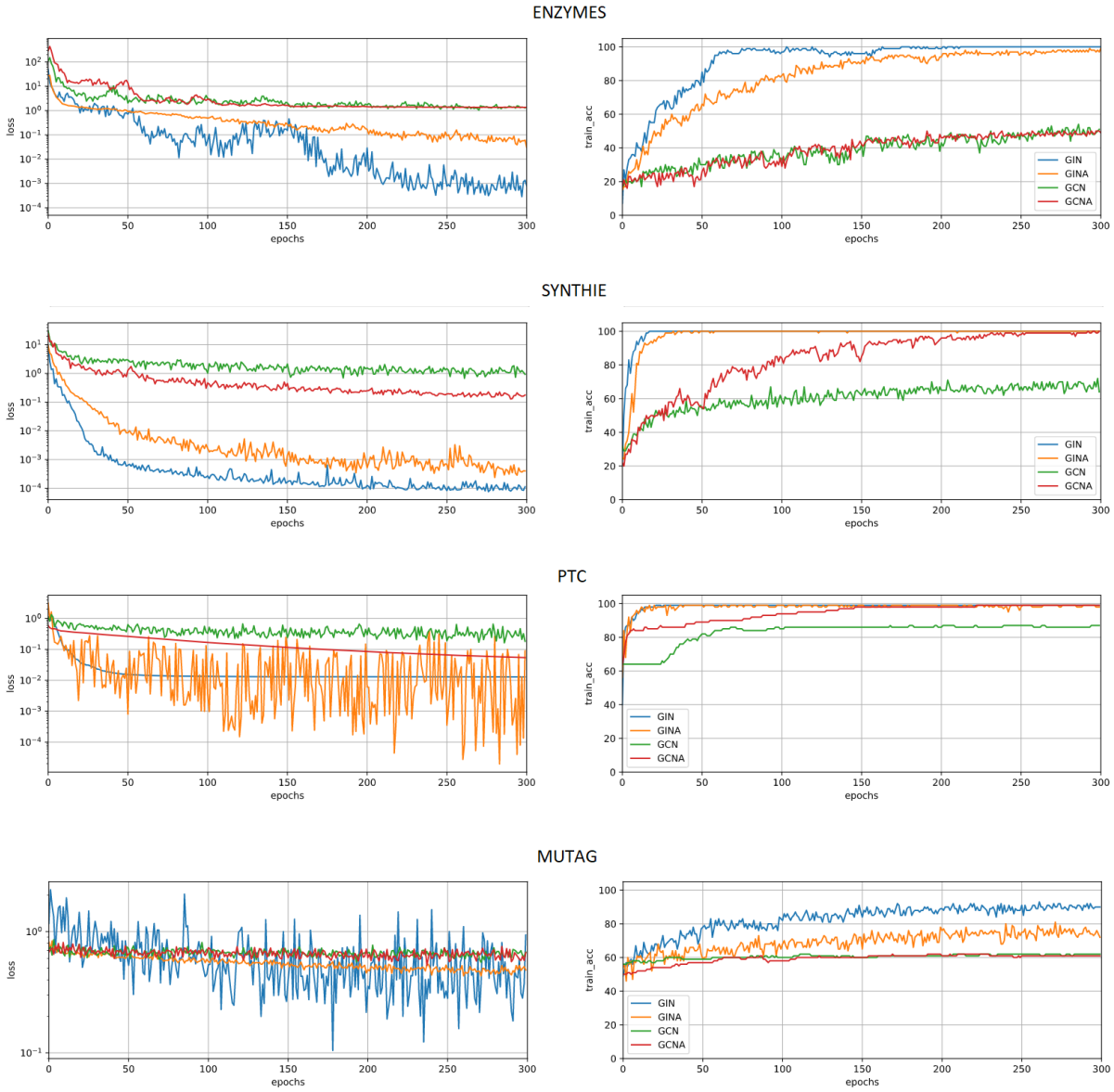


Figure 6.3: Learning curves for all the models on each dataset for configurations with the best test results. Left: The evolution of training loss for each model. Right: The evolution of training accuracy for each model.

