RÉPUBLIQUE ALGÉRIENNE DÉMOCRATIQUE ET POPULAIRE

MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR ET DE LA RECHERCHE SCIENTIFIQUE

UNIVERSITÉ ABDERRAHMANE MIRA - BÉJAÏA

Faculté des Sciences Exactes

Département de Mathématiques

Mémoire de Projet de Fin de Cycle

En vue de l'obtention du diplôme de Master

Domaine : Mathématiques et Informatique
Filière : Mathématiques
Spécialité : Mathématiques de l'Intelligence Artificielle

Tests et analyse de quelques modèles de réseaux de neurones sur graphes

---

Présenté par : TOUATI Nawel

Devant le jury composé de :

M. ACHROUFENE Achour	Président du jury
M. BOUZIDI Lhadi	Encadrant
Mme REBOUH Nadjette	Examinatrice
Mme BARACHE Bahia	Examinatrice

Année universitaire : 2024 – 2025

Remerciements

Je tiens tout d'abord à exprimer ma gratitude à toutes les personnes qui ont contribué, de près ou de loin, à la réalisation de ce mémoire.

Je tiens à adresser mes remerciements les plus sincères à **Monsieur BOUZIDI Lhadi**, mon encadrant, pour avoir accepté de superviser ce projet. Sa disponibilité, ses conseils avisés et son accompagnement constant ont joué un rôle essentiel dans la réalisation de ce mémoire.

J'exprime également ma reconnaissance à l'ensemble des membres du jury, pour l'honneur qu'ils me font en acceptant d'évaluer ce mémoire. Leurs remarques et suggestions seront pour moi d'un grand enrichissement.

Mes remerciements vont aussi à mes enseignants, pour la qualité de leur encadrement tout au long de ma formation, ainsi qu'à mes camarades et amis, pour leur soutien, leur bienveillance et leurs encouragements tout au long de ce parcours.

Dédicaces

Je dédie ce travail à :

Ma chère mère, pour son amour inconditionnel, sa patience et ses prières silencieuses.

Mon père, pour sa force, ses conseils et ses sacrifices constants.

Mon frère et ma sœur, pour leur présence, leur affection et leurs encouragements.

Ma famille toute entière, pour son soutien moral tout au long de mon parcours.

À vous tous, je témoigne ma profonde gratitude.

Résumé

Les réseaux de neurones sur graphes (GNN) sont des modèles puissants issus de l'apprentissage profond, conçus pour traiter des données structurées sous forme de graphes. Dans ce contexte, l'objectif de ce travail est à la fois théorique et pratique. Il s'agit, d'une part, de comprendre les fondements des réseaux de neurones classiques ainsi que ceux des GNN, et d'autre part, de mettre en œuvre, tester et analyser plusieurs architectures de GNN (GCN, GAT, GraphSAGE, GIN) sur des jeux de données de référence, afin d'évaluer leurs performances sur des tâches variées de prédiction d'attributs de nœuds et de graphes entiers.

Mots-clés : Graph Neural Networks (GNN), apprentissage profond, réseaux de neurones, classification de graphes, classification de nœuds, GCN, GAT, GraphSAGE, GIN.

Abstract

Graph Neural Networks (GNN) are powerful models derived from deep learning, specifically designed to process structured data in the form of graphs. In this context, the objective of this work is both theoretical and practical. On the one hand, it aims to understand the fundamentals of classical neural networks as well as GNNs. On the other hand, it involves implementing, testing, and analyzing several GNN architectures (GCN, GAT, GraphSAGE, GIN) on standard benchmark datasets in order to evaluate their performance on various tasks such as node attribute prediction and whole-graph classification.

Keywords : Graph Neural Networks (GNN), deep learning, neural networks, graph classification, node classification, GCN, GAT, GraphSAGE, GIN.

Table des matières

Table des figures

Liste des tableaux

Liste des abréviations

AGG Aggregation Layer
ANN Artificial Neural Network
CIN++ Complex Interaction Network ++
CNN Convolutional Neural Network
CONV Convolution Layer
ConvNet Convolutional Network
DL Deep Learning
FC Fully Connected Layer
GAT Graph Attention Network
GCN Graph Convolutional Network
GIN Graph Isomorphism Network
GNN Graph Neural Network
Graph-BERT Graph Bidirectional Encoder Representations from Transformers
GraphSAGE Graph Sample and Aggregate
IA Intelligence Artificielle
ML Machine Learning
MLP Multilayer Perceptron
MMA Multi-hop Mutual Attention
PANDA Propagation Attention-based Neighborhood-aware Data Aggregation
POOL Pooling Layer
RNN Recurrent Neural Network
WL Weisfeiler-Lehman

Introduction générale

Les avancées remarquables de l'intelligence artificielle (IA), principalement grâce à l'apprentissage profond (deep learning), ont permis de traiter efficacement des données structurées telles que des images, des textes ou des signaux vocaux. Dans le domaine de l'apprentissage automatique, l'apprentissage profond s'est imposé comme une approche indispensable, s'appuyant essentiellement sur l'utilisation de réseaux de neurones artificiels multicouches.

Parmi ces architectures, les réseaux de neurones artificiels (Artificial Neural Networks – ANN) ont démontré une efficacité remarquable dans le traitement de données vectorielles, notamment dans des domaines comme la vision par ordinateur ou le traitement automatique du langage naturel.

Cependant, les architectures traditionnelles telles que les perceptrons multicouches (MLP), les réseaux convolutifs (CNN) ou encore les réseaux récurrents (RNN) sont essentiellement conçues pour traiter des données euclidiennes, comme les vecteurs, les images ou les séquences.

Néanmoins, de nombreuses données provenant du monde réel, comme les réseaux sociaux, les structures moléculaires ou les systèmes de recommandation, possèdent une nature relationnelle et sont mieux modélisées sous forme de graphes. Ces graphes, composés de nœuds (représentant des entités) et d'arêtes (représentant les relations), nécessitent des modèles capables d'exploiter cette structure topologique complexe.

Face à ces nouvelles exigences, les modèles traditionnels montrent rapidement leurs limites. Ils peinent à capturer la topologie locale et globale des graphes, ce qui réduit leur capacité à exploiter efficacement l'information contextuelle et relationnelle entre les entités. C'est dans ce contexte qu'émergent les Graph Neural Networks (GNN), ou réseaux de neurones sur graphes, qui prolongent les principes de l'apprentissage profond aux données non-euclidiennes.

Malgré leur potentiel croissant, les Graph Neural Networks (GNN) demeurent encore peu abordés dans les formations classiques en intelligence artificielle.

Ce projet vise à combler cette lacune en explorant le fonctionnement des GNN à travers la découverte, l'implémentation et l'évaluation de plusieurs architectures, appliquées à des tâches concrètes de classification de nœuds et de graphes. Les modèles étudiés sont testés sur des jeux de données de référence dans le but d'analyser leurs performances, d'identifier leurs points forts et leurs limites, et de mettre en lumière leur potentiel d'application dans des contextes variés et réalistes.

Ce travail est organisé en quatre chapitres :

— **Le chapitre 1** : présente les bases des réseaux de neurones artificiels : de la modélisation du neurone au fonctionnement général des architectures classiques (perceptron, CNN, RNN), en passant par les mécanismes d'apprentissage (rétropropagation, fonction de coût, mise à jour des poids).

— **Le chapitre 2** : introduit les graphes comme structure de données, leurs types, représentations et domaines d'application, avant de se concentrer sur les réseaux neuronaux graphiques (GNN) : leur fonctionnement, leurs variantes (GCN, GAT, GIN, GraphSAGE) et les cas d'usage typiques.

— **Le chapitre 3** : est dédié à l'implémentation pratique : choix des bibliothèques (PyTorch Geometric, Taipy), description des jeux de données utilisés (Cora, Citeseer, PubMed, MUTAG, PROTEINS), processus d'entraînement, optimisation automatique des hyperparamètres avec Optuna, et développement d'un tableau de bord interactif pour la visualisation et l'analyse des performances.

— **Le chapitre 4** : présente les résultats expérimentaux et propose une analyse visuelle des courbes de loss pour chaque modèle et chaque dataset. Cette analyse permet d'identifier les comportements d'overfitting, underfitting, convergence ou instabilité, et d'en tirer des recommandations pratiques.

Chapitre 1

Les réseaux de neurones

1.1 Introduction

Le domaine de l'intelligence artificielle vise à développer des machines capables de reproduire certaines capacités de l'intelligence humaine. L'un de ses sous-domaines, l'apprentissage automatique (ML), donne aux machines la capacité d'apprendre à partir de données sans qu'elles soient programmées directement. Parmi les méthodes les plus avancées du l'apprentissage automatique on trouve l'apprentissage profond (DL), basé sur les réseaux de neurones artificiels (ANN).

Dans ce chapitre, nous présenterons les fondamentaux des réseaux de neurones, leur fonctionnement, les différents types , les techniques d'apprentissage ainsi que leurs diverses applications.

1.2 Définition

Un réseau de neurones artificiels (ANN) est un modèle informatique inspiré du fonctionnement biologique du cerveau humain [3], conçu pour traiter l'information de manière analogue aux neurones naturels. Ce système adaptatif donne la possibilité aux algorithmes de tirer des données et d'améliorer leur efficacité grâce à un processus d'optimisation basé sur les erreurs commises. Un réseau de neurones est constitué de :

— Couche d'entrée : reçoit les données comme les pixels, mots ou fréquences. Son rôle se limite à transmettre ces informations au réseau sans effectuer de traitement, tout en les formatant pour le système.

— Couches cachées : centres de traitement où s'effectuent les calculs complexes. Leur nombre détermine la capacité d'analyse : plus le réseau contient de couches, plus il est capable d'analyser finement les données complexes [14].

— Couche de sortie : regroupe l'ensemble de ces calculs pour produire un résultat final (décision, score, prédiction).

— Fonctions d'activation : elles introduisent des non-linéarités essentielles pour l'apprentissage [23].

1.3 Du neurone réel au neurone artificiel

À la base, le neurone artificiel a été créé pour reproduire le fonctionnement d'un neurone biologique en s'inspirant de sa structure et de ses mécanismes de transmission de l'information.

La figure 1.1 présente une comparaison entre le neurone biologique et le neurone artificiel.

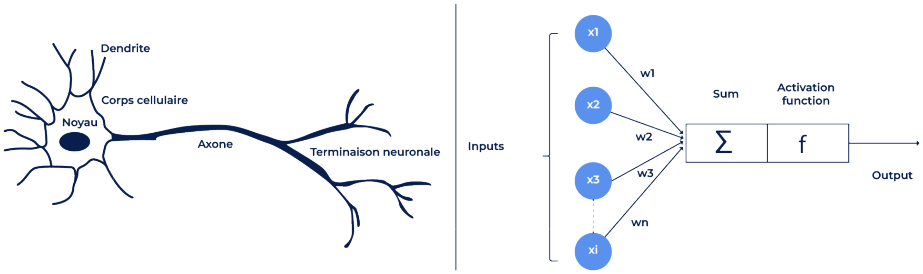


FIGURE 1.1 – Comparaison entre neurone biologique et neurone artificiel

Le tableau 1.1 montre la comparaison entre les neurones biologiques et les neurones artificiels, avec leurs éléments et leurs rôles.

Neurone biologique	**Neurone artificiel**
Synapses (Terminaisons neuronales) : Transmettent le signal à un autre neurone ou à une cellule cible .	Poids synaptiques : Pondèrent l'importance de chaque entrée. Valeurs ajustées pendant l'apprentissage
Axone : Transporte le signal électrique du corps cellulaire jusqu'aux terminaisons neuronales.	Sortie : Transmet le résultat (sortie activée) au neurone suivant dans le réseau.
Soma (Cellule) : Intègre les signaux reçus par les dendrites.	Somme pondérée : Combine les entrées et les poids (calcul de la somme pondérée).
Noyau : Situé dans le corps cellulaire, il contient l' ADN du neurone et contrôle ses fonctions vitales.	Fonction d'activation : Applique une transformation non linéaire pour décider de l'activation du neurone.
Dendrites : Reçoivent les signaux en provenance d'autres neurones.	Entrées : Reçoivent les données d'entrée (valeurs numériques).

TABLE 1.1 – L'analogie entre les neurones biologiques et les neurones artificiels, inspiré de [26]

1.3.1 Composition d'un neurone artificiel

Un neurone artificiel est une unité de traitement inspirée du neurone biologique. Il est composé des éléments suivants [26] :

— **Les entrées** E : elles proviennent soit d'autres neurones du réseau, soit directement de l'environnement (données d'entrée).

— **Les poids** W : ils déterminent l'influence ou l'importance de chaque entrée dans le calcul global.

— **Le biais (bias input)** : une entrée artificielle, associée à un poids, permettant d'ajuster dynamiquement le seuil d'activation du neurone.

— **La fonction de combinaison** s : elle effectue une somme pondérée des entrées, définie par la formule :

$$s = \sum_{i=1}^{n} W_i E_i \tag{1.1}$$

où :

— W_i : poids associé à la i-ème entrée,

— E_i : valeur de la i-ème entrée.

— **La fonction de transfert** f : elle applique une transformation (souvent non linéaire) sur la valeur s pour produire la sortie du neurone :

$$S = f(s) \tag{1.2}$$

La figure 1.2 illustre les différents éléments qui composent un neurone artificiel.

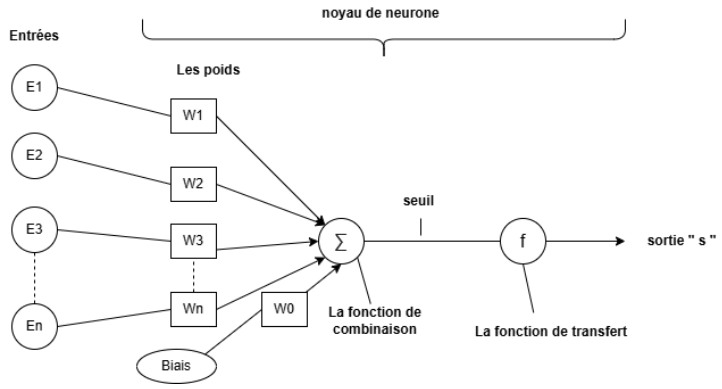


FIGURE 1.2 – Neurone artificiel

— La fonction de transfert calcule la sortie du neurone en fonction de la combinaison en entrée.

Le tableau 1.3 présente les différents types de fonctions de transfert utilisées dans les neurones artificiels :

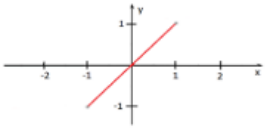
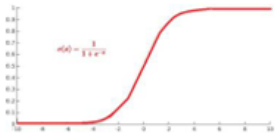
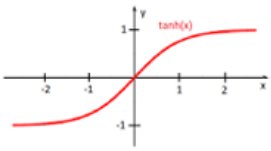
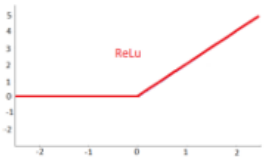
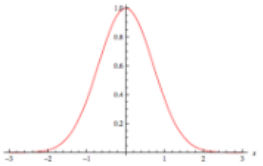
Fonction	Equation	Représentation graphique
Heaviside (Echelon)	$f(x) = \begin{cases} 0 \; si \;\; x < 0 \\ 1 \;\; si \;\; x \geq 0 \end{cases}$	
Rampe (Linéaire)	$f(x) = x$	
Sigmoïde	$\sigma(x) = \frac{1}{1+e^{-x}}$	
Tangente hyperbolique	$f(x) = \tanh(x)$	
Unité de rectification linéaire (ReLU)	$f(x) = \begin{cases} 0 \; si \;\; x < 0 \\ x \;\; si \;\; x \geq 0 \end{cases}$	
Gaussienne	$f(x) = e^{-x^2}$	

FIGURE 1.3 – Les fonctions de transfert

1.4 Fonctionnement d'un réseau de neurones

Le fonctionnement d'un réseau de neurones peut être divisé en plusieurs étapes fondamentales :

1.4.1 Propagation avant (Forward Propagation)

La propagation avant (forward propagation) est un processus essentiel dans lequel les données traversent le réseau de neurones couche après couche. Chaque neurone reçoit des entrées, effectue des calculs pondérés, puis transmet ses résultats, jusqu'à produire une sortie prédictive [15]. Voici les étapes détaillées impliquées dans la propagation avant :

1. **Entrée des données**

   Les données d'entrée **x** (features) sont reçues par la couche d'entrée du réseau.

2. **Calcul des activations dans les couches cachées**

   À chaque couche l, chaque neurone j effectue :

   — **Somme pondérée** : réalise une combinaison linéaire des activations précédentes pondérées par les poids, à laquelle on ajoute un biais.

$$z_j^{(l)} = \sum_i w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \tag{1.3}$$

   où :

   — $z_j^{(l)}$: somme pondérée des entrées reçues par le neurone j de la couche l.

   — $a_i^{(l-1)}$: activation (ou sortie) du neurone i de la couche précédente $l-1$.

   — $w_{ij}^{(l)}$: poids synaptique reliant le neurone i de la couche $l-1$ au neurone j de la couche l .

   — $b_j^{(l)}$: biais associé au neurone j de la couche l .

   — **Fonction d'activation** : introduit de la non-linéarité dans le modèle, essentielle pour apprendre des relations complexes entre les données.

$$a_j^{(l)} = f(z_j^{(l)}) \tag{1.4}$$

   où :

   — $a_j^{(l)}$: activation finale du neurone j à la couche l

   — f : fonction d'activation (comme ReLU, Sigmoïde ou Tanh), appliquée à $z_j^{(l)}$ pour introduire de la non-linéarité .

3. **Propagation à travers les couches**

   Les activations de chaque couche sont utilisées comme entrées pour la couche suivante.

4. **Sortie du réseau**

   La sortie finale $\hat{y}$ est obtenue après passage par la dernière couche :

$$\hat{y} = a^{(L)} \tag{1.5}$$

   où L est la couche de sortie.

   Le choix de la fonction d'activation dépend du type de tâche :

   — Pour la classification : fonction d'activation *sigmoïde* ou *softmax*.

   — Pour la régression : fonction d'activation linéaire.

1.4.2 Calcul de la fonction de coût (Loss Function)

Après avoir obtenu $\hat{y}$, l'erreur est mesurée en utilisant une fonction de coût :

— **Erreur quadratique moyenne** : Cette fonction de coût est utilisée pour les problèmes de régression, où le modèle doit prédire une valeur continue.

$$\mathcal{L}(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}_i - y_i)^2 \tag{1.6}$$

où :

— m : nombre total d'exemples d'apprentissage ;
— $\hat{y}_i$: prédiction du modèle pour l'exemple i ;
— y_i : valeur réelle (attendue) pour l'exemple i.

— **Entropie croisée** : Cette fonction est utilisée dans les tâches de classification, notamment pour la classification binaire ou multi-classes avec Softmax.

$$\mathcal{L}(\hat{y}, y) = -\sum_i y_i \log(\hat{y}_i) \tag{1.7}$$

où :

— y_i : variable réelle (valeur binaire ou vecteur one-hot) indiquant si l'exemple appartient à la classe i ;
— $\hat{y}_i$: probabilité prédite que l'exemple appartienne à la classe i.

1.4.3 Rétropropagation (Backpropagation)

La rétropropagation calcule les gradients nécessaires permet de mesurer l'influence de chaque poids du réseau sur cette erreur. Ce calcul repose sur l'application de la règle de la chaîne. Le gradient peut être décomposé comme suit :

$$\frac{\partial L}{\partial w_{ij}^{(l)}} = \frac{\partial L}{\partial a_j^{(l)}} \cdot \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \cdot \frac{\partial z_j^{(l)}}{\partial w_{ij}^{(l)}} \tag{1.8}$$

avec :
— $z_j^{(l)} = \sum_i w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}$: somme pondérée à l'entrée du neurone j,
— $a_j^{(l)} = f(z_j^{(l)})$: activation du neurone via une fonction f (ReLU, sigmoïde, etc.).