General Conclusion

In this work, we motivated the study of graphs as a powerful data structure that is particularly relevant to control engineering applications, as graphs are able to capture rich information about the structure of physical systems. As such, we introduced and extensively reviewed the state of the art in graph neural newtork (GNN) research, including breakthrough applications in control engineering such as GNNs for inference, system identification, control and human action recognition. We then took two reference GNN architectures (GCN [40] and GIN [84]) as baselines and augmented them with an attention mechanism (introduced in [74]), preserving the fundamental property of invariance to graph isomorphism and proposing two novel architectures: GCNA and GINA.

Through a series of experiments on benchmark graph datasets, we studied the impact of attention mechanisms on a supervised learning, graph classification task. These experiments highlighted some interesting findings: the attention mechanism we use is particularly useful in datasets where nodes have non-trivial features, helping GINA achieve impressive performance gains of up to 14%. The experiments also highlight how critical hyperparameter tuning is to achieving good performance when training machine learning models.

Our findings, coupled with our review of GNNs in control-related applications, open up some very interesting perspectives for future work, building on the foundation we provided here:

- The architectures we introduced have very high expressive power, giving them the ability to learn complex representations for input graph data. As such, it may be useful to think of ways to apply them to human pose estimation tasks, as graphs are ideally suited to capturing the human body's physical structure. For the same

reasons, another application with a potentially high impact would be to combine our architectures with a recurrent neural network architecture and use it for sequential tasks such as human pose estimation through skeleton data sequences, in a similar way to what Si et al. did in [66].

- Proper hyperparameter tuning is critical in determining a model's performance and can be particularly tedious to do by hand, especially on deep, complex architectures like ours. An interesting perspective would be to use AutoML tools which automate the process of tuning. Some are specifically designed for neural architecture search and hyperparameter optimization such as the emerging Auto-PyTorch [48], which offers BOHB [18], a multi-fidelity/Bayesian optimization-based algorithm, far more sophisticated than grid or random search. SMAC [34] is another interesting hyperparameter optimization tool.

Appendices

APPENDIX A

All Tested Configurations

A.1 GIN(A)

Config. n°	Hyperparameters					
	lr	#epochs	Hid. dim.	Dropout	#GC layers	#MLP layers
1	0.001	300	128	0.0	2	4
2	0.001	300	128	0.0	2	6
3	0.001	300	128	0.0	2	8
4	0.001	300	128	0.0	3	4
5	0.001	300	128	0.0	3	6
6	0.001	300	128	0.0	3	8
7	0.001	300	128	0.0	4	4
8	0.001	300	128	0.0	4	4
9	0.001	300	128	0.0	4	6
10	0.01	300	128	0.0	2	8
11	0.0001	300	128	0.0	2	4
12	0.001	300	128	0.05	2	6
13	0.001	300	128	0.1	2	6
14	0.001	300	128	0.15	2	6
15	0.001	300	128	0.2	2	6
16	0.001	300	128	0.25	2	6
17	0.001	300	128	0.3	2	6
18	0.001	300	128	0.35	2	6
19	0.001	300	128	0.40	2	6
20	0.001	300	128	0.45	2	6
21	0.001	300	16	0.0	2	6