1.4.4 Mise à jour des poids (Weight Update)

Les poids sont mis à jour pour minimiser la fonction de coût, à l'aide de techniques telles que : descente de gradient stochastique (SGD). Cette méthode repose sur la mise à jour itérative des poids selon la formule suivante :

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial L}{\partial w_{ij}^{(l)}} \tag{1.9}$$

où :

— η est le taux d'apprentissage, un hyperparamètre qui contrôle la taille des mises à jour,
— $\frac{\partial L}{\partial w_{ij}^{(l)}}$ est le gradient calculé précédemment.

Il existe également des variantes plus avancées comme l'algorithme *Adam*, qui adaptent dynamiquement les taux d'apprentissage pour chaque poids [30].

1.4.5 Répétition

Ces étapes sont répétées à plusieurs reprises au cours de l'entraînement. Le réseau ajuste ses paramètres progressivement à chaque époque, ce qui lui permet d'améliorer ses prédictions.

1.5 Types de perceptron

Il existe deux types différents : perceptron simple et perceptron multicouche.

1.5.1 Perceptron simple

Le perceptron constitue le premier modèle de neurone artificiel proposé, développé par Frank Rosenblatt en 1957. Ce modèle vise à reproduire le fonctionnement des neurones biologiques [29].

Son fonctionnement repose sur trois étapes :

— Chaque entrée est multipliée par un coefficient.

— Les résultats sont additionnés.

— On compare la somme à un seuil : le résultat est 1 si la somme dépasse le seuil, sinon il est 0 [29] .

L'équation mathématique de la sortie y du perceptron se présente comme suit :

$$y = H\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1.10}$$

où :

— H : fonction de Heaviside (fonction d'activation)

— w_i : poids de la connexion pour l'entrée x_i

— x_i : valeur de la i-ème entrée

— n : nombre total d'entrées

— b : biais (terme constant)

— y : sortie binaire (0 ou 1)

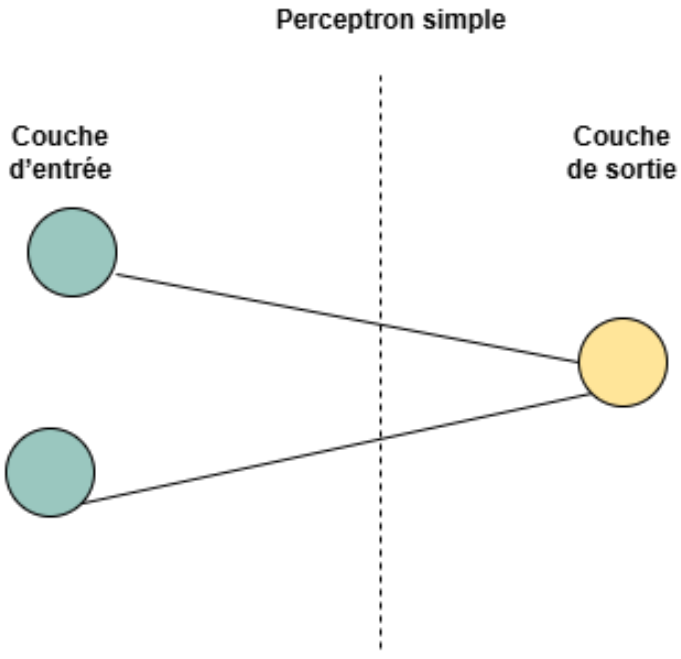


FIGURE 1.4 – Perceptron simple

1.5.2 Perceptron multicouche

Un perceptron multicouche (Multilayer Perceptron, MLP) est un type de réseau neuronal artificiel [25]. Il se compose de :

— D'au moins une couche cachée, permettant une combinaison hiérarchique des caractéristiques.

— De multiples transformations non linéaires via les fonctions d'activation comme ReLU, sigmoïde ou tanh [1]

Contrairement aux méthodes heuristiques du perceptron simple, le MLP s'appuie sur :

— La descente de gradient (optimisation des poids).

— La rétropropagation (calcul efficace des gradients via la règle de la chaîne).

Ce mécanisme permet un ajustement précis des poids couche par couche, en tenant compte de l'erreur globale [30].

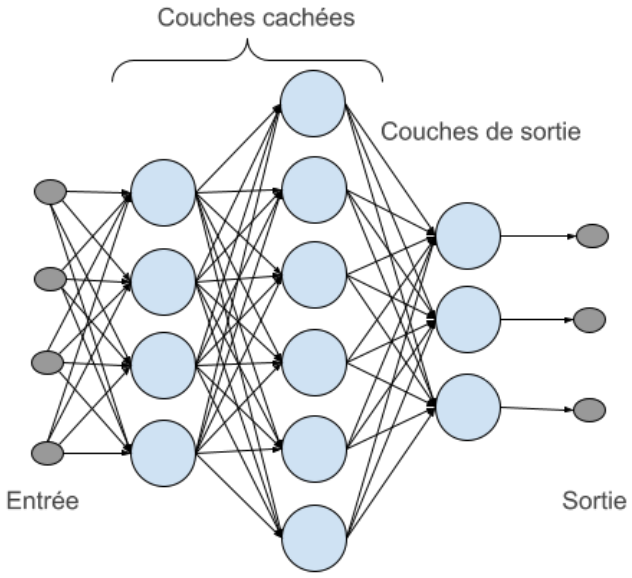


FIGURE 1.5 – Perceptron multicouche

1.6 Les différents types de réseaux de neurones artificiels

Les réseaux de neurones artificiels peuvent être classés selon plusieurs critères, notamment : le nombre de couches séparant les entrées des sorties, le nombre de nœuds cachés dans le modèle et le nombre d'entrées et de sorties par nœud. Parmi les principaux types, on peut citer les trois suivants :

1.6.1 Réseau de neurones feed-forward

Les réseaux neuronaux à propagation avant (Feed-Forward Neural Networks) représentent l'architecture la plus basique des réseaux de neurones. Dans ce type de réseau, l'information circule dans un seul sens : des neurones de la couche d'entrée vers ceux de

la couche de sortie, en passant éventuellement par une ou plusieurs couches cachées, sans rétroaction ni boucle [10].

Chaque neurone d'une couche envoie ses activations à la couche suivante, jusqu'à l'obtention d'une sortie [31]. Cette approche linéaire rend ces réseaux particulièrement adaptés aux tâches de classification ou de régression simples .

Un exemple d'application d'un réseau Feed-Forward

Dataset IRIS : Un exemple classique d'application des réseaux de neurones Feed-Forward est la classification supervisée des fleurs à partir du dataset Iris [13]. Ce jeu de données contient 150 échantillons répartis en trois classes : Setosa, Versicolor et Virginica.

Chaque fleur est décrite à l'aide de quatre mesures :

— la longueur du sépale,

— la largeur du sépale,

— la longueur du pétale,

— la largeur du pétale.

Un réseau de neurones est entraîné à partir de ces caractéristiques afin de prédire la classe de chaque fleur. Ce cas d'usage illustre le fonctionnement d'un réseau Feed-Forward sur un problème de classification multiclasse simple.

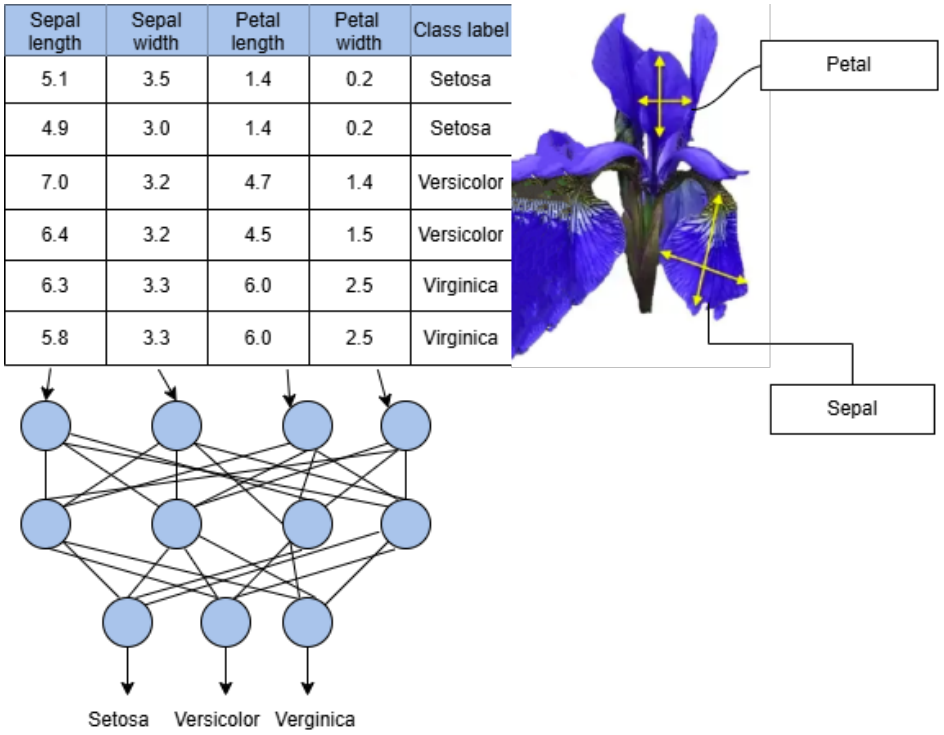


FIGURE 1.6 – Réseau de Neurones avec dataset IRIS, inspiré de [2]

1.6.2 Réseau neuronal convolutif (CNN ou ConvNet)

Les architectures neuronales convolutives (CNN) sont spécifiquement conçues pour traiter des images. Ils détectent avec efficacité les motifs et caractéristiques présents dans les données visuelles.

Leur efficacité résulte de leurs couches de convolution qui analysent et modifient l'entrée pour extraire des caractéristiques tels que les formes, les textures ou les contours. L'architecture d'un CNN est :

— Couche de convolution (CONV) : Le rôle de la couche de convolution (CONV) est d'examiner l'image d'entrée en mettant en disposition des filtres pour détecter des caractéristiques locales importantes, générant par conséquent un ensemble de cartes de caractéristiques (feature maps) qui représentent les composantes de l'image.

— Couche de Pooling (POOL) : Réduit les dimensions spatiales des feature maps tout en conservant les informations essentielles. Les méthodes courantes incluent le max-pooling (valeur maximale) et l'average pooling (la moyenne). À la sortie, le nombre de cartes de caractéristiques reste identique, mais elles sont considérablement compressées [19].

— Couche d'activation ReLU : cette couche remplace les valeurs négatives par zéro, rendant le modèle non linéaire et plus performant pour détecter des motifs complexes.

— Couche Fully Connected (FC) : La dernière couche du CNN, elle connecte tous les neurones de sortie. Elle applique une combinaison linéaire suivie d'une fonction activation pour classifier l'image, renvoyant un vecteur de probabilités pour chaque classe .

Un exemple d'application d'un réseau convolutif

Dataset MNIST : Un autre exemple d'application des réseaux de neurones est la classification d'images issues du dataset MNIST [17].

Dans ce cas, un réseau de neurones convolutionnel (CNN) est utilisé pour analyser une image manuscrite du chiffre "2", représentée en niveaux de gris au format 28×28 pixels. Le réseau extrait automatiquement des caractéristiques visuelles à l'aide de filtres de convolution et de max-pooling, avant de transmettre ces informations à des couches fully-connected avec la fonction d'activation ReLU, afin de prédire la classe de l'image.

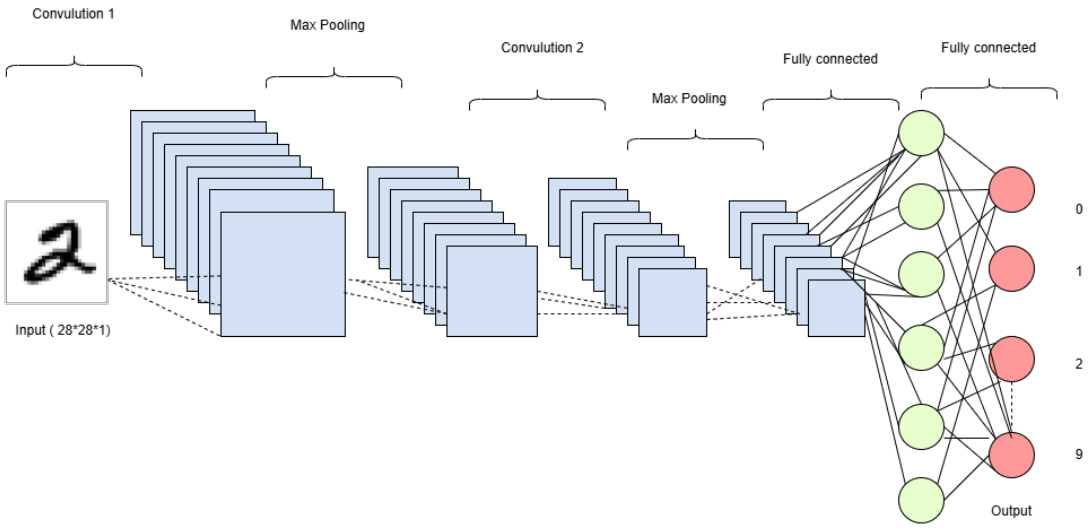


FIGURE 1.7 – Réseau de neurones avec dataset MNIST, inspiré de [39]

1.6.3 Réseau neuronal récurrent (RNN)

Les réseaux de neurones récurrents (RNN) sont une classe de réseaux de neurones artificiels conçus pour traiter des données séquentielles (le texte, le son) [28]. Grâce à leur architecture unique, ils ont la capacité de prendre en compte non seulement l'entrée actuelle, mais aussi les entrées précédentes, leur conférant une capacité à "se souvenir" des informations d'une étape à l'autre. Ils sont idéaux pour des tâches telles que la traduction automatique ou la reconnaissance vocale.

Un exemple d'application d'un réseau récurrent

Un exemple classique d'application est la classification de sentiments dans des phrases. Le dataset utilisé contient des phrases réparties en trois classes :

— **Positif (Classe A)** : phrases exprimant de la satisfaction ou de la joie ;

— **Négatif (Classe B)** : phrases exprimant un avis défavorable ;

— **Neutre (Classe C)** : phrases informatives ou factuelles.

Chaque phrase est traitée mot par mot, et l'état caché du RNN est mis à jour à chaque étape. À la fin de la séquence, le modèle prédit la classe émotionnelle correspondante.

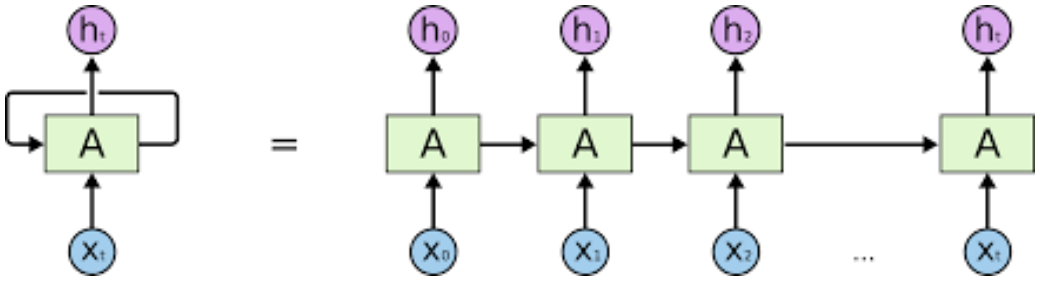


FIGURE 1.8 – Architecture d'un réseau récurrent déroulé dans le temps

La figure 1.8 montre le fonctionnement d'un RNN déroulé dans le temps. Chaque cellule reçoit une entrée (mot), ainsi que l'état précédent, et transmet l'information à l'étape suivante, permettant au modèle de capturer le contexte de la phrase.

1.7 Apprentissage des réseaux de neurones

L'apprentissage est une étape du développement d'un réseau de neurones dans laquelle le comportement du réseau est modifié jusqu'à obtenir le comportement souhaité. On dénombre plusieurs types de règles d'apprentissage, classables en quatre catégories : supervisé, non supervisé, par renforcement ou semi-supervisé.

1.7.1 Apprentissage supervisé

L'apprentissage supervisé est une méthode d'entraînement dans laquelle un modèle apprend à partir de données d'entrée associées à des sorties (étiquettes) connues. Le but est de prédire correctement les sorties pour de nouvelles données, en minimisant une fonction de perte qui mesure l'écart entre les sorties prédites et les sorties réelles [12].

La figure 1.9 illustre le processus d'apprentissage supervisé. Le réseau prend des entrées et génère une sortie obtenue, qui est comparée à la sortie désirée fournie par un superviseur. La différence entre les deux est calculé sous forme d'une erreur. Le signal d'erreur ainsi généré est ensuite utilisé pour mettre à jour les paramètres du réseau lors de l'apprentissage.

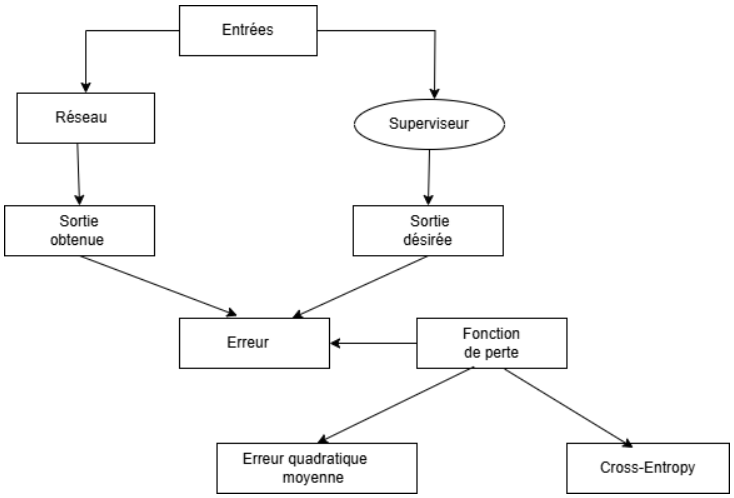


FIGURE 1.9 – Apprentissage supervisé, inspiré de [26]

.

1.7.2 Apprentissage non supervisé

L'apprentissage non supervisé est une approche dans laquelle un modèle est entraîné uniquement à partir de données non étiquetées, c'est-à-dire sans indication explicite de la sortie attendue [12].

La figure 1.10 illustre le principe de l'apprentissage non supervisé. Le modèle reçoit des entrées non étiquetées, les traite à travers un réseau neuronal ou un algorithme non supervisé, et produit une sortie obtenue. Contrairement à l'apprentissage supervisé, aucun superviseur n'intervient, et aucune sortie cible n'est utilisée pour guider l'apprentissage.

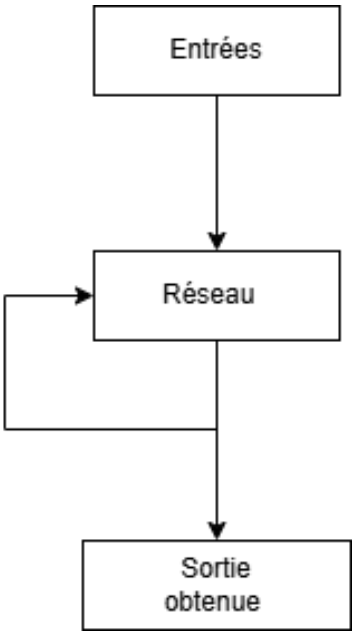


FIGURE 1.10 – Apprentissage non supervisé, inspiré de [26]

1.7.3 Apprentissage par renforcement

L'apprentissage par renforcement est une méthode d'apprentissage dans lequel un agent apprend à prendre des décisions en interagissant avec un environnement[16]. L'agent reçoit des récompenses en fonction des actions qu'il effectue, et il ajuste sa stratégie afin de maximiser la récompense cumulée au fil du temps. Contrairement aux approches supervisées, aucune paire entrée/sortie explicite n'est fournie [16] .

La figure 1.11 présente le fonctionnement de l'apprentissage par renforcement. Un agent interagit avec un environnement : il observe l'état actuel de l'environnement, sélectionne une action, et reçoit en retour une récompense ainsi qu'un nouvel état. Ce cycle se répète, formant une boucle d'apprentissage où l'agent affine sa stratégie de décision afin de maximiser les récompenses à long terme.