22	0.001	300	32	0.0	2	6
23	0.001	300	64	0.0	2	6
24	0.001	300	256	0.0	2	6
25	0.001	300	128	0.05	2	4
26	0.001	300	128	0.1	2	4
27	0.001	300	128	0.15	2	4
28	0.001	300	128	0.20	2	4
29	0.001	300	128	0.25	2	4
30	0.001	300	128	0.30	2	4
31	0.001	300	128	0.35	2	4
32	0.001	300	128	0.4	2	4
33	0.001	300	128	0.45	2	4
34	0.001	300	16	0.0	2	4
35	0.001	300	32	0.0	2	4
36	0.001	300	64	0.0	2	4
37	0.001	300	256	0.0	2	4
38	0.001	300	512	0.0	2	4
39	0.0001	300	128	0.05	4	8
40	0.0001	300	128	0.1	4	8
41	0.0001	300	128	0.15	4	8
42	0.0001	300	128	0.2	4	8
43	0.001	300	8	0.0	4	8
44	0.001	300	16	0.0	4	8
45	0.001	300	32	0.0	4	8
46	0.001	300	40	0.0	4	8
47	0.001	300	48	0.0	4	8
48	0.001	500	192	0.0	2	4
49	0.0005	500	192	0.0	2	4
50	0.0003	500	192	0.0	2	4
51	0.0001	500	192	0.0	2	4
52	8e-05	500	192	0.0	2	4
53	5e-05	500	192	0.0	2	4
54	3e-05	500	192	0.0	2	4
55	1e-05	500	192	0.0	2	4

Table A.1: All tested hyperparameter configurations for GIN and GINA architectures on all datasets.

A.2 GCN(A)

Config n°	Hyperparameters			
	lr	#epochs	Hid. dim.	Dropout
1	0.001	300	128	0.0
2	0.001	300	128	0.05
3	0.001	300	128	0.1

Config n°	Hyperparameters			
	lr	#epochs	Hid. dim.	Dropout
18	0.0001	300	128	0.15
19	0.0001	300	128	0.20
20	0.001	300	128	0.0

4	0.001	300	128	0.15	21	0.0001	300	192	0.0
5	0.001	300	128	0.20	22	0.0001	500	192	0.0
6	0.001	300	128	0.25	23	0.0003	300	192	0.0
7	0.001	300	128	0.30	24	0.0003	500	192	0.0
8	0.001	300	128	0.35	25	0.0005	500	192	0.0
9	0.001	300	128	0.4	26	0.0008	300	192	0.0
10	0.001	300	128	0.45	27	0.0008	500	192	0.0
11	0.001	300	16	0.0	28	0.001	300	192	0.0
12	0.001	300	32	0.0	29	0.001	300	192	0.0
13	0.001	300	64	0.0	30	0.001	500	192	0.0
14	0.01	300	128	0.0	31	1e-05	300	192	0.0
15	0.0001	300	128	0.0	32	1e-05	500	192	0.0
16	0.0001	300	128	0.05	33	5e-05	300	192	0.0
17	0.0001	300	128	0.10	34	5e-05	500	192	0.0

Table A.2: All tested hyperparameter configurations for GCN and GCNA architectures on all datasets.

Bibliography

[1] A. Ahmed, N. Shervashidze, S. N. V. J., and Smola, A. Distributed large-scale natural graph factorization. In *WWW* (2013).

[2] Atamna, A., Sokolovska, N., and Crivello, J. A principled approach to analyze expressiveness and accuracy of graph neural networks. In *International Symposium on Intelligent Data Analysis (IDA)* (2020), pp. 27–39.

[3] Atwood, J., and Towsley, D. Diffusion-convolutional neural networks. In *NIPS* (2016).

[4] Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations (ICLR)* (2015).

[5] Bai, Y., Ding, H., Qiao, Y., Marinovic, A., Gu, K., Chen, T., Sun, Y., and Wang, W. Unsupervised inductive graph-level representation learning via graph-graph proximity. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)* (2019), pp. 1988–1994.

[6] Belkin, M., and Niyogi, P. Laplacian eigenmaps and spectral techniques for embedding and clustering. In *NIPS* (2002).

[7] Bergstra, J., and Bengio, Y. Random search for hyper-parameter optimization. *The Journal of Machine Learning Research 13*, 1 (2012), 281–305.

[8] Borgwardt, K. M., Ong, C. S., Schönauer, S., Vishwanathan, S. V. N., Smola, A. J., and Kriegel, H.-P. Protein function prediction via graph kernels. *Bioinformatics 21*, 1 (2005), 47–56.

[9] Boser, B. E., Guyon, I. M., and Vapnik, V. N. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory* (New York, NY, USA, 1992), COLT '92, Association for Computing Machinery, p. 144–152.

[10] Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. Spectral networks and locally connected networks on graphs. In *ICLR* (2014).

[11] Cao, S., Lu, W., and Xu, Q. Deep neural networks for learning graph representations. In *AAAI* (2016).

[12] Chen, J., Ma, T., and Xiao, C. FastGCN: Fast learning with graph convolutional networks via importance sampling. In *International Conference on Learning Representations* (2018).

[13] Chicco, D. Ten quick tips for machine learning in computational biology. *BioData mining 10*, 1 (2017), 35.

[14] Debnath, A. K., Lopez de Compadre, R. L., Debnath, G., Shusterman, A. J., and Hansch, C. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry 34*, 2 (1991), 786–797.

[15] Defferrard, M., Bresson, X., and Vandergheynst, P. Convolutional neural networks on graphs with fast localized spectral filtering. In *NIPS* (2016).

[16] Douglas, B. L. The weisfeiler-lehman method and graph isomorphism testing, 2011.

[17] Duvenaud, D. K., Maclaurin, D., Iparraguirre, J., Bombarell, R., Hirzel, T., Aspuru-Guzik, A., and Adams, R. P. Convolutional networks on graphs for learning molecular fingerprints. In *NIPS* (2015).

[18] Falkner, S., Klein, A., and Hutter, F. BOHB: Robust and efficient hyperparameter optimization at scale. In *Proceedings of the 35th International Conference on Machine Learning* (2018), J. Dy and A. Krause, Eds., vol. 80 of *Proceedings of Machine Learning Research*, PMLR, pp. 1437–1446.

[19] Fey, M. Just jump: Dynamic neighborhood aggregation in graph neural networks, 2019.

[20] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *ICML* (2017).

[21] Goodfellow, I., Bengio, Y., and Courville, A. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[22] Grover, A., and Leskovec, J. node2vec: Scalable feature learning for networks. In *KDD* (2016).

[23] Gu, W., Gao, F., Lou, X., and Zhang, J. Link prediction via graph attention network, 2019.

[24] Hagberg, A. A., Schult, D. A., and Swart, P. J. Exploring network structure, dynamics, and function using networkx. In *Proceedings of the 7th Python in Science Conference* (2008), G. Varoquaux, T. Vaught, and J. Millman, Eds., pp. 11–15.

[25] Hamilton, W., Ying, R., and Leskovec, J. Inductive representation learning on large graphs. In *arXiv:1603.04467* (2017).

[26] Hamilton, W. L., Ying, R., and Leskovec, J. Representation learning on graphs: Methods and applications, 2017.

[27] Hammond, D. K., Vandergheynst, P., and Gribonval, R. Wavelets on graphs via spectral graph theory. *Applied and Computational Harmonic Analysis 30*, 2 (Mar. 2011), 129–150.

[28] He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 770–778.

[29] Helma, C., and Kramer, S. A survey of the predictive toxicology challenge 2000–2001. *Bioinformatics 19*, 10 (2003), 1179–1182.

[30] Hinton, G., and Salakhutdinov, R. Reducing the dimensionality of data with neural networks. In *Science, 313(5786):504–507* (2006).

[31] Hochreiter, S., and Schmidhuber, J. Long Short-Term Memory. *Neural Comput. 9*, 8 (1997), 1735–1780.

[32] Hornik, K., Stinchcombe, M., and White, H. Multilayer feedforward networks are universal approximators. In *Neural networks, 2(5):359–366* (1989).

[33] Hunter, J. D. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering 9*, 3 (2007), 90–95.

[34] Hutter, F., Hoos, H. H., and Leyton-Brown, K. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th International Conference on Learning and Intelligent Optimization* (2011), Springer-Verlag, pp. 507–523.

[35] Ioffe, S., and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *ICML* (2015).