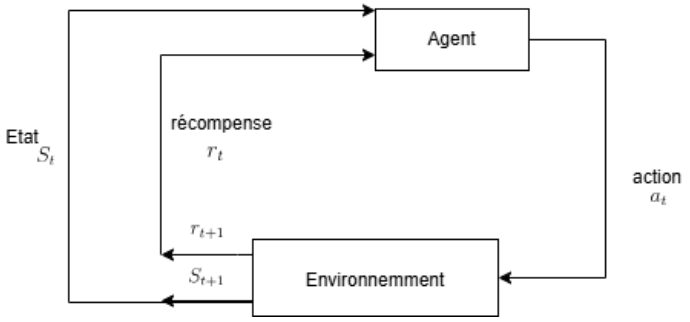


FIGURE 1.11 – Apprentissage par renforcement, inspiré de [26]

1.7.4 Apprentissage semi-supervisé

L'apprentissage semi-supervisé est une méthode d'entraînement qui exploite à la fois un petit ensemble de données étiquetées et un grand ensemble de données non étiquetées. Elle permet d'améliorer les performances du modèle tout en réduisant le coût lié à l'annotation manuelle des données [12].

La figure 1.12 illustre un apprentissage semi-supervisé basé sur le pseudo-étiquetage. Un modèle initial est entraîné à partir d'un petit jeu de données étiquetées, puis utilisé pour attribuer automatiquement des étiquettes aux données non étiquetées. Ces pseudo-étiquettes sont ensuite combinées aux données initiales pour réentraîner le modèle et améliorer sa capacité de généralisation.

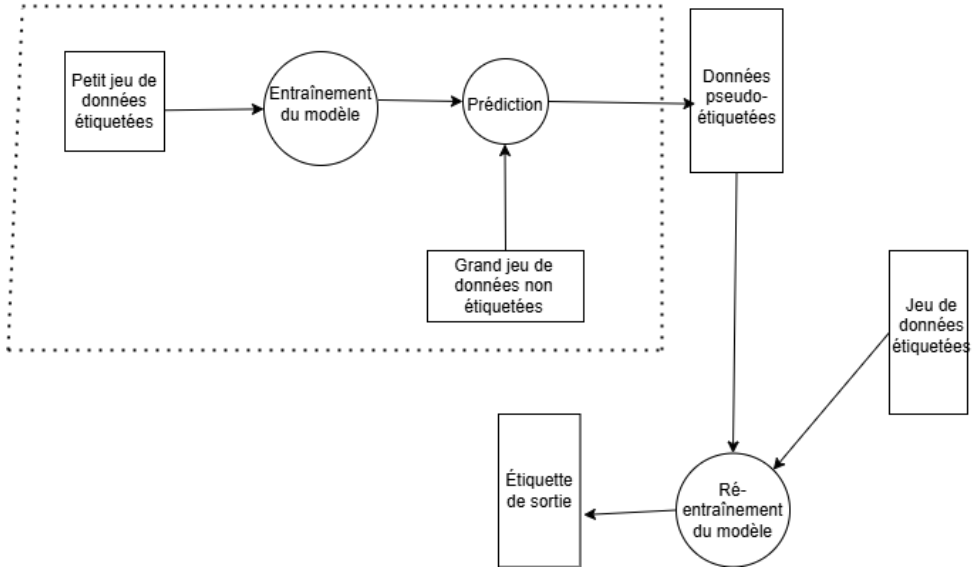


FIGURE 1.12 – Apprentissage semi-supervisé, inspiré de [33]

1.8 Applications des réseaux de neurones

Les réseaux de neurones, grâce à leur capacité à modéliser des relations complexes et à apprendre à partir de grandes quantités de données, sont aujourd'hui au cœur de nombreuses avancées en intelligence artificielle. On retrouve leur utilisation dans une grande variété de domaines. Voici quelques-unes des applications les plus courantes des réseaux de neurones :

— Identification d'objets, de visages, ou de scènes dans des images ou des vidéos.

— Traduction automatique, génération de texte, analyse de sentiments, et chatbots.

— Systèmes de recommandation pour des produits, des films, ou de la musique.

— Aide à la détection de maladies à partir d'images médicales comme les radiographies ou les IRM.

— Détection d'objets et prise de décision en temps réel pour la conduite autonome.

— Prévision des marchés financiers, détection de fraudes, et évaluation des risques.

1.9 Les avantages et les inconvénients des réseaux de neurones

Les réseaux de neurones constituent une approche performante pour traiter des problèmes complexes dans divers domaines. Toutefois, comme toute méthode, ils présentent à la fois des avantages et des inconvénients :

1.9.1 Avantages

— Capacité à apprendre à partir des données.
— Utilisable pour plusieurs tâches.
— Traitement rapide de grandes quantités de données.
— Fonctionne même avec des données imparfaites.
— Devient plus précis avec plus de données.

1.9.2 Inconvénients

— Nécessite beaucoup de données et d'informations pour l'entraînemnt.
— L'entraînement peut être lent et coûte beaucoup de temps.
— Difficile à comprendre et à interpréter.
— Une performance qui dépend des paramètres et de l'architecture.
— Risque de bien performer sur les données d'entraînement mais pas sur les nouvelles données.

1.10 Conclusion

Dans ce chapitre, nous avons présenté les bases fondamentales des réseaux de neurones artificiels, leur fonctionnement ainsi que les principales architectures. Dans le chapitre suivant, nous introduirons la notion de graphe ainsi que les réseaux de neurones sur graphe (GNN), en soulignant leurs caractéristiques, leurs modes de représentation et les architectures utilisées dans le cadre de ce travail.

Chapitre 2

Les réseaux de neurones sur graphes

2.1 Introduction

Les réseaux neuronaux graphiques (Graph Neural Networks, GNN) forment une évolution remarquable dans le domaine de l'apprentissage automatique. Ils autorisent le traitement de données structurées sous forme de graphes, contrairement aux méthodes classiques conçues pour les données tabulaires ou séquentielles. En exploitant explicitement les relations entre les entités, les GNN offrent une représentation détaillée de systèmes complexes, tels que les réseaux sociaux ou les interactions moléculaires. Grâce à leur capacité à capturer à la fois les dépendances locales et globales d'un graphe, ils sont particulièrement adaptés à des tâches comme la prédiction de liens ou la détection de communautés, là où les méthodes traditionnelles peinent à prendre en compte la topologie relationnelle [22].

Ce chapitre explore les fondements théoriques des GNN, présente quelques modèles existants et illustre leurs applications.

2.2 Généralités sur les graphes

2.2.1 Définition

Un graphe est une structure de données complexe aidant à démontrer les objets leurs relations et liens entre eux. Il se compose d'un groupe de nœuds (ou sommets) et d'arêtes (ou liens). Les nœuds peuvent représenter différents éléments comme des individus, des endroits ou des objets, tandis que les arêtes définissent les relations entre ces nœuds. Formellement, un graphe G est défini par un couple

$$G = (V, E) \tag{2.1}$$

où :

— $V = \{v_1, ..., v_n\}$ est l'ensemble des nœuds (vertices),

— $E \subseteq V \times V$ est l'ensemble des arêtes (edges).

2.2.2 Formulation mathématique

Pour l'étude et le traitement des graphes, on utilise souvent des représentations matricielles :

— Une **matrice d'adjacence** $A \in \{0,1\}^{n \times n}$ où $A_{ij} = 1$ si $(v_i, v_j) \in E$

— Une **matrice de caractéristiques** $X \in \mathbb{R}^{n \times d}$ pour les attributs des nœuds[21].

Exemple

Considérons que nous avons un graphe simple à 3 nœuds.

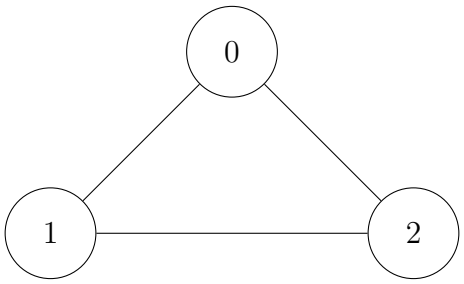


FIGURE 2.1 – Représentation d'un graphe simple à 3 nœuds

La matrice d'adjacence associée à ce graphe est donnée par :

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix} \tag{2.2}$$

Chaque nœud du graphe est également associé à un vecteur de caractéristiques, représenté par la matrice suivante :

$$X = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 1 \end{bmatrix} \tag{2.3}$$

2.2.3 Représentations usuelles

Dans un contexte informatique, les graphes sont généralement manipulés sous forme de structures de données adaptées aux algorithmes. Ces représentations dites usuelles facilitent la lecture, l'implémentation et le traitement des graphes. Parmi les plus courantes, on trouve :

— **Matrice d'adjacence** : sous forme de tableau ou de numpy array.

```
    import numpy as np

A = np.array([
    [0, 1, 1, 0],
    [1, 0, 0, 1],
    [1, 0, 0, 1],
    [0, 1, 1, 0]
])
```

Listing 2.1 – Matrice d'adjacence

— **Liste d'adjacence** : chaque nœud est associé à la liste de ses voisins.

```
adj_list = {
    1: [2, 3],
    2: [1, 4],
    3: [1, 4],
    4: [2, 3]
}
```

Listing 2.2 – Liste d'adjacence

— **Liste d'arêtes** : ensemble des paires de nœuds connectés.

```
    edge_list = [(1, 2), (1, 3), (2, 4), (3, 4)]
```

Listing 2.3 – Liste d'arêtes

2.2.4 Types des graphes

On distingue plusieurs types de graphes, parmi lesquels les plus courants sont les graphes dirigés et non dirigés, ainsi que les graphes homogènes et hétérogènes [37].

1. Graphes orientés et non orientés :

Les graphes dirigés (ou orientés) sont des graphes dans lesquels chaque arête possède une direction, représentée par une flèche allant d'un sommet de départ (la queue) vers un sommet d'arrivée (la tête). Ce type de représentation est essentiel lorsque la relation entre les noeuds n'est pas symétrique. Par exemple, un réseau routier à sens unique [27].

Sur le plan formel, un graphe dirigé est défini par un couple :

$$G = (V, E) \tag{2.4}$$

où :

— V est l'ensemble des sommets (ou nœuds),

— $E \subseteq V \times V$ est l'ensemble des arêtes orientées, représentées par des *paires ordonnées* (u, v), ce qui signifie qu'il existe une arête allant de u vers v, mais pas nécessairement dans l'autre sens ($\{u, v\} \neq \{v, u\}$).

À l'inverse, dans un graphe non dirigé, les arêtes n'ont pas de direction : elles sont représentées par de simples lignes reliant deux sommets, sans indication de point de départ ou d'arrivée. Ce modèle est adapté aux relations symétriques, comme dans un réseau d'amitiés, où si A est ami avec B, alors B l'est aussi avec A [27].

Mathématiquement, un graphe non dirigé est défini par :

$$G = (V, E) \tag{2.5}$$

où :

— V est l'ensemble des sommets,

— $E \subseteq \{\{u, v\} \mid u, v \in V, u \neq v\}$ est l'ensemble des arêtes non orientées, représentées par des *paires non ordonnées* $\{u, v\}$, ce qui signifie que $\{u, v\} = \{v, u\}$.

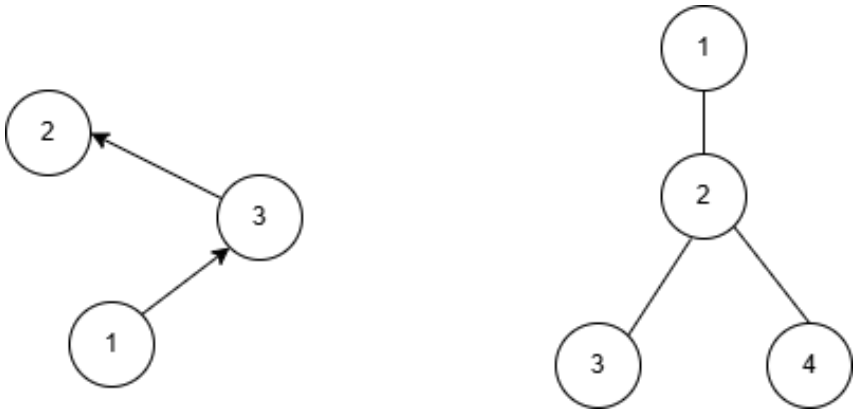


FIGURE 2.2 – Exemple d'un graphe orienté (à gauche) et un graphe non orienté (à droite)

2. Graphes homogènes et hétérogènes :

Un graphe homogène est un graphe dans lequel tous les noeuds (et parfois toutes les arêtes) sont du même type ou possèdent les mêmes propriétés. Cela donne au graphique une structure homogène. Par exemple, dans un réseau social classique, tous les nœuds peuvent représenter des utilisateurs, reliés par des liens d'amitié identiques [24].

D'un point de vue mathématique, un graphe homogène peut être défini comme un triplet :

$$G = (V, E, T) \tag{2.6}$$

où :

— V est l'ensemble des nœuds, tous du même type t_v,

— $E \subseteq V \times V$ est l'ensemble des arêtes, toutes du même type t_e,

— $T = \{t_v, t_e\}$ est l'ensemble des types (un seul type de nœud et un seul type d'arête).

Cependant, un graphe hétérogène contient plusieurs types de nœuds ou d'arêtes, chacun représentant des entités ou des relations distinctes. Un cas classique est un graphe bibliographique, où les entités peuvent représenter des auteurs, des ouvrages et des éditeurs, connectées par des liens tels que « a écrit » ou « publié par ». Ce type de graphe permet de modéliser des systèmes complexes riches en interactions variées [24].

Formellement, un graphe hétérogène est défini par un ensemble de quatre éléments :

$$G = (V, E, \phi, \psi) \tag{2.7}$$

où :

— V : ensemble des nœuds,

— $E \subseteq V \times V$: ensemble des arêtes,

— $\phi : V \to \mathcal{T}_v$: fonction qui attribue un type à chaque nœud,

— $\psi : E \to \mathcal{T}_e$: fonction qui attribue un type à chaque arête,

avec $\mathcal{T}_v$ et $\mathcal{T}_e$ représentant les ensembles des types de nœuds et d'arêtes.

FIGURE 2.3 – Exemple d'un graphe homogène (à gauche) et un graphe hétérogène (à droite), inspiré de [11]

2.2.5 Domaine d'application des graphes

Les graphes sont des outils fondamentaux pour modéliser, représenter et résoudre des problèmes complexes dans une grande variété de domaines, tels que :

— **Réseaux et télécoms** : Les graphes modélisent les amis sur Facebook ou les contacts sur LinkedIn.

— **Transport et logistique** : Les GPS utilisent des graphes pour calculer des itinéraires rapides (Dijkstra, A*).

— **Informatique** : Les bases de données comme Neo4j utilisent des graphes pour des recherches complexes.

— **Biologie et médecine** : On étudie les interactions entre protéines avec des graphes.

— **Jeux vidéo** : Les ennemis dans StarCraft se déplacent avec l'algorithme A*.

— **Sécurité** : Les graphes surveillent les connexions pour bloquer les cyberattaques.

2.3 Réseaux de neurones sur des graphes (GNN)

2.3.1 Définition

Un réseau neuronal graphique (GNN) est un type de réseau neuronal artificiel (ANN) utilisé pour traiter des données représentées sous forme de graphes. Leur capacité à intégrer à la fois la topologie du graphe et les attributs des nœuds et des arêtes en fait un outil puissant pour l'analyse de données relationnelles complexes comme : les réseaux sociaux (liens entre utilisateurs), les molécules (connexions entre atomes) et les systèmes de recommandation (interactions utilisateurs-produits).

2.3.2 Différence entre GNN et CNN

— On utilise les réseaux de neurones convolutifs CNN pour des tâches telles que classification d'images, détection d'objets. Ils sont efficaces avec les données structurées dans des espaces euclidiens, où les informations sont organisées de manière régulière (ex : les pixels d'une image en grille 2D). malgré cela, leur efficacité n'est pas idéale avec des données non structurées comme les données météorologiques et les signaux audio.

— Les GNN sont conçus spécifiquement pour modéliser des données en graphe (structurés et non structurés), ils sont spécialement créés pour les espaces non euclidiens [4] , caractérisés par :

— Une distribution dynamique des nœuds (pas de géométrie fixe)
— Des relations irrégulières (ex : réseaux sociaux, structures moléculaires).

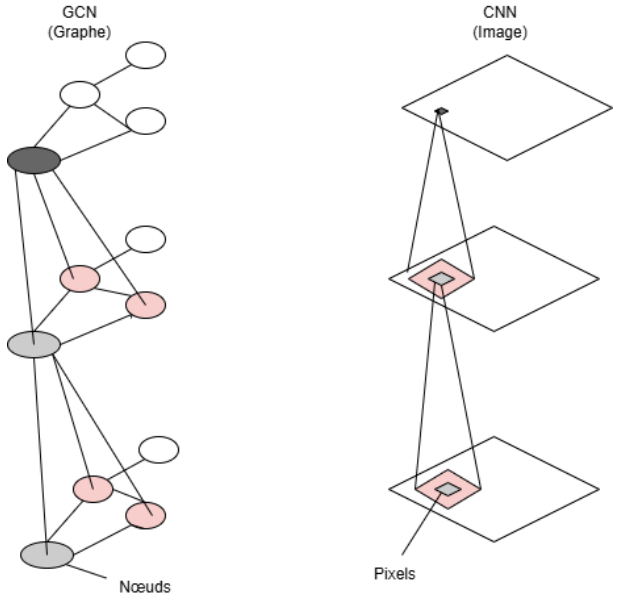


FIGURE 2.4 – Différence entre GNN et CNN, inspiré de [4]

2.3.3 Fonctionnement des GNN

Les réseaux de neurones sur des graphes (GNN) traitent les données en graphes à travers plusieurs étapes :

1. **Initialisation des Noeuds (Node Initialization)** : Chaque nœud $v \in \mathcal{V}$ est initialisé avec un vecteur de caractéristiques $\mathbf{h}_v^{(0)}$, souvent appelé embedding qui peut être défini par les attributs du nœud ou généré aléatoirement.
2. **Passage de Messages (Message Passing)** :
   À chaque couche t, chaque nœud v envoie son vecteur de caractéristiques $\mathbf{h}_v^{(t)}$ à ses voisins $u \in \mathcal{N}(v)$, Cela permet de partager des informations locales à travers le graphe [36].
3. **Agrégation de Messages (Message Aggregation)** : Chaque nœud agrège les messages de ses voisins pour produire un message agrégé $\mathbf{m}_v^{(t)}$. Cela peut se faire par des opérations comme la somme ou le calcul de la moyenne. Formellement, si $\mathcal{M}$ désigne la fonction d'agrégation, on a :

$$\mathbf{m}_v^{(t)} = \mathcal{M}\left(\left\{\mathbf{h}_u^{(t)} \mid u \in \mathcal{N}(v)\right\}\right) \tag{2.8}$$

4. **Mise à Jour des Nœuds (Node Updation)** : Chaque nœud met à jour sa propre représentation en utilisant les informations agrégées. Cela améliore la compréhension du contexte local et global du nœud. Cela se fait généralement avec une transformation linéaire suivie d'une activation :

$$\mathbf{h}_v^{(t+1)} = \sigma\left(\mathbf{W} \cdot \mathbf{m}_v^{(t)} + \mathbf{b}\right) \tag{2.9}$$

   où

— σ est une fonction d'activation (comme ReLU),
— $\mathbf{W}$ est une matrice de poids,
— $\mathbf{b}$ est un vecteur de biais.

Ce processus est généralement appliqué de manière récursive sur plusieurs couches du réseau, permettant ainsi à chaque noeud d'intégrer progressivement l'information provenant de ses voisins à plusieurs sauts dans le graphe.

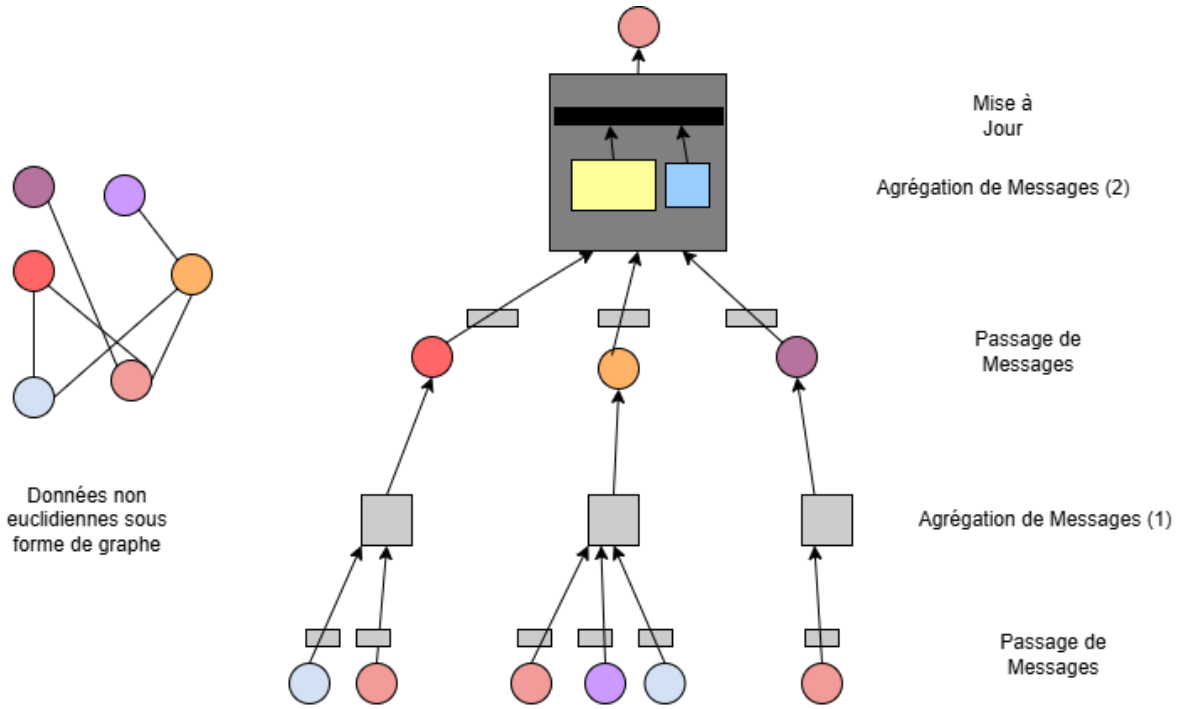


FIGURE 2.5 – Fonctionnement des réseaux neuronaux graphiques, inspiré de [5]

2.3.4 Les principaux types de GNN

Il existe plusieurs types de réseaux de neurones en graphes (GNNs), Voici les principaux types de GNNs :

1. **Graph Convolutional Networks (GCNs) :** Les réseaux neuronaux convolutifs graphiques (GCN) réalisent des convolutions sur les embeddings des nœuds et des arêtes pour regrouper les informations des nœuds voisins . Chaque nœud modifie sa représentation en intégrant les données locales.
   — La formule de mise à jour :

$$h_v^{(l)} = \sigma\left(W^l \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|}\right) \tag{2.10}$$

où :

$h_v^{(l)}$: Représentation du nœud v à la couche l.
W^l : Matrice de poids pour la couche l.
$N(v)$: Ensemble des voisins du nœud v.
$h_u^{(l-1)}$: Représentation du voisin u à la couche précédente $l-1$.
σ : Fonction d'activation non linéaire.

— Message (chaque voisin) :

$$m_u^{(l)} = \frac{1}{|N(v)|} W^{(l)} h_u^{(l-1)} \tag{2.11}$$

— Agrégation : Moyenne des représentations des voisins, suivie d'une transformation linéaire et d'une activation. [35].

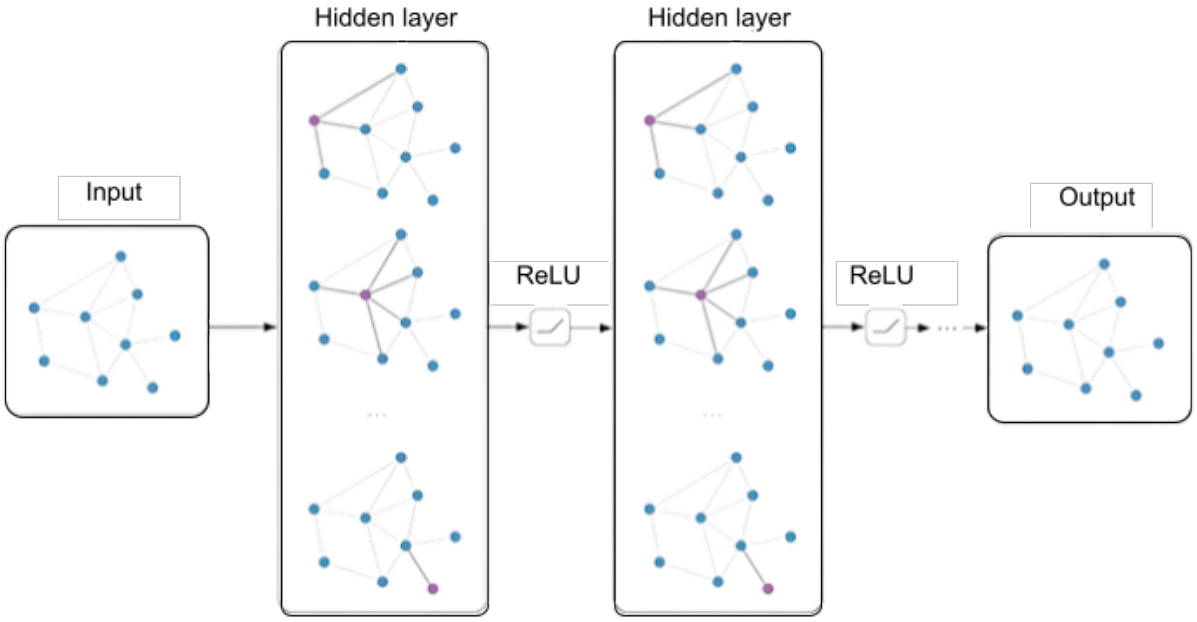


FIGURE 2.6 – Graph Convolutional Networks[8]

Le schéma de la figure 2.6 illustre l'architecture d'un GCN, où un graphe est traité couche par couche. À chaque étape, les nœuds intègrent l'information de leurs voisins selon un mécanisme d'agrégation, suivi d'une fonction d'activation (ReLU).

2. **Graph Attention Networks (GATs)** : Les GATs introduisent un mécanisme pour modifier les contributions des voisins lors de l'agrégation des informations. Au lieu de traiter tous les voisins de manière uniforme, les GATs attribuent des poids d'attention différents à chaque voisin, permettant ainsi de se concentrer sur les relations les plus importantes.

— La formule de mise à jour :

$$h_v^{(l)} = \sigma\left(\sum_{u \in N(v)} \alpha_{vu} W^l h_u^{(l-1)}\right) \tag{2.12}$$

où :

α_{vu} : Coefficient d'attention entre les nœuds v et u.

— α_{uv} est le résultat d'un calcul de coefficients d'attention bruts e_{vu} :

$$e_{vu} = \text{ATT}(W^l h_u^{(l-1)}, W^l h_v^{(l-1)}) \tag{2.13}$$

— Les e_{vu} sont normalisés par softmax [34], pour que $\sum_{u \in N(v)} \alpha_{vu} = 1$

$$\alpha_{uv} = \frac{\exp(e_{vu})}{\sum_{k \in N(v)} \exp(e_{vk})} \tag{2.14}$$

— Le mécanisme d'attention ATT peut être un réseau de neurones simple couche, dont les paramètres sont appris en même temps que W^l [35].

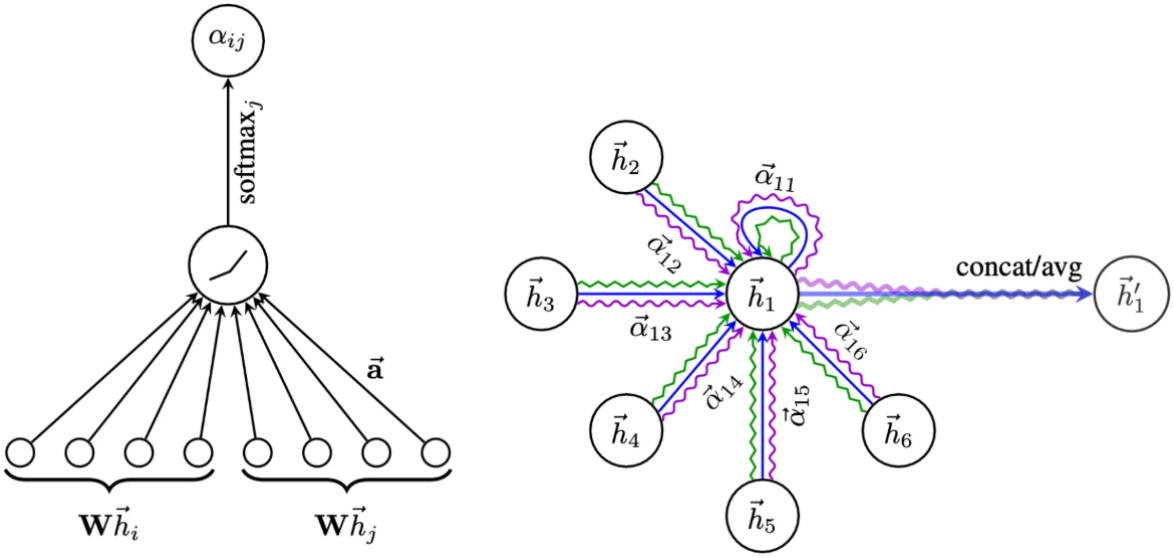


FIGURE 2.7 – À gauche : le mécanisme d'attention. À droite : illustration de l'attention multi-tête appliquée à son voisinage [6].

Le schéma présenté dans la figure 2.7 illustre le mécanisme d'attention utilisé par le modèle GAT, où chaque nœud pondère ses voisins selon leur importance. Ce mécanisme repose sur une transformation linéaire suivie d'une normalisation softmax [6]. L'attention multi-tête, représentée dans la seconde partie, permet de combiner plusieurs vues indépendantes des voisins pour obtenir une représentation finale plus riche du nœud.

3. **GraphSAGE (Sample and Aggregation)** :

GraphSAGE est créé pour organiser des graphes de grande taille en sélectionnant un sous-échantillon des voisins d'un nœud au lieu de prendre en compte l'ensemble de ses voisins. Il présente différentes méthodes d'agrégation, comme la moyenne, l'addition ou les réseaux de neurones complètement connectés, afin de regrouper les données des voisins échantillonnés.

— La formule de mise à jour :

$$h_v^{(l)} = \sigma\left(W^{(l)} \cdot \text{CONCAT}\left(h_v^{(l-1)}, \text{AGG}\left(\{h_u^{(l-1)}, u \in N(v)\}\right)\right)\right) \tag{2.15}$$

où :

AGG : Fonction d'agrégation qui combine les caractéristiques des voisins.

— Message : AGG (variable)

— Agrégation en deux phases :

(a) On agrège le voisinage.
(b) on agrègre avec le nœud lui-même.

AGG peut être réalisée de différentes manières :

— **Moyenne Pondérée** : L'agrégation est réalisée en calculant la moyenne pondérée des caractéristiques des voisins.

$$\text{AGG} = \sum_{u \in N(v)} \frac{h_u^{(l-1)}}{|N(v)|} \tag{2.16}$$

— **Pooling** : Utilisation d'une fonction de pooling, comme la moyenne, appliquée aux caractéristiques des voisins transformées par un perceptron multicouche (MLP) [35].

$$\text{AGG} = \text{Mean}(\text{MLP}(h_u^{(l-1)}), \forall u \in N(v)) \tag{2.17}$$

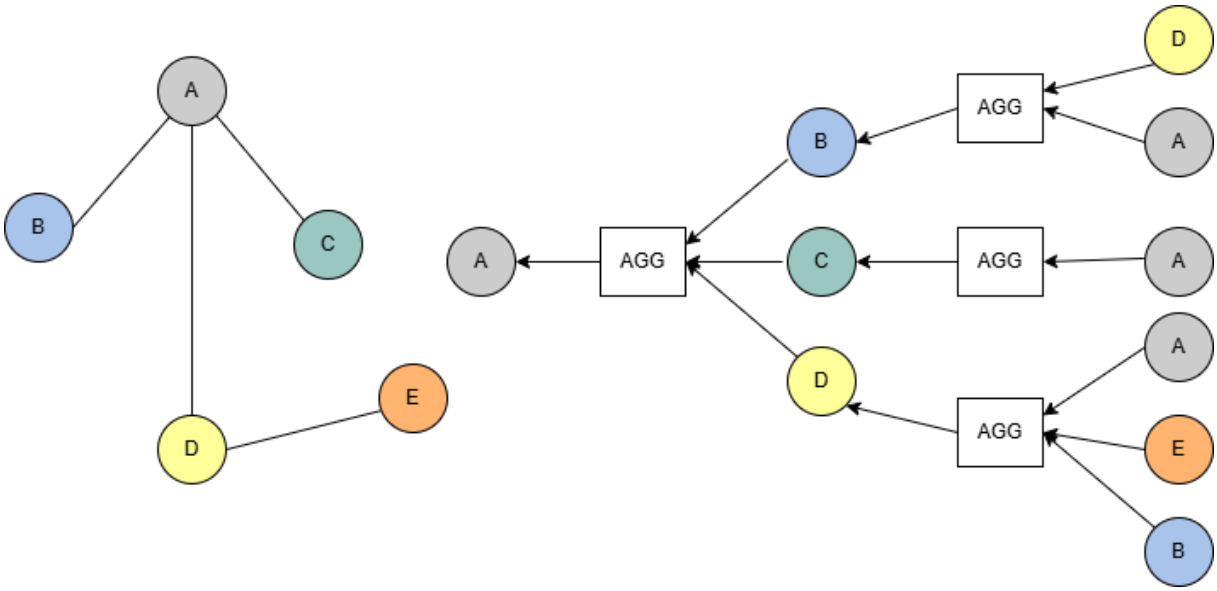


FIGURE 2.8 – Graphe donné (à gauche) et l'architecture Graph-SAGE (à droite), inspiré de [18]

Le schéma présenté dans la figure 2.8 illustre la méthode de GraphSAGE, où chaque nœud met à jour sa représentation en agrégeant les informations de ses voisins à l'aide d'une fonction (moyenne, somme, etc.). Cette agrégation est répétée de manière récursive sur plusieurs couches, permettant d'intégrer l'information de voisinages de plus en plus larges.

4. **Graph Isomorphism Network (GIN)** : GIN est une architecture de réseau de neurones graphiques conçue pour maximiser la capacité de distinction entre les graphes. Proposé par Xu *et al.* (2019), GIN s'inspire directement du test d'isomorphisme de graphes de Weisfeiler-Lehman. Cette approche lui confère une *puissance expressive* supérieure à celle d'autres modèles GNN classiques.

   La mise à jour des représentations des nœuds à chaque couche suit la formule suivante :

$$h_v^{(k)} = \text{MLP}^{(k)} \left((1 + \epsilon^{(k)}) \cdot h_v^{(k-1)} + \sum_{u \in \mathcal{N}(v)} h_u^{(k-1)} \right) \tag{2.18}$$

   où :

   — $h_v^{(k)}$: vecteur de caractéristiques du nœud v à la couche k,

— $\mathcal{N}(v)$: ensemble des voisins du nœud v,
— $\epsilon^{(k)}$: paramètre (fixe ou appris) pondérant la contribution du nœud central,
— $\text{MLP}^{(k)}$: perceptron multicouche appliqué à la somme des représentations.

Cette formule repose sur une *agrégation par somme*, à la différence de GCN ou GraphSAGE qui utilisent des moyennes ou des normalisations. Cette approche permet à GIN de mieux capturer les différences structurelles entre graphes.

L'architecture de GIN se compose généralement de :

— plusieurs couches *GINConv*, chacune suivie d'une activation non linéaire (ReLU ou ELU),
— une fonction d'agrégation globale (somme, moyenne ou max) pour obtenir une représentation du graphe complet,
— un ou plusieurs MLP au sein de chaque couche pour enrichir la transformation des messages,
— des techniques de régularisation telles que le dropout ou la batch normalization.

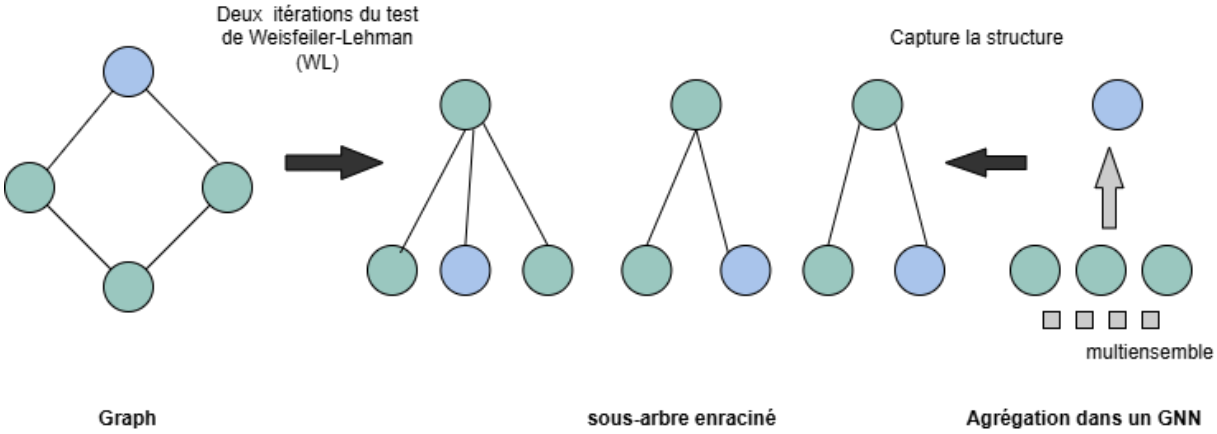


FIGURE 2.9 – Illustration du lien entre agrégation GNN et test de Weisfeiler-Lehman dans le cadre théorique de GIN, inspiré de [38]

Le schéma présenté dans la figure 2.9 montre comment le modèle GIN (Graph Isomorphism Network) s'inspire du test de (WL) pour distinguer les structures de graphes. L'idée est de représenter un nœud à partir des informations de ses voisins, mais en conservant toute la structure locale. Pour cela, GIN utilise une somme au lieu d'une moyenne pour agréger les voisins, ce qui lui permet de prendre en compte le multiensemble des voisins (c'est-à-dire leurs valeurs et leur nombre). Grâce à cette approche, GIN peut capturer des sous-structures complexes et est considéré comme l'un des GNN les plus expressifs, aussi puissant que le test WL.

2.3.5 Applications des GNNs

Les réseaux de neurones graphiques sont désormais indispensables pour le traitement de données structurées sous forme de graphes. Aujourd'hui, les GNNs trouvent des applications dans différentes domaines :

— **Classification des nœuds** : Prédire l'étiquette d'un nœud en exploitant ses relations locales.

  - *Exemple* : Catégorisation de publications académiques (arXiv) ou détection de comptes frauduleux sur les réseaux sociaux.
  - *Datasets* : Cora, PubMed (réseaux de citations), ou Facebook Ego (relations sociales).
- **Prédiction de liens** : Identifier des connexions manquantes ou futures entre nœuds.
  - *Exemple* : Recommandation d'amitié (Facebook) ou prédiction d'interactions protéiques.
  - *Datasets* : Facebook-Live, STRING (interactions protéiques), ou MovieLens (recommandation de films).
- **Détection de communautés** : Grouper des nœuds interconnectés partageant des propriétés communes.
  - *Exemple* : Segmentation d'utilisateurs en groupes d'intérêt (Twitter) ou analyse de réseaux biologiques.
  - *Datasets* : Amazon Co-Purchase (produits associés), DBLP (collaborations académiques).
- **Intégration de graphes** : Représenter des graphes en vecteurs pour des tâches downstream.
  - *Exemple* : Visualisation de réseaux sociaux ou prétraitement pour la recommandation.
  - *Techniques* : node2vec (Facebook), GraphSAGE (Reddit).
- **Génération de graphes** : Créer des graphes réalistes (molécules, réseaux sociaux synthétiques).
  - *Exemple* : Découverte de médicaments (molécules) ou simulation de trafic urbain.
  - *Datasets* : ZINC (molécules), BREC (graphes synthétiques).