[36] James, G., Witten, D., Hastie, T., and Tibshirani, R. *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company, Incorporated, 2014.

[37] John Boaz Lee, R. R., and Kong., X. Graph classification using structural attention. In *KDD* (2018).

[38] Kersting, K., Kriege, N. M., Morris, C., Mutzel, P., and Neumann, M. Benchmark data sets for graph kernels, 2016.

[39] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization. In *ICLR* (2015).

[40] Kipf, T. N., and Welling, M. Semi-supervised classification with graph convolutional networks. In *ICLR* (2017).

[41] Kiran K. Thekumparampil, Chong Wang, S. O., and Li., L.-J. Attention-based graph neural network for semi-supervised learning. In *arXiv* (2018).

[42] Kumar, A., Irsoy, O., Ondruska, P., Iyyer, M., Bradbury, J., Gulrajani, I., Zhong, V., Paulus, R., and Socher, R. Ask me anything: Dynamic memory networks for natural language processing, 2015.

[43] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., and Jackel, L. D. Backpropagation applied to handwritten zip code recognition. *Neural Comput. 1*, 4 (Dec. 1989), 541–551.

[44] Lee, J. B., Rossi, R. A., Kim, S., Ahmed, N. K., and Koh, E. Attention models in graphs: A survey. *ACM Transactions on Knowledge Discovery from Data (TKDD) 13*, 6 (2019), 1–25.

[45] Li, Q., Gama, F., Ribeiro, A., and Prorok, A. Graph neural networks for decentralized multi-robot path planning, 2019.

[46] M. Ou, P. Cui, J. P. Z. Z., and Zhu, W. Asymmetric transitivity preserving graph embedding. In *KDD* (2016).

[47] Manso, L. J., Jorvekar, R. R., Faria, D. R., Bustos, P., and Bachiller, P. Graph neural networks for human-aware social navigation, 2019.

[48] Mendoza, H., Klein, A., Feurer, M., Springenberg, J. T., Urban, M., Burkart, M., Dippel, M., Lindauer, M., and Hutter, F. Towards automatically-tuned deep neural networks. In *AutoML: Methods, Sytems, Challenges*, F. Hutter, L. Kotthoff, and J. Vanschoren, Eds. Springer, 2018, ch. 7, pp. 141–156.

[49] Morris, C., Kriege, N. M., Kersting, K., and Mutzel, P. Faster kernels for graphs with continuous attributes via hashing, 2016.

[50] Nguyen, D. Q., Nguyen, T. D., and Phung, D. Universal self-attention network for graph classification, 2019.

[51] Niepert, M., Ahmed, M., and Kutzkov, K. Learning convolutional neural networks for graphs. In *ICML* (2016).

[52] Oliphant, T. E. *A guide to NumPy*, vol. 1. Trelgol Publishing USA, 2006.

[53] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop* (2017).

[54] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[55] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[56] Perozzi, B., Al-Rfou, R., and Skiena, S. Deepwalk: Online learning of social representations. In *KDD* (2014).

[57] Piergiovanni, A., and Ryoo, M. S. Representation flow for action recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (June 2019).

[58] Prorok, A. Graph neural networks for learning robot team coordination, 2018.

[59] Qiu, Z., Yao, T., Ngo, C.-W., Tian, X., and Mei, T. Learning spatio-temporal representation with local and global diffusion, 2019.

[60] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. Learning representations by back-propagating errors. *Nature 323* (1986), 533–536.

[61] S. Cao, W. L., and Xu, Q. Grarep: Learning graph representations with global structural information. In *KDD* (2015).

[62] Sanchez-Gonzalez, A., Heess, N., Springenberg, J. T., Merel, J., Riedmiller, M., Hadsell, R., and Battaglia, P. Graph networks as learnable physics engines for inference and control. In *Proceedings of the International Conference on Machine Learning* (2018), vol. 80 of *Proceedings of Machine Learning Research*, pp. 4470–4479.