2.4 Conclusion

Dans ce chapitre nous avons présenté les Graph Neural Networks (GNN) en soulignant la notion de graphe ainsi que les principales architectures étudiées : GCN, GAT, GraphSAGE et GIN

Le chapitre suivant sera consacré à la mise en œuvre pratique de ces modèles, depuis la préparation des jeux de données jusqu'à l'entraînement, l'optimisation et la visualisation des résultats à l'aide d'un tableau de bord interactif .

Chapitre 3

Implémentation de quelques GNN

3.1 Introduction

Après avoir introduit les réseaux de neurones artificiels (ANN) et détaillé les spécificités des réseaux de neurones sur graphes (GNN), ce chapitre présente la mise en œuvre pratique de plusieurs modèles de GNN. L'objectif est de traduire les concepts théoriques explorés précédemment en applications concrètes à travers une série d'expériences. Ces modèles sont testés sur deux types de tâches différentes :

— **Classification de nœuds** ;

— **Classification de graphes**.

3.2 Modèles utilisés

Dans le cadre de ce projet, quatre architectures de réseaux de neurones sur graphes ont été mises en œuvre pour effectuer les différentes tâches de classification :

— **GCN (Graph Convolutional Network)** : convolution de graphes classique, opérant sur les matrices d'adjacence normalisées.

— **GAT (Graph Attention Network)** : utilise des mécanismes d'attention pour pondérer les voisins.

— **GraphSAGE (Graph Sample and Aggregate)** : apprentissage inductif par agrégation des voisins.

— **GIN (Graph Isomorphism Network)** : agrège les caractéristiques des nœuds voisins via une somme suivie de passages dans un MLP (Multi-Layer Perceptron), sans normalisation.

3.3 Jeux de Données

Dans le cadre de cette analyse, plusieurs jeux de données de référence ont été étudiés. Ces datasets sont associés à diverses tâches d'apprentissage telles que la classification de nœuds et de graphes.

3.3.1 Caractéristiques des jeux de données

Le tableau 3.1 présente les principales caractéristiques des jeux de données utilisés dans nos expériences.

Dataset	**Description**	**Nœuds / Graphes**	**Arêtes**	**Features**	**Classes**
Cora [32]	Articles scientifiques (réseaux de citations)	2708	10556	1433	7
CiteSeer [32]	Publications scientifiques	3327	9104	3703	6
PubMed [20]	Articles biomédicaux sur le diabète	19717	88648	500	3
MUTAG [9]	Graphes représentant des molécules mutagènes	188 (graphes)	Variable	7	2
PROTEINS [7]	Graphes représentant des structures de protéines	1113 (graphes)	Variable	3	2

TABLE 3.1 – Caractéristiques et description des jeux de données utilisés

3.3.2 Nature des graphes : homogénéité, orientation et complexité

Cora, Citeseer et PubMed

— **Homogénéité** :
Ces graphes sont homogènes, c'est-à-dire que tous les nœuds représentent des entités de même nature (documents scientifiques) et toutes les arêtes traduisent un même type de relation (la citation). Il n'existe donc qu'un seul type de nœud et un seul type de lien.

— **Orientation** :
Les arêtes sont orientées, car elles modélisent le sens des citations entre documents (par exemple, un article A cite un article B, ce qui est différent de B citant A). Cette orientation permet de conserver l'asymétrie des relations et de mieux représenter la dynamique des réseaux de publications.

— **Complexité** :
Ces graphes sont considérés comme simples : ils ne contiennent ni boucles (un nœud relié à lui-même), ni arêtes multiples entre les mêmes paires de nœuds. La structure

globale reste relativement régulière et uniforme, ce qui facilite leur traitement par des modèles standards de GNN.

MUTAG et PROTEINS

— **Hétérogénéité** :
Contrairement aux précédents, ces jeux de données sont composés de graphes hétérogènes. Les nœuds peuvent représenter différents types d'éléments (atomes, résidus protéiques) et les arêtes correspondent à des liaisons ou interactions variées. Cette diversité rend la structure plus riche et plus expressive.

— **Orientation** :
Les graphes sont non orientés : les connexions sont symétriques et il n'existe pas de sens privilégié dans les interactions. Ce choix est cohérent avec la nature des liaisons chimiques ou biologiques, où l'ordre d'interaction n'a pas d'importance.

— **Complexité** :
Ces graphes sont structurellement plus complexes. Ils varient d'un échantillon à l'autre (chaque molécule ou protéine a sa propre structure), et leurs nœuds/arêtes peuvent porter des attributs supplémentaires (type, charge, rôle biologique, etc.). Leur traitement nécessite des modèles capables de gérer une topologie variable et des informations hétérogènes.

3.3.3 Visualisation des jeux de données

Les figures présentées ci-après illustrent des exemples de graphes extraits de chacun des jeux de données utilisés dans ce travail. L'objectif de cette visualisation est d'offrir une représentation intuitive de la structure des graphes, en mettant en évidence les relations entre les entités (nœuds et arêtes) ainsi que les particularités topologiques propres à chaque dataset.

Cette étape de visualisation permet également de mieux analyser la complexité structurelle des données en entrée, ce qui constitue un élément essentiel pour comprendre les performances et les comportements des modèles de GNN appliqués par la suite.

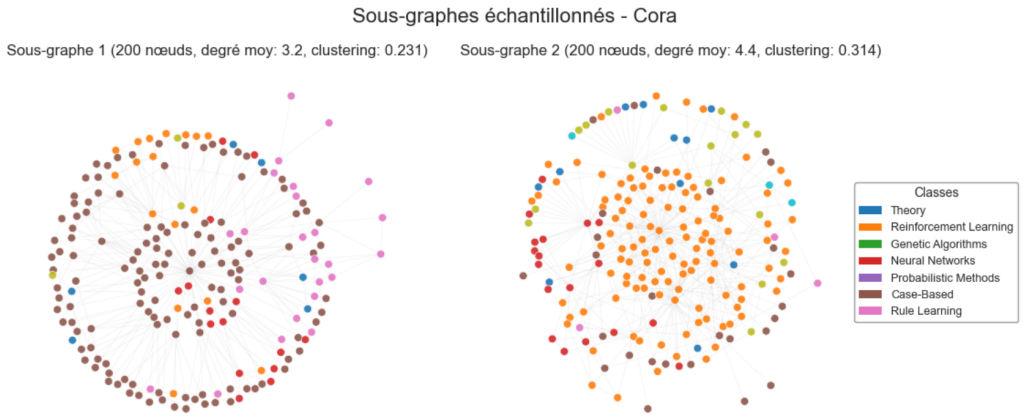


FIGURE 3.1 – Sous-graphes échantillonnés - Cora

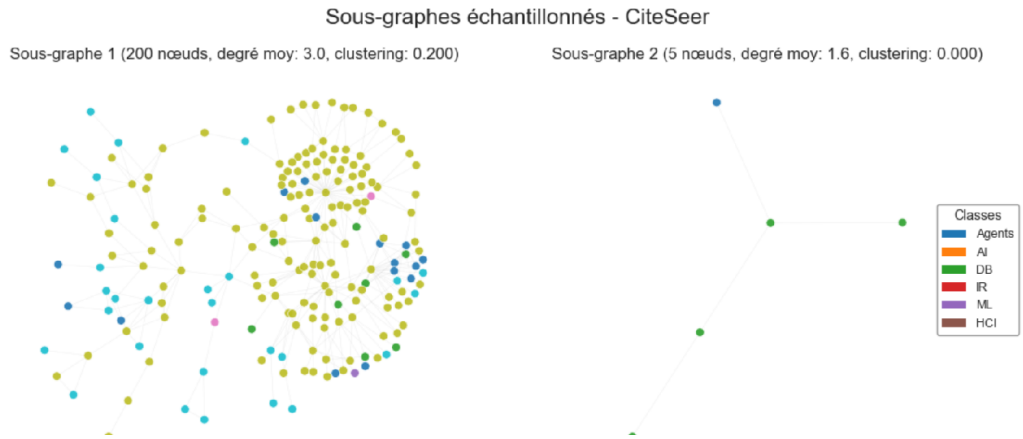


FIGURE 3.2 – Sous-graphes échantillonnés - CiteSeer

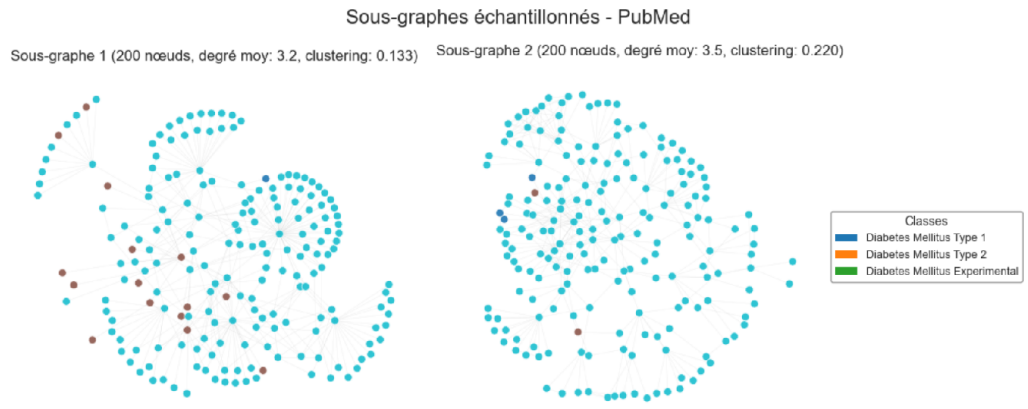


FIGURE 3.3 – Sous-graphes échantillonnés - PubMed

Visualisation des données

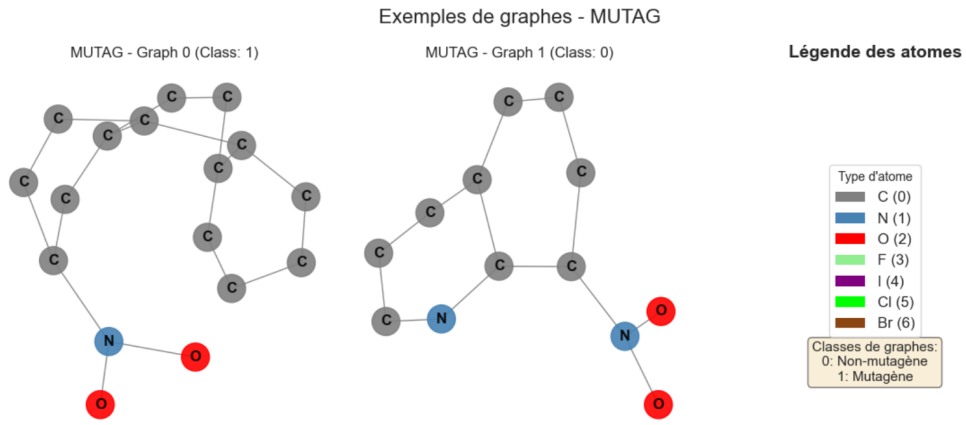


FIGURE 3.4 – Exemples de graphes - MUTAG

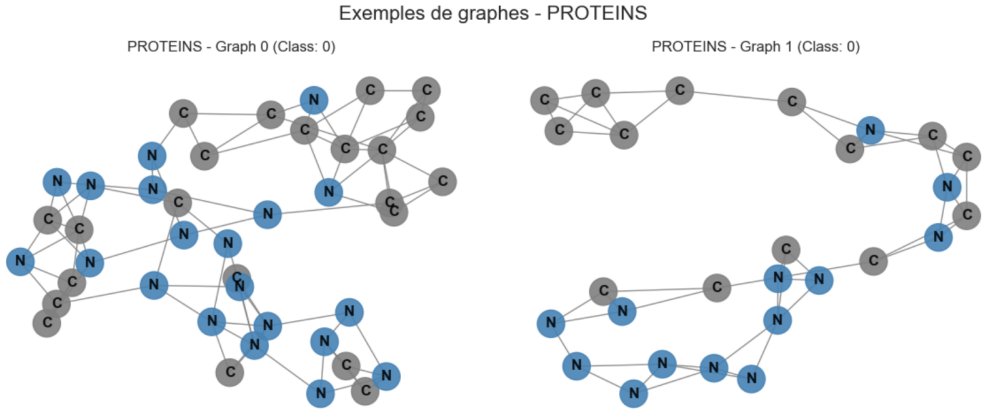


FIGURE 3.5 – Exemples de graphes - PROTEINS

3.4 Langage, environnement et bibliothèques utilisées

Python

Python est un langage de programmation simple, gratuit , très riche avec de nombreuses fonctionnalités et compatible avec tous les systèmes (Windows, Mac, Linux), il est très utilisé en bio-informatique, intelligence artificielle et analyse de données.

Visual Studio Code

Cet éditeur offre une interface flexible, la prise en charge de nombreuses extensions et des fonctionnalités utiles pour le débogage et la gestion de projets Python.

PyTorch

PyTorch est un framework de deep learning utilisé pour l'entraînement des réseaux de neurones, Il propose une gestion efficace des tenseurs, gère les calculs différentiables ,offre la possibilité d'utiliser le GPU.

PyTorch Geometric

PyTorch Geometric est une bibliothèque spécialisée dans le traitement des données graphiques, qui contient des modules dédiés aux Graph Neural Networks (GNNs) , permet une manipulation efficace des structures graphiques.

Optuna

Optuna a été employée pour l'optimisation des hyperparamètres. Elle permet d'automatiser la recherche des configurations optimales à l'aide de stratégies efficaces telles que la recherche bayésienne ou TPE (Tree-structured Parzen Estimator).

Taipy

Utilisée pour le développement d'une interface interactive permettant de visualiser les performances des modèles et de naviguer entre les expériences de manière intuitive.

Matplotlib

Matplotlib est une bibliothèque de visualisation utilisée pour tracer des courbes, des graphiques d'évolution des métriques , facilitant ainsi l'analyse des performances des modèles.

Pandas

Pandas est une bibliothèque utilisée pour l'analyse de données. Elle est particulièrement utile pour l'importation de données , sa classe Data Frame est une méthode excellente pour représenter des données tabulaires.

Scikit-learn

Scikit-learn est une bibliothèque dédiée à l'apprentissage automatique. Elle fournit des outils simples et efficaces pour l'analyse de données et la modélisation prédictive.

3.5 Architecture et organisation de l'implémentation

L'implémentation est structurée en plusieurs modules fonctionnels regroupés dans des scripts (fichiers) séparés. Cette structure modulaire permet une séparation claire entre la configuration, les modèles, l'entraînement, l'évaluation, les outils auxiliaires et les tests expérimentaux.

Fichier	**Rôle principal**
`best_hyperparams_optuna.json`	Fichier JSON contenant les meilleurs hyperparamètres obtenus via Optuna.
`models.py`	Fichier central définissant les architectures des modèles GCN, GAT, GraphSAGE, et GIN.
`train_eval.py`	Script principal gérant l'entraînement et l'évaluation des modèles.
`utils.py`	Contient des fonctions utilitaires pour le prétraitement, l'affichage ou la gestion des métriques.
`GNN_tests.ipynb`	Notebook principal regroupant l'orchestration des expériences et la visualisation.
`final_results.csv`	Fichier CSV synthétisant les résultats finaux d'évaluation des modèles.
`dashboard_taipy.ipynb`	Notebook Jupyter contenant la définition de l'interface interactive du tableau de bord Taipy.
`trained_models/`	Répertoire pour enregistrer les modèles entraînés.
`loss_logs/`	Regroupe les fichiers de suivi des pertes d'entraînement et de validation.
`data/`	Contient les sous-répertoires et fichiers liés aux jeux de données utilisés.
`images/`	Stocke les figures générées (courbes de pertes, performances, etc.).

TABLE 3.2 – Résumé des fichiers du projet et de leurs rôles respectifs

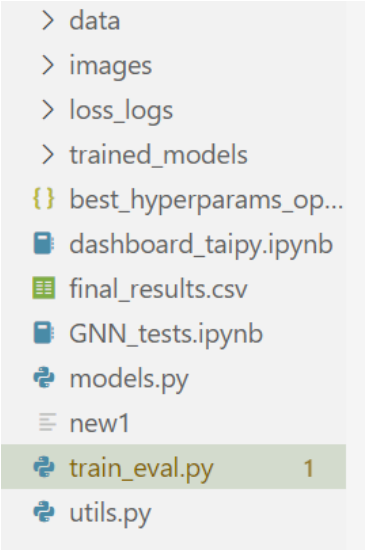


FIGURE 3.6 – Architecture de l'implémentation

3.6 Choix des hyperparamètres d'entraînement

L'optimisation des hyperparamètres constitue une étape essentielle pour assurer des performances optimales des modèles de GNN. Dans le cadre de ce travail, nous avons utilisé Optuna, une bibliothèque d'optimisation automatique basée sur la recherche bayésienne, dans le but de déterminer les configurations les plus efficaces pour chaque combinaison (modèle et dataset). Pour chaque modèle *(GCN, GAT, GraphSAGE, GIN)*, Optuna a exploré plusieurs combinaisons d'hyperparamètres, parmi lesquels :

- Dimension cachée (hidden-dim) : 16, 32, 64
- Nombre de couches (num -layers) : de 1 à 6
- Taux d'apprentissage (lr) : entre 1e-4 et 1e-2 (échelle logarithmique)
- Poids de régularisation (weight -decay) : entre 1e-6 et 1e-3
- Taux de dropout (dropout) : entre 0.0 et 0.6
- Fonction d'activation : ReLU ou ELU
- Nombre de têtes d'attention (heads) : uniquement pour le modèle GAT, entre 1 et 8

  Les modèles ont été entraînés pendant 100 époques en utilisant une taille de lot de 32, et leur performance a été évaluée à l'aide de la précision (accuracy). Les configurations optimales obtenues ont été automatiquement enregistrées pour chaque modèle et jeu de données.

Remarque

Nous avons intentionnellement limité l'étendue des paramètres à explorer en raison de contraintes liées aux ressources de calcul que nous avons utilisées. Pour une meilleure expérimentation, il conviendrait d'augmenter ces étendues. Par exemple, limiter le nombre d'époques à 100 peut être suffisant dans certains cas, mais pas dans d'autres, notamment lorsque les jeux de données, comme *MUTAG*, sont de taille réduite.

3.7 Déroulement d'une expérience

Cette section décrit le processus complet de réalisation d'une expérience, en prenant comme exemple la classification de graphes avec le modèle *GCN* sur le dataset *MUTAG*. L'objectif est de montrer, étape par étape, comment chaque module du projet intervient, depuis la configuration jusqu'à l'évaluation du modèle. Nous présentons ci-dessous chaque étape :

3.7.1 Étape 1 : Chargement des données

La première étape consiste à charger le jeu de données adapté à la tâche visée (classification de nœuds ou de graphes). Les jeux de données proviennent de bibliothèques spécialisées telles que `Planetoid` (pour Cora, CiteSeer, PubMed) ou `TUDataset` (pour MUTAG et PROTEINS). Ces jeux de données sont automatiquement téléchargés, transformés en objets Data ou Batch de PyTorch Geometric, et stockés dans des dictionnaires selon leur type :

```
node_datasets = ['Cora', 'CiteSeer', 'PubMed']
graph_datasets = ['MUTAG', 'PROTEINS']

node_data = {}
for name in node_datasets:
    node_data[name] = Planetoid(root=f'./data/{name}', name=name)

graph_data = {}
for name in graph_datasets:
    graph_data[name] = TUDataset(root=f'./data/{name}', name=name)
```

Listing 3.1 – Chargement des jeux de données

Remarque

L'étape de chargement des données permet de télécharger les datasets depuis leurs sites officiels s'ils ne sont pas déjà téléchargés dans le dossier "/data"

3.7.2 Étape 2 : Visualisation des données

Cette étape permet de présenter et de comprendre comment sont structurés les différents datasets que nous avons utilisés. A cet effet, nous avons écrit plusieurs fonction permettant de :

- de charger les datasets (**load_datasets()**)
- d'afficher la structure des datasets dédiés à la tâche de classification de graphe avec affichage de quelques exemple de graphes (**display_graph_datasets(graph_data)**)
- d'afficher la structure des datasets dédiés à la tâche de classification de noeuds avec affichage de quelques exemple de sous graphes (**display_node_datasets(node_data)**)

```python
from utils import   load_datasets,display_graph_datasets,
    display_node_datasets,
                    display_summary, display_connectivity_stats

node_data, graph_data = load_datasets()
display_graph_datasets(graph_data)
display_node_datasets(node_data)
display_summary(node_data, graph_data)
display_connectivity_stats(node_data)
```

Listing 3.2 – Visualisation des données

Voici un exemple de rendu (ici on montre le rendu d'un seul dataset, mais le code fait la même chose pour tous les datasets) :

Exemple de visualisation des données

```
======================================================================
DATASETS DE CLASSIFICATION DE GRAPHES
======================================================================

==================================================
Dataset: MUTAG (Classification de graphes)
==================================================
Nombre de graphes: 188
Nombre de classes: 2
Nombre de features par nœud: 7
Nombre de features par arête: 4

Statistiques des graphes:
  Nœuds - Min: 10, Max: 28, Moy: 17.9
  Arêtes - Min: 20, Max: 66, Moy: 39.6
```

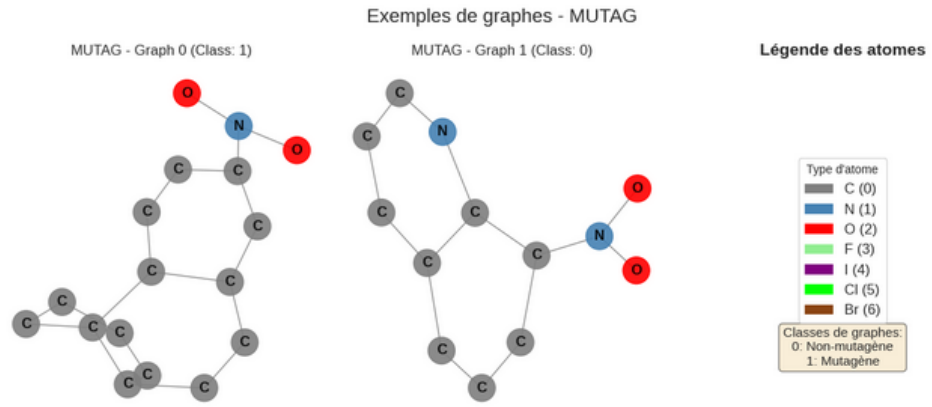


FIGURE 3.7 – Exemple de visualisation de données (dataset MUTAG)

3.7.3 Étape 3 : Définition du modèle

Cette étape consiste à construire l'architecture du modèle GNN. Les modèles comme *GCN, GraphSAGE, GIN ou GAT* sont constitués de couches convolutives sur les graphes, suivies de couches linéaires ou de pooling selon le type de tâche (classification de nœuds ou de graphes). Le modèle *GCN* est défini dans le fichier *models.py*, il applique des convolutions sur graphes en agrégeant les informations des nœuds voisins via des couches successives.

```
class GCN(torch.nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers,
        task_type, dropout=0.5, activation=F.relu):
         super().__init__()
         self.task_type = task_type
         self.activation = activation
         self.dropout = dropout

         self.convs = ModuleList()
         self.convs.append(GCNConv(input_dim, hidden_dim))
         for _ in range(num_layers - 2):
             self.convs.append(GCNConv(hidden_dim, hidden_dim))
         self.convs.append(GCNConv(hidden_dim, output_dim))

    def forward(self, x, edge_index, batch=None):
         for conv in self.convs[:-1]:
             x = self.activation(conv(x, edge_index))
             x = F.dropout(x, p=self.dropout, training=self.training)
         x = self.convs[-1](x, edge_index)
         if self.task_type == 'graph classification':
             x = global_mean_pool(x, batch)
         return x
```

Listing 3.3 – Extrait de code – modèle GCN

3.7.4 Étape 4 : Choix des hyperparamètres

L'optimisation des hyperparamètres est essentielle pour obtenir de bonnes performances. La fonction `objective(...)`, appuyée par la bibliothèque Optuna, permet de sélectionner automatiquement les hyperparamètres du modèle en explorant différents choix (comme `lr, num_layers, hidden_dim,etc.`) via une optimisation bayésienne afin de maximiser les performances sur l'ensemble de validation.

```
 def objective(trial, model_class, model_name, data, input_dim,
    output_dim):

hidden_dim = trial.suggest_categorical("hidden_dim", [16, 32, 64])
num_layers = trial.suggest_int("num_layers", 1, 6)
lr         = trial.suggest_float("lr", 1e-4, 1e-2, log=True)
weight_decay = trial.suggest_float("weight_decay", 1e-6, 1e-3, log=True)
dropout     = trial.suggest_float("dropout", 0.0, 0.6)
activation_name = trial.suggest_categorical("activation", ["relu", "elu"
   ])
activation   = F.relu if activation_name == "relu" else F.elu

    model = create_model(
        model_class,
        model_name,
        trial,
        input_dim,
        hidden_dim,
        output_dim,
        num_layers
    )
```

Listing 3.4 – Extrait de code – Définition des hyperparamètres avec Optuna

Dans le cas du dataset MUTAG, Optuna a été utilisé pour optimiser les hyperparamètres du modèle GCN :

```
},
        "GCN": {
            "graph classification": {
                "hidden_dim": 32,
                "num_layers": 5,
                "lr": 0.005,
                "weight_decay": 0.0003,
                "dropout": 0.2054,
                "activation": "relu",
                "epochs": 100,
                "batch_size": 32
            }
        },
```

Listing 3.5 – Résultats de l'optimisation des hyperparamètres via Optuna

3.7.5 Étape 5 : Entraînement du modèle

L'entraînement est réalisé par la fonction `train(model, optimizer, criterion, data, task type, device)` définie dans le fichier *train eval.py*. Elle prend en entrée le modèle, les données, un optimiseur et une fonction de perte , l'évolution de la perte et de la précision est suivie au fil des époques.

```
def train(model, optimizer, criterion, data, task_type, device):
    model.train()
    optimizer.zero_grad()
    if task_type == 'node classification':
        out = model(data.x.to(device), data.edge_index.to(device))
        loss = criterion(out[data.train_mask], data.y[data.train_mask].
            to(device))
    else:
        out = model(data.x.to(device), data.edge_index.to(device), data.
            batch.to(device))
        loss = criterion(out, data.y.to(device))
    loss.backward()
    optimizer.step()
    return loss.item()
```

Listing 3.6 – Extrait de code – fonctions d'entraînement

Pour MUTAG, le modèle GCN a été entraîné pendant 100 epochs, en utilisant la fonction de perte `CrossEntropyLoss` pour les graphes mutagènes et non mutagènes. La rétropropagation a permis de minimiser cette perte au fil des epochs.

3.7.6 Étape 6 : Évaluation du modèle

Une fois le modèle entraîné, le modèle est évalué sur les données de test via `evaluate(model, data, tasktype, device)` pour mesurer sa capacité de généralisation .

```
def evaluate(model, data, task_type, device):
    model.eval()
    with torch.no_grad():
        if task_type == 'node classification':
            out = model(data.x.to(device), data.edge_index.to(device))
            pred = out.argmax(dim=1)
            correct = (pred[data.test_mask] == data.y[data.test_mask].to
                (device)).sum()
            acc = int(correct) / int(data.test_mask.sum())
        else:
            out = model(data.x.to(device), data.edge_index.to(device),
                data.batch.to(device))
            pred = out.argmax(dim=1)
            acc = accuracy_score(data.y.cpu(), pred.cpu())
    return acc
```

Listing 3.7 – Extrait de code – fonctions d'évaluation

3.7.7 Étape 7 :Visualisation des résultats

Après l'évaluation, les résultats sont visualisés afin de mieux interpréter le comportement du modèle.

Visualisation des résultats

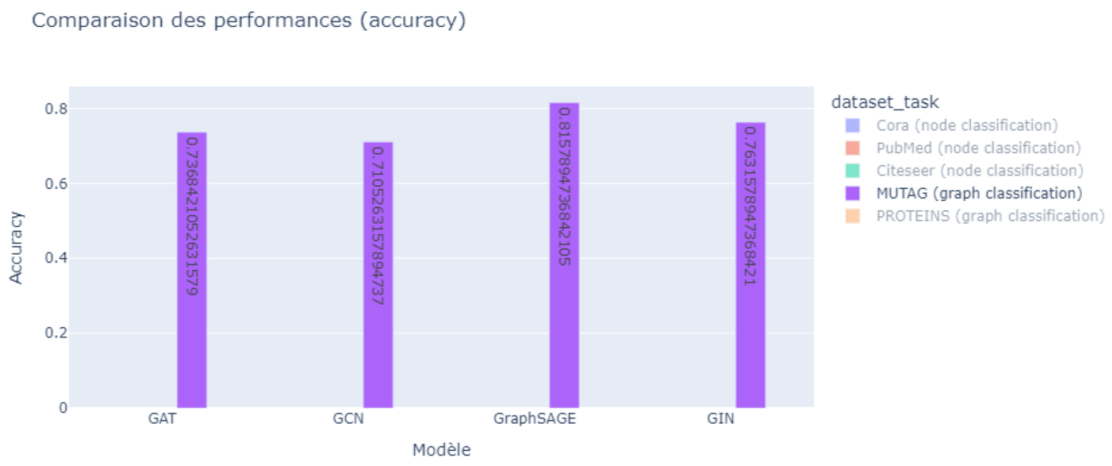


FIGURE 3.8 – Accuracy par modèle et dataset

Remarque

Cette structure expérimentale est appliquée à l'ensemble des expériences menées dans ce projet, qu'il s'agisse de tâches de classification de nœuds ou de classification de graphes , seuls le modèle, le dataset et la tâche changent.

3.8 Tableau de bord interactif

Afin de faciliter l'analyse et la comparaison des résultats obtenus lors des expériences menées sur différents modèles de réseaux de neurones graphiques, un tableau de bord interactif a été développé. Cette interface centralise de manière claire et accessible les performances des modèles, les courbes d'apprentissage ainsi que les hyperparamètres utilisés. L'objectif est d'offrir un outil qui rende l'évaluation des modèles plus intuitive, visuelle et structurée.

3.8.1 Objectifs et utilité

Le tableau de bord a pour but principal de fournir une interface simple et fonctionnelle permettant :

— la visualisation comparative des performances (accuracy) des modèles par dataset et par modèle,

— l'analyse des courbes de perte (train et validation) au fil des époques,

— la consultation des hyperparamètres optimaux trouvés via l'optimisation Optuna.

— Offrir une interface unique pour naviguer entre les différentes analyses expérimentales sans devoir manipuler manuellement les fichiers ou scripts.

3.8.2 Architecture générale du tableau de bord

L'interface du tableau de bord a été développée en Python à l'aide de la bibliothèque Taipy GUI, qui permet de construire des applications web interactives centrées sur les données. Il utilise les fichiers générés lors des expériences d'entraînement des modèles de réseaux de neurones sur graphes (GNN). Trois types de données sont exploités :

— **Performances finales** (`final_results.csv`) : Ce fichier contient l'accuracy obtenue pour chaque combinaison modèle/dataset, à la fin de l'entraînement.

— **Courbes de pertes** (`loss_logs`) : Pour chaque couple (modèle, dataset), un fichier CSV enregistre l'évolution des pertes d'entraînement et de validation à chaque époque.

— **Hyperparamètres optimaux** (`best_hyperparams_optuna.json`) : Ce fichier JSON stocke les meilleurs hyperparamètres trouvés automatiquement via la librairie Optuna.

Sur le plan technique, l'architecture repose sur plusieurs éléments :

— Le chargement automatique des données avec Pandas.

— Des variables d'état dynamiques (`selected_model` ou `selected_dataset`), qui permettent à l'interface de s'adapter aux choix de l'utilisateur.

— Des fonctions de rappel (`on_change, on_menu`) qui assurent la mise à jour des données affichées.

— Une organisation modulaire en pages (en Markdown enrichi), chacune dédiée à un aspect spécifique : performance par dataset, par modèle, comparaison globale, visualisation des courbes de loss, hyperparamètres optimisés .

3.8.3 Étapes de création du tableau de bord

La construction de cette interface suit plusieurs étapes logiques que nous décrivons ci-dessous :

— **Chargement des bibliothèques** : Cette étape permet de charger les bibliothèques essentielles pour : la lecture et la manipulation des données (pandas, json), la visualisation des résultats (matplotlib) ainsi que la création de l'interface graphique (taipy.gui).

Une fois importées, ces bibliothèques offrent plusieurs fonctions utiles pour construire le tableau de bord, notamment :

— `Gui` : pour créer et gérer l'interface graphique avec Taipy

— `navigate` : pour changer de page dans le tableau de bord

```
import os
import json
import pandas as pd
import matplotlib.pyplot as plt
from taipy.gui import Gui, navigate
```

Listing 3.8 – Extrait de code – Chargement des bibliothèques

— **Génération des courbes de loss** : Cette étape consiste à générer des courbes illustrant l'évolution de la loss (fonction de perte) au fil des époques d'entraînement.

```
def plot_all_loss(output_path="all_losses.png"):
    fig, axes = plt.subplots(len(datasets), len(models),
                             figsize=(4 * len(models), 3 * len(
                                datasets)),
                             sharey='col')
    ylims = {}
    for model in models:
        losses = []
        for dataset in datasets:
            path = f"loss_logs/{model}_{dataset}_losses.csv"
            if os.path.exists(path):
                df = pd.read_csv(path)
                losses.extend(df[['train_loss', 'val_loss']].values
                    .flatten())
        if losses:
            ylims[model] = (min(losses), max(losses))
```

Listing 3.9 – Extrait de code – Génération des courbes de loss

Dans cette étape, plusieurs fonctions sont utilisées pour générer et afficher les courbes de loss à partir des fichiers de résultats, parmi lesquelles :

- `plt.subplots(...)` : crée une grille d'axes pour afficher plusieurs courbes.
- `os.path.exists(...)` : vérifie si un fichier existe.
- `pd.read_csv(...)` : lit les fichiers .csv contenant les valeurs de loss.
- `ax.plot(...)` : trace les courbes.
- `fig.savefig(...)` : sauvegarde le graphique complet en image pour affichage dans Taipy.

— **Chargement des résultats de performance** : Dans cette étape, les résultats de performance (accuracy) des différents modèles sur plusieurs jeux de données sont chargés .

```
final_results = pd.read_csv("final_results.csv")
datasets = sorted(final_results['dataset'].unique())
models = sorted(final_results['model'].unique())

plot_all_loss("all_losses.png")
```

Listing 3.10 – Extrait de code – Chargement des résultats de performance

Pour charger et organiser les résultats de performance, certaines fonctions sont essentielles, notamment :

- `pd.read_csv(...)` : lit le fichier contenant les scores .
- `unique()` : extrait les noms uniques des datasets et modèles.
- `sorted(...)` : tri les noms pour les menus déroulants.

— **Chargement et transformation des hyperparamètres** : Dans cette étape, les meilleurs hyperparamètres, identifiés automatiquement par l'outil Optuna lors de l'entraînement des modèles, sont extraits.

```
with open("best_hyperparams_optuna.json", "r") as f:
    hyperparams = json.load(f)


couleurs = ["#1f77b4", "#ff7f0e", "#2ca02c", "#d62728", "#9467bd",
   "#8c564b"]


records = []
for dataset, model_dict in hyperparams.items():
    for model, task_dict in model_dict.items():
        for task, params in task_dict.items():
            record = {"dataset": dataset, "model": model, "task":
               task}
            record.update(params)
            records.append(record)

hyperparams_df = pd.DataFrame(records)
```

Listing 3.11 – Extrait de code –Chargement et transformation des hyperparamètres

L'extraction des hyperparamètres repose sur l'utilisation de certaines fonctions clés, comme :

- `json.load(...)` : charge le fichier JSON.
- `pd.DataFrame(...)` : convertit les hyperparamètres en tableau pour affichage.

— **Définir les variables d'état** : Dans cette étape, les états initiaux sont définis afin de sélectionner par défaut un jeu de données et un modèle lors du lancement du tableau de bord. Cette initialisation s'appuie sur quelques fonctions comme `final_results[final_results.dataset == ...]` qui filtre les données selon la sélection.

```
page = "perf_dataset"
selected_dataset = datasets[0]
selected_model = models[0]
selected_dataset_loss = datasets[0]
selected_model_loss = models[0]

final_results_dataset = final_results[final_results.dataset ==
   selected_dataset]
```

Listing 3.12 – Extrait de code –Définition des variables d'état

— **Préparation des graphiques** : Dans cette étape, les données sont préparées afin de générer un barplot groupé permettant de comparer visuellement les performances de tous les modèles sur l'ensemble des jeux de données. La préparation du graphique repose sur l'utilisation de certaines fonctions clés, notamment :

- `pivot(...)` : transforme le tableau pour que chaque colonne représente un dataset.
- Dictionnaire `chart_all_perf` : spécifie la configuration d'un graphe barplot.

```
pivot_perf = final_results.pivot(index='model', columns='dataset',
    values='accuracy').reset_index()

chart_all_perf = {
    "type": "bar",
    "x": "model",
    "barmode": "group",
    "height": 500,
    "title": "Accuracy par modele et dataset "
}
```

Listing 3.13 – Extrait de code –Préparation des graphiques

— **Fonctions de rappel (callbacks)** : Dans cette étape, les données affichées dans le tableau de bord sont mises à jour automatiquement dès qu'un utilisateur modifie la sélection du jeu de données ou du modèle .

La mise à jour dynamique des données affichées repose sur certaines fonctions clés, telles que :

— `on_change` : mettre à jour les données affichées

— `on_menu(...)` : permet de naviguer entre les pages du menu.

```
def on_change(state, var_name, var_value):
    if var_name == "selected_dataset":
        state.final_results_dataset = final_results[final_results.
            dataset == state.selected_dataset]
    elif var_name == "selected_model":
        state.final_results_model = final_results[final_results.
            model == state.selected_model]
        state.perf_model_data = state.final_results_model[['dataset
            ', 'accuracy']]
```

Listing 3.14 – Extrait de code – Fonctions de rappel

— **Création des pages de l'interface** : Dans cette étape, un menu interactif est mis en place afin de permettre à l'utilisateur de naviguer entre les différentes pages du tableau de bord. L'interactivité du tableau de bord s'appuie sur plusieurs éléments visuels dynamiques, tels que :

— `<|{selected_dataset}|selector|lov={datasets}|on_change=on_change|>` : menu déroulant dynamique.

— `<|{final_results_dataset}|table|>` : affichage d'un tableau filtré.

— <...|chart|properties=...|>| : graphique généré à partir des données sélectionnées.

— <loses|image|>| : affichage de l'image des losses.

```
menu = [
    ('perf_dataset', 'Dataset Performance'),
    ('perf_model', ' Model Performance'),
    ('all_perf', ' Toutes Perfs'),
    ('all_loss', ' Toutes Loss'),
    ('hyperparams', 'Hyperparametres')
]

main_page = """<|menu|label=Menu|lov={menu}|on_action=on_menu|>"""

perf_dataset_md = """

<|{selected_dataset}|selector|lov={datasets}|dropdown|on_change=
   on_change|>
 <|{selected_dataset}|text|>
<|{final_results_dataset}|table|page_size=5|>
<|{final_results_dataset}|chart|properties={chart_perf_dataset}|>
"""
```

Listing 3.15 – Extrait de code – Création des pages de l'interface

— **Lancement de l'application** : Dans cette étape, toutes les pages du tableau de bord sont regroupées dans un dictionnaire, puis l'application Web est lancée à l'aide de la classe Gui.

Le lancement de l'application repose sur l'utilisation de la classe Gui, à travers certaines fonctions clés telles que :

— `Gui().run()` : lance l'application sur sur un port local.

— `dark_mode=False` : permet d'activer le thème clair.

```
pages = {
    "/": main_page,
    "perf_dataset": perf_dataset_md,
    "perf_model": perf_model_md,
    "all_perf": all_perf_md,
    "all_loss": all_loss_md,
    "hyperparams": hyperparams_md
}



Gui(pages=pages).run(dark_mode=False, port=5002)
```

Listing 3.16 – Extrait de code –Lancement de l'application

3.8.4 Structure de l'interface

Le tableau de bord est organisé autour de plusieurs pages thématiques, accessibles via un menu latéral. Chaque page vise un objectif précis, et les fonctionnalités qui y sont intégrées permettent de répondre concrètement à cet objectif. Nous décrivons ci-dessous les différentes pages qui composent le tableau de bord :

Performance par dataset

Cette page permet à l'utilisateur d'analyser les performances des différents modèles GNN sur un dataset spécifique, à l'aide d'un menu déroulant, d'un tableau récapitulatif des accuracy et d'un graphique en barres illustrant les écarts de performance.