[63] Santi, P. Topology control in wireless ad hoc and sensor networks. *ACM computing surveys (CSUR) 37*, 2 (2005), 225–232.

[64] Schomburg, I., Chang, A., Ebeling, C., Gremse, M., Heldt, C., Huhn, G., and Schomburg, D. Brenda, the enzyme database: updates and major new developments. *Nucleic acids research 32*, suppl_1 (2004), D431–D433.

[65] Schöning, U. Graph isomorphism is in the low hierarchy. In *Annual Symposium on Theoretical Aspects of Computer Science* (1987), Springer, pp. 114–124.

[66] Si, C., Chen, W., Wang, W., Wang, L., and Tan, T. An attention enhanced graph convolutional LSTM network for skeleton-based action recognition. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2019), pp. 1227–1236.

[67] Simonyan, K., and Zisserman, A. Very deep convolutional networks for large-scale image recognition, 2014.

[68] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research 15*, 56 (2014), 1929–1958.

[69] Sun, K., Xiao, B., Liu, D., and Wang, J. Deep high-resolution representation learning for human pose estimation, 2019.

[70] Sutskever, I., Vinyals, O., and Le, Q. V. Sequence to sequence learning with neural networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2* (Cambridge, MA, USA, 2014), NIPS'14, MIT Press, p. 3104–3112.

[71] T. Pham, T. Tran, D. P., and Venkatesh, S. Column networks for collective classification. In *AAAI* (2017).

[72] Tjomsland, J., Shafti, A., and Faisal, A. A. Human-robot collaboration via deep reinforcement learning of real-world interactions, 2019.

[73] Van Der Walt, S., Colbert, S. C., and Varoquaux, G. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering 13*, 2 (2011), 22.

[74] Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. *ICLR* (2018).

[75] Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. Show and tell: A neural image caption generator. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2015), pp. 3156–3164.

[76] Volodymyr Mnih, Nicolas Heess, A. G., and Kavukcuoglu, K. Recurrent models of visual attention. In *NIPS* (2014).

[77] Wang, D., Cui, P., and Zhu, W. Structural deep network embedding. In *KDD* (2016).

[78] Wang, R., Li, B., Hu, S., Du, W., and Zhang, M. Knowledge graph embedding via graph attenuated attention networks. *IEEE Access 8* (2020), 5212–5224.

[79] Wang, T., Liao, R., Ba, J., and Fidler, S. Nervenet: Learning structured policy with graph neural networks. In *ICLR* (2018).

[80] Wang, T., Zhou, Y., Fidler, S., and Ba, J. Neural graph evolution: Towards efficient automatic robot design, 2019.

[81] Weisfeiler, B., and Lehman, A. A. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia 9*, 2 (1968), 12–16.

[82] Wilson, R. J. *Introduction to graph theory*. Pearson Education India, 1979.

[83] Wu, F., Souza, A., Zhang, T., Fifty, C., Yu, T., and Weinberger, K. Simplifying graph convolutional networks. In *Proceedings of the International Conference on Machine Learning* (2019), vol. 97 of *Proceedings of Machine Learning Research*, pp. 6861–6871.

[84] Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *ICLR* (2019).

[85] Yanardag, P., and Vishwanathan, S. Deep graph kernels. In *SIGKDD* (2015).

[86] Yang, W., Li, S., Ouyang, W., Li, H., and Wang, X. Learning feature pyramids for human pose estimation, 2017.

[87] Zhang, A., Lipton, Z. C., Li, M., and Smola, A. J. *Dive into Deep Learning*. 2020. `https://d2l.ai`.

[88] Zhang, J., Zhang, H., Xia, C., and Sun, L. Graph-bert: Only attention is needed for learning graph representations, 2020.

[89] Zhang, M., Cui, Z., Neumann, M., and Chen, Y. An end-to-end deep learning architecture for graph classification. In *AAAI* (2018).

[90] Zichao Yang, Xiaodong He, J. G. L. D., and Smola., A. Stacked attention networks for image question answering. In *CVPR* (2016).