Visualisation des performances par dataset

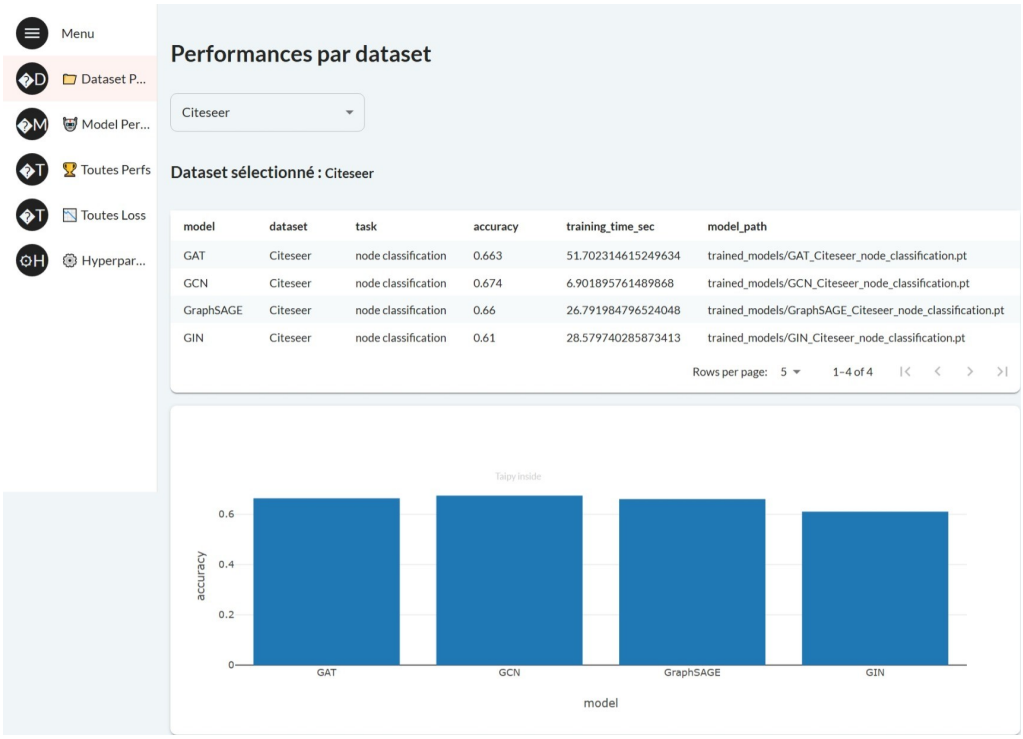


FIGURE 3.9 – Performances des modèles sur un dataset sélectionné

Performance par modèle

Cette page du tableau de bord permet d'observer les performances d'un modèle sur tous les jeux de données, grâce à un menu de sélection, un tableau récapitulatif des accuracy et un graphique comparatif.

Visualisation des performances par modèle

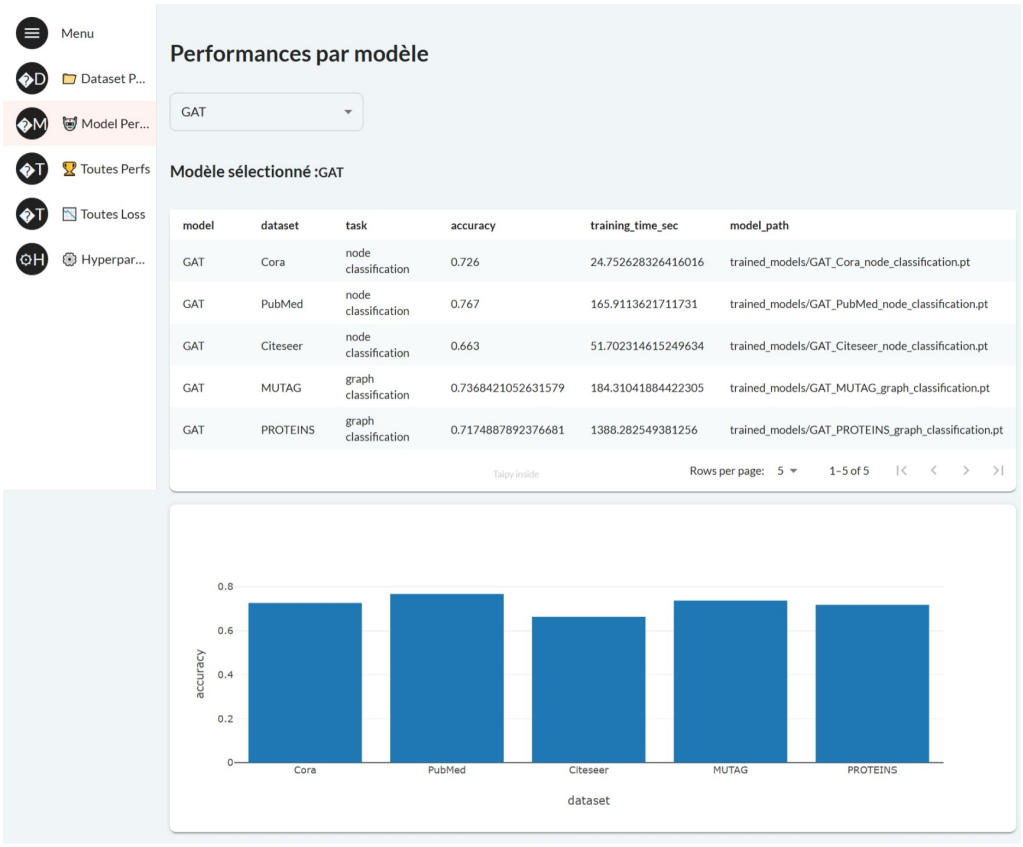


FIGURE 3.10 – Performances du modèle sélectionné sur différents datasets

Toutes les performances

Cette page permet de comparer tous les modèles sur l'ensemble des datasets en une seule visualisation, à l'aide d'un tableau croisé et d'un graphique en barres groupées pour observer les performances globales.

Visualisation des performances pour toutes les expériences

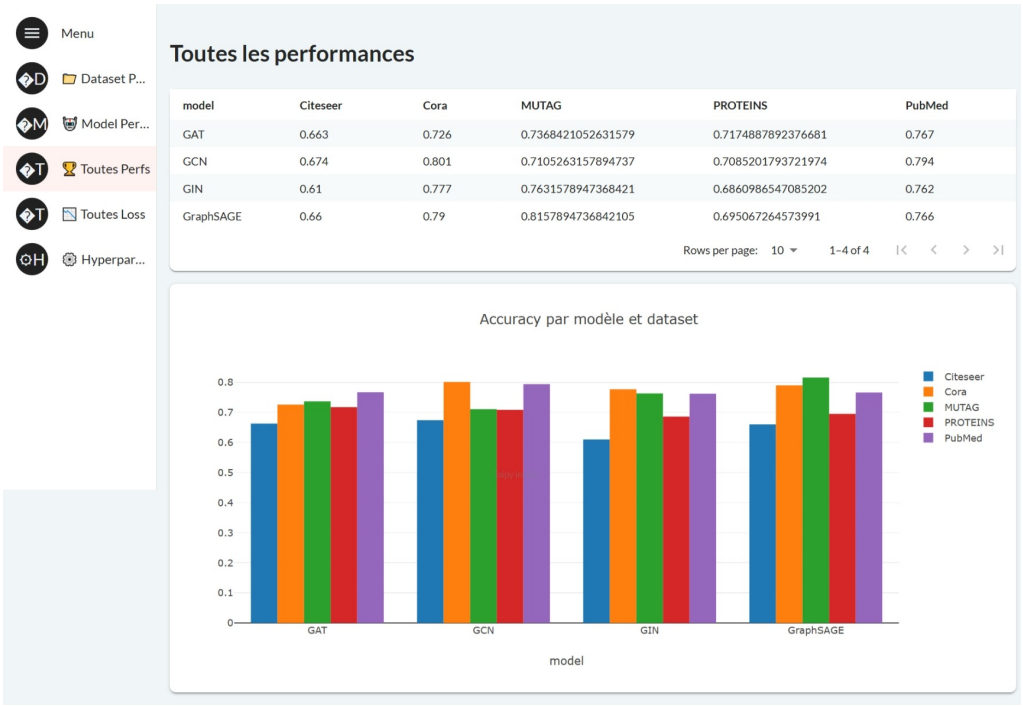


FIGURE 3.11 – Comparaison globale des modèles

Toutes les losses

Cette page permet de visualiser la dynamique d'apprentissage de chaque modèle sur chaque dataset, à travers des courbes de perte (train/val) générées automatiquement et affichées sous forme d'image.

Visualisation des courbes des loss

FIGURE 3.12 – Courbes d'apprentissage (train/val loss) pour tous les modèles et datasets

Hyperparamètres optimaux

Cette page présente les meilleurs hyperparamètres trouvés pour chaque configuration (modèle, dataset, tâche), à partir du fichier JSON généré par Optuna, affichés sous forme de tableau pour une lecture claire.

Visualisation des hyper-paramètres utilisés

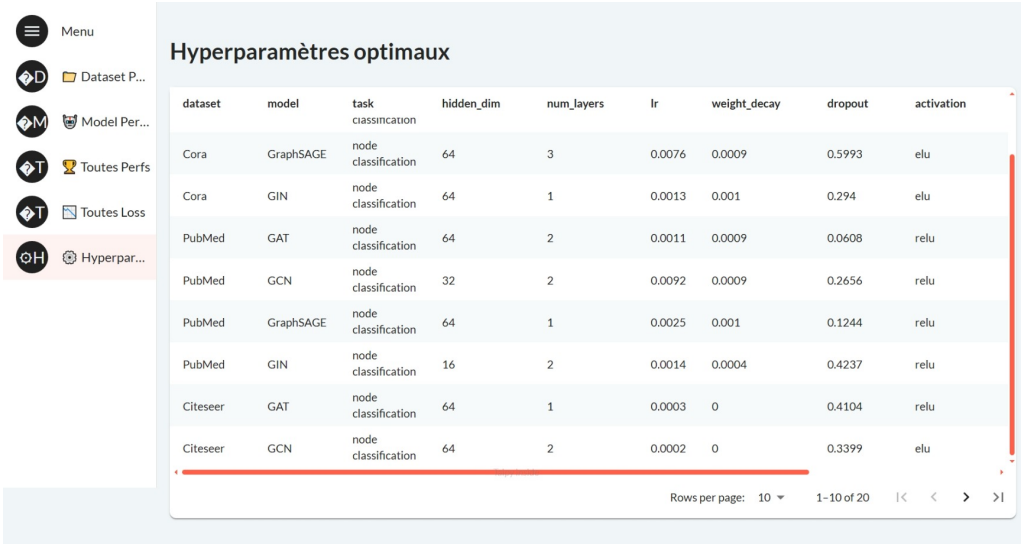


FIGURE 3.13 – Tableau des hyperparamètres optimisés avec Optuna

3.9 Conclusion

Ce chapitre a présenté le cadre technique établi pour la mise en œuvre des modèles de réseaux de neurones sur graphes. En décrivant les outils, bibliothèques, jeux de données et modèles utilisés, il a permis de construire une base solide pour les expérimentations. L'utilisation d'Optuna pour l'optimisation des hyperparamètres, ainsi que celle de Taipy pour la visualisation interactive, aide à mieux organiser et comprendre les résultats. Cette organisation facilite la reproduction des expériences et leur adaptation à d'autres contextes. Elle constitue ainsi une préparation efficace à l'analyse des résultats abordée dans le chapitre suivant.

Chapitre 4

Résultats et discussion

4.1 Introduction

Ce chapitre présente les résultats expérimentaux obtenus lors de l'évaluation des modèles *GCN*, *GAT*, *GIN* et *GraphSAGE* sur cinq jeux de données (Cora, CiteSeer, PubMed, MUTAG et PROTEINS). L'objectif est de comparer les performances, les résultats sont ensuite analysés pour tirer des conclusions sur l'efficacité relative de chaque architecture .

4.2 Présentation des résultats

Dans cette section, nous présentons les résultats obtenus lors de nos expérimentations sur les différents modèles de réseaux de neurones graphiques (GCN, GAT, GraphSAGE, GIN) appliqués aux tâches de classification de nœuds et de classification de graphes. Les performances sont évaluées à l'aide de la métrique d'accuracy, et sont comparées, lorsque cela est possible, aux références issues de la littérature.

4.2.1 Classification de nœuds

L'évaluation des performances des modèles GNN sur la tâche de classification de nœuds a été réalisée sur trois jeux de données standards : Cora, PubMed et Citeseer .

Résultats de nos expériences

Le tableau 4.1 regroupe les résultats d'accuracy obtenus par *GCN, GAT, GraphSAGE* et *GIN* sur les jeux de données : Cora, PubMed et Citeseer.

Modèle	Cora	PubMed	Citeseer
GCN	**0.801**	**0.794**	**0.674**
GAT	0.726	0.767	0.663
GraphSAGE	0.79	0.766	0.66
GIN	0.777	0.762	0.61

TABLE 4.1 – Résultats obtenus pour la classification de nœuds

Sur les trois jeux de données, le GCN se distingue comme le modèle le plus performant dans nos expériences, atteignant (80.1 %) sur Cora et (79.4%) sur PubMed. Le modèle GraphSAGE offre des résultats proches sur Cora (79.0%). Le GAT, malgré ses mécanismes d'attention, obtient des performances inférieures à celles attendues dans la littérature, en particulier sur Cora (72.6%). Quant à GIN, il montre des performances plus modestes, notamment sur Citeseer (61.0%), ce qui suggère que son architecture est moins adaptée aux graphes hétérogènes avec des structures clairsemées.

La figure 4.1 reprend visuellement les résultats du tableau précédent afin de faciliter la comparaison entre les modèles sur chaque dataset.

Résultats

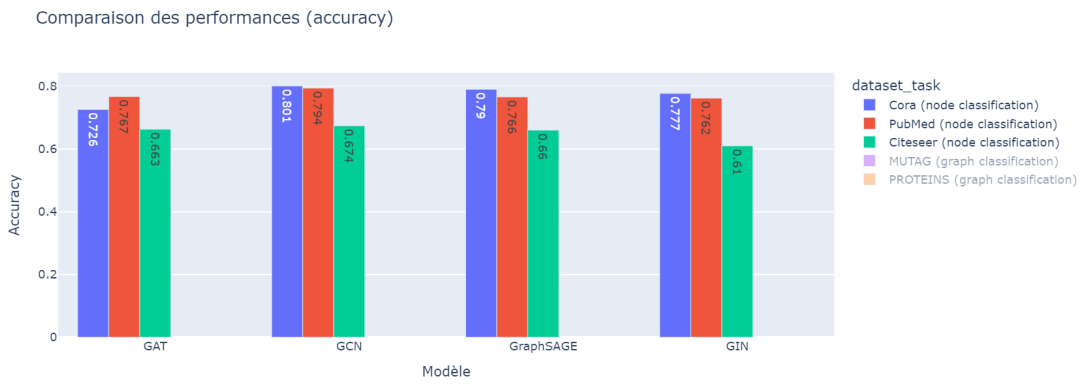


FIGURE 4.1 – Accuracy des modèles sur les jeux de données de classification de nœuds

Comparaison avec l'état de l'art

Dans la littérature, plusieurs variantes améliorées ont été proposées pour augmenter les performances, comme l'ajout de mécanismes d'attention ou l'utilisation de stratégies d'agrégation plus efficaces. Ces versions avancées obtiennent souvent de meilleurs résultats sur des jeux de données classiques comme Cora, PubMed et Citeseer.

Le Tableau 4.2 présente une comparaison des performances obtenues par nos modèles sur les jeux de données Cora, PubMed et Citeseer, en les confrontant aux résultats de l'état de l'art .

Les écarts constatés entre nos résultats et ceux de l'état de l'art s'expliquent principalement par l'utilisation, dans les travaux de référence, de variantes améliorées des modèles

Dataset	Modèle testé	Accuracy	Réf. état de l'art	Accuracy (réf)
Cora	GCN	80.1 %	GCN	81.5 %
	GAT	72.6 %	GAT	83.0 % ± 0.7 %
	GraphSAGE	79.0 %	GraphSAGE (MMA)	85.8 %
	GIN	77.7 %	GIN	87.78 %
PubMed	GCN	79.4 %	GCN	79.0 %
	GAT	76.7 %	GAT	79.0 % ± 0.3 %
	GraphSAGE	76.6 %	GraphSAGE (MMA)	86.0 %
	GIN	76.2 %	Graph-BERT (avec GIN)	79.3 %
Citeseer	GCN	67.4 %	GCN	70.3 %
	GAT	66.3 %	GAT	72.5 % ± 0.7 %
	GraphSAGE	66.0 %	GraphSAGE (MMA)	76.3 %
	GIN	61.0 %	Graph-BERT (avec GIN)	71.2 %

TABLE 4.2 – Comparaison des résultats expérimentaux avec les références de l'état de l'art sur Cora, PubMed et Citeseer.

classiques, souvent combinées avec des techniques de régularisation avancées, des agrégateurs plus complexes (comme *MMA* pour GraphSAGE) ou des modèles hybrides (GIN + *BERT*). Par ailleurs, le tuning d'hyperparamètres et les tailles d'échantillons utilisées dans ces publications jouent un rôle non négligeable.

4.2.2 Classification de graphes (MUTAG, PROTEINS)

L'évaluation des performances des modèles GNN sur la tâche de classification de graphes a été réalisée sur deux jeux de données standards : MUTAG et PROTEINS.

Résultats de nos expériences

Le Tableau 4.3 regroupe les résultats d'accuracy obtenus par GCN, GAT, GraphSAGE et GIN sur les jeux de données : MUTAG et PROTEINS .

Modèle	MUTAG	PROTEINS
GCN	0.711	0.709
GAT	0.737	**0.717**
GraphSAGE	**0.816**	0.695
GIN	0.763	0.686

TABLE 4.3 – Résultats obtenus pour la classification de graphe

Sur le jeu de données MUTAG, GraphSAGE se distingue avec la meilleure performance

(81.6%), suivi de GIN (76.3%). Le modèle GCN obtient le score le plus faible à (71.1%), ce qui montre ses limites sur ce type de données chimiques. Pour PROTEINS, les scores sont plus rapprochés : GAT affiche la meilleure précision (71.7%), légèrement au-dessus de GCN (70.9%). GraphSAGE et GIN montrent des performances plus modestes.

La figure 4.2 offre une représentation visuelle des résultats présentés précédemment, permettant une comparaison plus intuitive entre les modèles pour chaque dataset.

Résultats

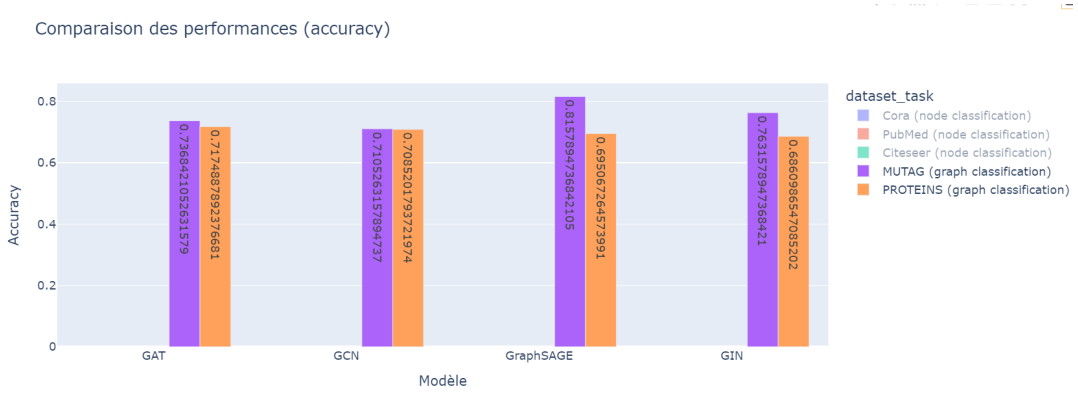


FIGURE 4.2 – Accuracy des modèles sur les jeux de données de classification de graphe

Comparaison avec l'état de l'art

Dans l'état de l'art, certaines méthodes améliorées, comme *PANDA* ou *CIN++*, permettent d'obtenir de meilleurs scores en combinant différentes techniques d'agrégation ou en ajoutant des couches spécifiques d'interaction.

Le Tableau 4.4 présente une comparaison des performances obtenues par nos modèles sur les jeux de données MUTAG et PROTEINS, en les confrontant aux résultats de l'état de l'art .

Dataset	**Modèle**	**Accuracy**	**Réf. état de l'art**	**Accuracy (réf)**
MUTAG	GCN	71.05 %	GCN + PANDA	85.75 %
	GAT	73.7 %	GAT-GC (f-Scaled)	90.44 %
	GraphSAGE	81.6 %	CIN++	94.4 %
	GIN	76.3 %	GIN	85.7 %
PROTEINS	GCN	70.9 %	GCN	75.54 % ± 1.62
	GAT	71.7 %	GAT	76.79 % ± 1.67
	GraphSAGE	69.5 %	GraphSAGE	73.0 %
	GIN	68.6 %	GIN	75.54 % ± 1.85

TABLE 4.4 – Comparaison des résultats expérimentaux avec les performances de l'état de l'art sur les datasets MUTAG et PROTEINS.

Les écarts observés avec l'état de l'art s'expliquent principalement par l'utilisation de variantes sophistiquées des modèles de base. Par exemple, GCN + PANDA intègre des mécanismes d'agrégation hiérarchique plus riches que le GCN standard, et GAT-GC(f-Scaled) propose un mécanisme d'attention multiscalaire plus fin que notre GAT de base. Le modèle CIN++, basé sur GraphSAGE, offre une structure plus profonde et des interactions croisées, ce qui améliore nettement ses performances. Nos modèles, bien que standards, restent compétitifs sur certains jeux, notamment MUTAG avec GraphSAGE.

Remarque

Les modèles utilisés dans ce travail, pour les tâches de classification de nœuds et de graphes, correspondent à des versions développées et simplifiées des architectures standards. Néanmoins, l'optimisation des hyperparamètres à l'aide de la bibliothèque Optuna a permis d'obtenir des performances proches de celles rapportées dans l'état de l'art. Ce résultat met en évidence l'importance du réglage des paramètres, même dans le cadre d'architectures relativement simples.

4.3 Explication des résultats

Cette section a pour objectif d'expliquer les performances observées, en s'appuyant sur les choix de conception des modèles, les caractéristiques des jeux de données utilisés, ainsi que les paramètres d'entraînement adoptés.

4.3.1 Rôle de l'architecture des modèles

Les différences de performance observées entre les modèles GNN peuvent être expliquées par leur architecture spécifique :

— *GCN* s'appuie sur un mécanisme simple de propagation de l'information via des convolutions sur graphes. Cette approche lui confère une bonne stabilité et une rapidité d'entraînement, en particulier sur des graphes homogènes et peu bruités comme Cora ou PubMed. Cependant, sa simplicité peut devenir une limite face à des structures plus complexes, ce qui peut expliquer ses performances plus modestes sur des jeux comme MUTAG ou PROTEINS.

— *GAT* introduit un mécanisme d'attention permettant à chaque nœud de pondérer l'importance de ses voisins. Dans notre expérience, ce modèle a donné de bons résultats sur des graphes bien structurés comme MUTAG, mais s'est révélé moins performant sur des jeux présentant une structure topologique moins marquée, tels que Citeseer ou Cora. En outre, la complexité de ses calculs peut limiter son efficacité sur des graphes de grande taille, tels que PROTEINS.

— *GraphSAGE* utilise une méthode d'échantillonnage et d'agrégation efficace, lui permettant d'apprendre à partir de sous-structures locales. Il s'est révélé particulièrement performant sur MUTAG, et a montré une bonne robustesse sur l'ensemble des jeux de données testés, malgré un score plus faible sur PROTEINS.

— *GIN* se distingue par sa forte expressivité théorique, mais sa sensibilité aux hyperparamètres peut nuire à ses performances. Dans notre cas, ses résultats varient fortement selon les jeux de données : satisfaisants sur MUTAG, plus faibles sur

PROTEINS et Citeseer, ce qui confirme sa dépendance au tuning et à la nature du graphe.

Le tableau ci-dessous propose une synthèse des avantages et des limitations des principaux modèles GNN utilisés dans cette étude.

Modèle	**Points forts**	**Limitations**
GCN	Simple, rapide, stable	Moins flexible, limité sur graphes complexes
GAT	Focalisation contextuelle efficace	Coût de calcul élevé
GraphSAGE	Bonne généralisation, induction locale	Dépend du type d'agrégation
GIN	Théoriquement puissant	Sensible aux hyperparamètres

TABLE 4.5 – Avantages et limites des modèles GNN

4.3.2 Impact du type de tâche

Au-delà de l'architecture, la nature de la tâche à accomplir influence fortement les performances des modèles :

— **classification de nœuds** : dans cette tâche, les modèles opèrent sur un seul grand graphe et doivent prédire la classe de chaque nœud à partir de sa structure locale. Dans ce contexte, GCN s'est montré particulièrement performant dans nos expériences, grâce à sa stabilité et son efficacité sur des graphes citationnels homogènes comme Cora et PubMed. GraphSAGE et GAT obtiennent également de bons résultats, bien que légèrement inférieurs, notamment sur Cora. En revanche, GIN peine à se démarquer, en particulier sur Citeseer, probablement en raison de sa sensibilité au surapprentissage dans des graphes denses et peu structurés.

— **classification de graphes** : ici, chaque exemple est un graphe indépendant (comme une molécule ou une protéine), et l'objectif est de prédire une étiquette globale pour le graphe. GraphSAGE s'avère bien adapté à ce type de tâche, comme le montre sa performance élevée sur MUTAG, grâce à sa capacité à généraliser à partir de sous-structures locales. GIN a également montré de bons résultats, notamment sur MUTAG, tirant parti de son pouvoir expressif pour capturer des motifs complexes. Toutefois, ses performances peuvent varier selon les réglages et la nature du dataset, comme observé sur PROTEINS.

4.3.3 Influence du type de données sur les performances des modèles GNN

L'efficacité des modèles GNN varie selon la structure des graphes utilisés. Des facteurs comme la taille, la densité et l'homogénéité des graphes influencent leur capacité d'apprentissage .

— **Petits graphes à structure simple** : Le jeu de données MUTAG, composé de graphes moléculaires de petite taille, est particulièrement adapté à l'extraction de

motifs structuraux. Dans ce contexte, GraphSAGE a obtenu les meilleures performances dans nos expériences, grâce à sa capacité d'agrégation locale efficace. GIN montre également de bons résultats sur ce type de graphes, bénéficiant de son pouvoir expressif. La petite taille des graphes et leur structure locale bien définie limitent le risque de surapprentissage, rendant inutile l'utilisation de modèles trop profonds.

— **Graphes de grande taille et topologie complexe** : Les ensembles de données tels que Cora, PubMed et Citeseer sont des réseaux de citations étendus, avec des milliers de nœuds et une connectivité importante. Dans ce contexte, GCN s'est montré le plus performant, en particulier sur Cora et PubMed, grâce à sa capacité à propager efficacement l'information dans des graphes homogènes. De son côté, GAT, bien qu'un peu moins performant dans nos expériences, conserve un bon niveau grâce à son mécanisme d'attention, qui lui permet d'ajuster dynamiquement l'influence des voisins. En revanche, GIN souffre de surapprentissage et de sensibilité aux réglages sur ces graphes, notamment sur Citeseer, où ses performances ont été les plus faibles.

— **Effet de l'homogénéité structurelle** : Les graphes relativement homogènes tels que Cora et PubMed, caractérisés par des degrés de nœuds similaires et une structure régulière, favorisent les modèles comme GCN et GAT, qui bénéficient d'une diffusion d'information stable. En revanche, dans des graphes plus hétérogènes comme Citeseer (structure divisée, faible densité) ou PROTEINS (diversité topologique élevée), les performances des modèles sensibles à la structure globale, comme GIN, peuvent chuter. Dans ces cas, GraphSAGE reste compétitif grâce à son approche inductive et sa capacité à s'adapter à la variabilité topologique. De son côté, GAT conserve de bonnes performances en ajustant dynamiquement l'importance attribuée à chaque voisin en fonction du contexte .

4.3.4 Influence des hyperparamètres

Les hyperparamètres ont une grande influence sur les résultats des modèles de graphes. Un bon réglage permet un apprentissage efficace, tandis qu'un mauvais choix peut freiner la performance.

— **Nombre de couches (num_layers)** : Plus de couches permettent de mieux capter la structure du graphe, mais trop de couches peuvent dégrader les performances (sur-lissage).

- Exemple : GCN fonctionne bien avec une seule couche sur Cora, alors que GAT utilise 5 couches grâce à son architecture plus complexe.

— **Dimension cachée (hidden_dim)** : Une grande dimension peut mieux représenter les données, mais risque le sur-apprentissage.

- Exemple : Sur PubMed et MUTAG, une dimension cachée de 64 fonctionne bien pour GAT et GraphSAGE. En revanche, GIN, plus expressif, et GCN obtiennent de bons résultats avec des dimensions plus réduites, telles que 16 ou 32.

— **Taux d'apprentissage (lr)** : Un petit taux assure une convergence stable, un grand taux apprend plus vite mais peut devenir instable.

- Exemple : Sur Citeseer, les bons résultats de GCN et GAT avec un lr très faible (0.0002–0.0003) montrent l'importance de la stabilité.

— **Dropout** : Le dropout est une méthode efficace pour limiter le surapprentissage.
  - Exemple : Des taux élevés (> 0.4) sont observés sur les graphes plus complexes, comme Cora ou Citeseer avec GraphSAGE. En revanche, des valeurs modérées (0.20–0.40) suffisent pour des datasets plus petits, comme MUTAG.

— **Fonction d'activation** : La fonction d'activation introduit la non-linéarité nécessaire à la modélisation de relations complexes .
  - Exemple : GCN et GAT utilisent souvent ReLU. GIN et GraphSAGE utilisent ELU, mieux adapté à certains graphes.

— **Nombre de têtes (spécifique à GAT)** : Le nombre de têtes d'attention permet une agrégation d'information plus riche.
  - Exemple : GAT utilise jusqu'à 7 têtes sur PubMed et MUTAG, ce qui améliore la stabilité et la diversité des représentations.

— **Taille du batch (graph classification)** : Utilisée pour MUTAG et PROTEINS, une taille de 32 rend l'apprentissage plus stable, contrairement à la classification de nœuds qui se fait en batch complet (full-batch).

4.4 Interprétation des courbes de loss

L'analyse des courbes de perte (loss) pour les modèles GIN, GraphSAGE, GCN et GAT, appliqués à cinq jeux de données (CiteSeer, Cora, MUTAG, PROTEINS, PubMed), permet d'évaluer les dynamiques d'apprentissage, la capacité de généralisation des modèles, ainsi que les limitations spécifiques à chaque architecture.

Les courbes en bleu représentent la perte calculée sur les données d'entraînement, tandis que les courbes en rouge correspondent à la perte mesurée sur l'ensemble de validation. Pour chaque dataset, une figure met en évidence le comportement d'un modèle représentatif, accompagnée d'une synthèse comparative des autres modèles.

4.4.1 Analyse des courbes de loss – Cora

La figure 4.3 présente des courbes de loss pour GAT sur le dataset CORA.

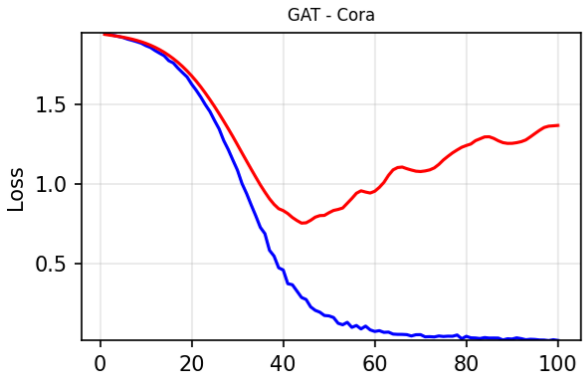


FIGURE 4.3 – Évolution des courbes de loss pour GAT sur le dataset CORA.

La courbe de loss d'entraînement diminue rapidement et de manière régulière, traduisant une bonne progression de l'apprentissage. En parallèle, la courbe de validation diminue dans un premier temps, avant de se stabiliser, puis de remonter légèrement après environ 30 à 40 époques. Cela indique que, bien que le modèle apprenne efficacement sur les données d'entraînement, la remontée progressive de la courbe de validation signale un début de sur-apprentissage.

Dans ce contexte, GAT présente un overfitting modéré sur Cora. De son côté, GCN affiche un bon apprentissage avec des courbes proches et stables. En revanche, GraphSAGE présente une courbe de validation plus fluctuante, traduisant une certaine instabilité. Enfin, GIN converge correctement, mais de manière plus lente que GCN .

4.4.2 Analyse des courbes de loss – MUTAG

La figure présente 4.4 l'évolution des courbes de loss pour GCN sur le dataset MUTAG.

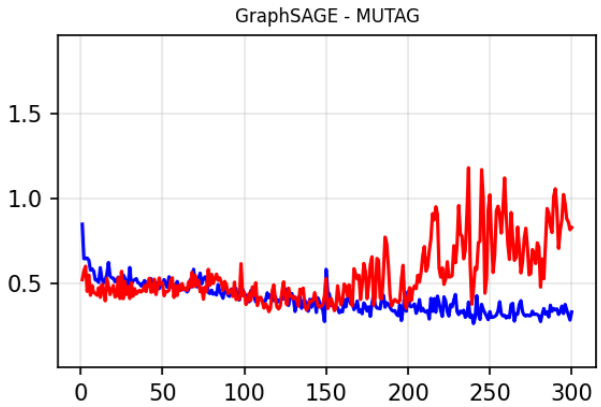


FIGURE 4.4 – Évolution des courbes de loss pour GraphSAGE sur le dataset MUTAG.

La courbe d'entraînement est très basse et stable. En revanche, la courbe de validation est fortement bruitée, avec de grandes fluctuations. Ce décalage suggère une instabilité importante, probablement liée à une forte variance dans les données de validation ou à

un sur-paramétrage du modèle. Plus précisément, GraphSAGE montre un comportement instable sur MUTAG : un bon apprentissage sur les données d'entraînement, mais un risque clair d'overfitting en validation.

Concernant les autres modèles, GAT présente également une courbe de validation très bruitée, traduisant une forte instabilité. À l'inverse, GCN est plus régulier, avec des courbes rapprochées. GIN, quant à lui, reste globalement stable, mais affiche une loss de validation légèrement plus élevée que celle de GCN.

4.4.3 Analyse des courbes de loss – PROTEINS

La figure 4.5 présente l'évolution des courbes de loss pour GIN sur le dataset PROTEINS .

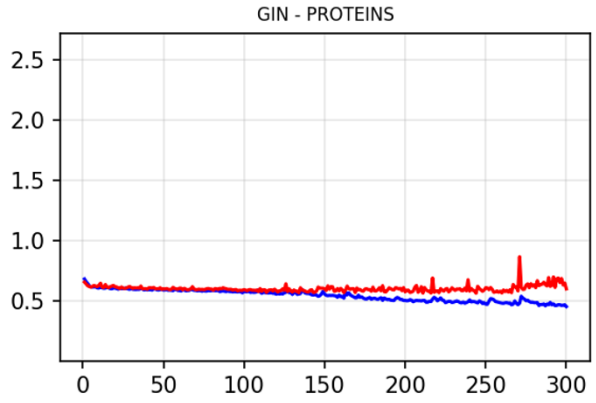


FIGURE 4.5 – Évolution des courbes de loss pour GIN sur le dataset PROTEINS.

Les courbes d'apprentissage et de validation sont quasi superposées, très proches et stables sur toute la durée de l'entraînement (300 époques). Cela reflète une excellente adaptation du modèle, avec une très bonne stabilité et une généralisation efficace, sans signe de sur-apprentissage.

Dans ce contexte, GIN se distingue par un apprentissage particulièrement performant, combinant stabilité remarquable et capacité de généralisation optimale. Concernant les autres modèles, GCN et GAT présentent des courbes similaires, légèrement en dessous de celles de GIN. GraphSAGE, quant à lui, est un peu moins stable, mais reste globalement compétitif.

4.4.4 Analyse des courbes de loss – Citeseer

La figure 4.6 présente l'évolution des courbes de loss pour GraphSAGE sur le dataset Citeseer.

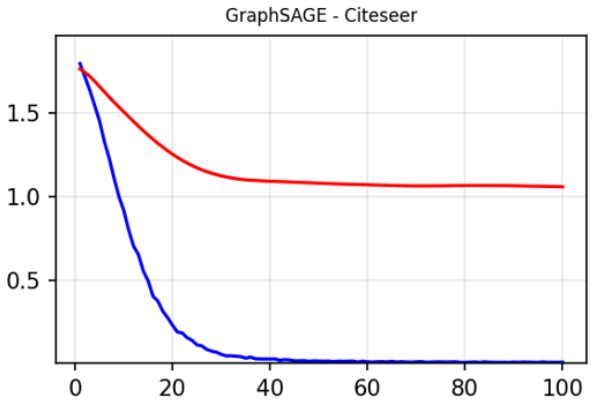


FIGURE 4.6 – Évolution des courbes de loss pour GraphSAGE sur le dataset Citeseer

La loss d'entraînement diminue rapidement, tandis que la courbe de validation suit de façon fluide, sans oscillation marquée, avec un écart modéré entre les deux. Ce comportement reflète un bon équilibre entre biais et variance, traduisant une convergence maîtrisée.

Dans ce contexte, GraphSAGE montre un apprentissage stable et une bonne capacité de généralisation sur Citeseer. En ce qui concerne les autres modèles, GIN présente une loss de validation croissante, caractéristique d'un sur-apprentissage. GCN converge plus lentement, tandis que GAT atteint un bon niveau de performance, mais reste légèrement en retrait par rapport à GraphSAGE.

4.4.5 Analyse des courbes de loss – PubMed

La figure 4.7 présente l'évolution des courbes de loss pour GIN sur le dataset PubMed
.

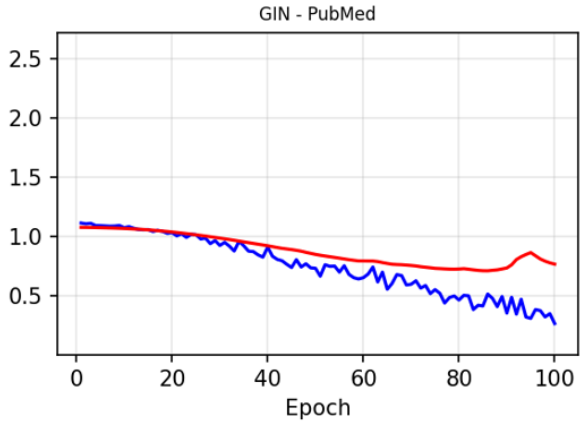


FIGURE 4.7 – Évolution des courbes de loss pour GIN sur le dataset PubMed

La loss d'entraînement diminue régulièrement, mais la courbe de validation reste globalement plus élevée, sans amélioration significative. Cela montre que GIN parvient à apprendre sur les données d'entraînement, mais peine à généraliser, la courbe de validation restant persistante. Ainsi, GIN présente un sous-apprentissage partiel sur PubMed, accompagné d'une légère tendance à l'overfitting.

Concernant les autres modèles, GCN obtient les meilleurs résultats, avec des courbes bien alignées. GAT converge plus lentement, tandis que GraphSAGE affiche un comportement intermédiaire, mais plus stable que GIN.

L'ensemble des courbes obtenues lors de l'entraînement des différents modèles GNN sur plusieurs jeux de données est présenté ci-dessous.

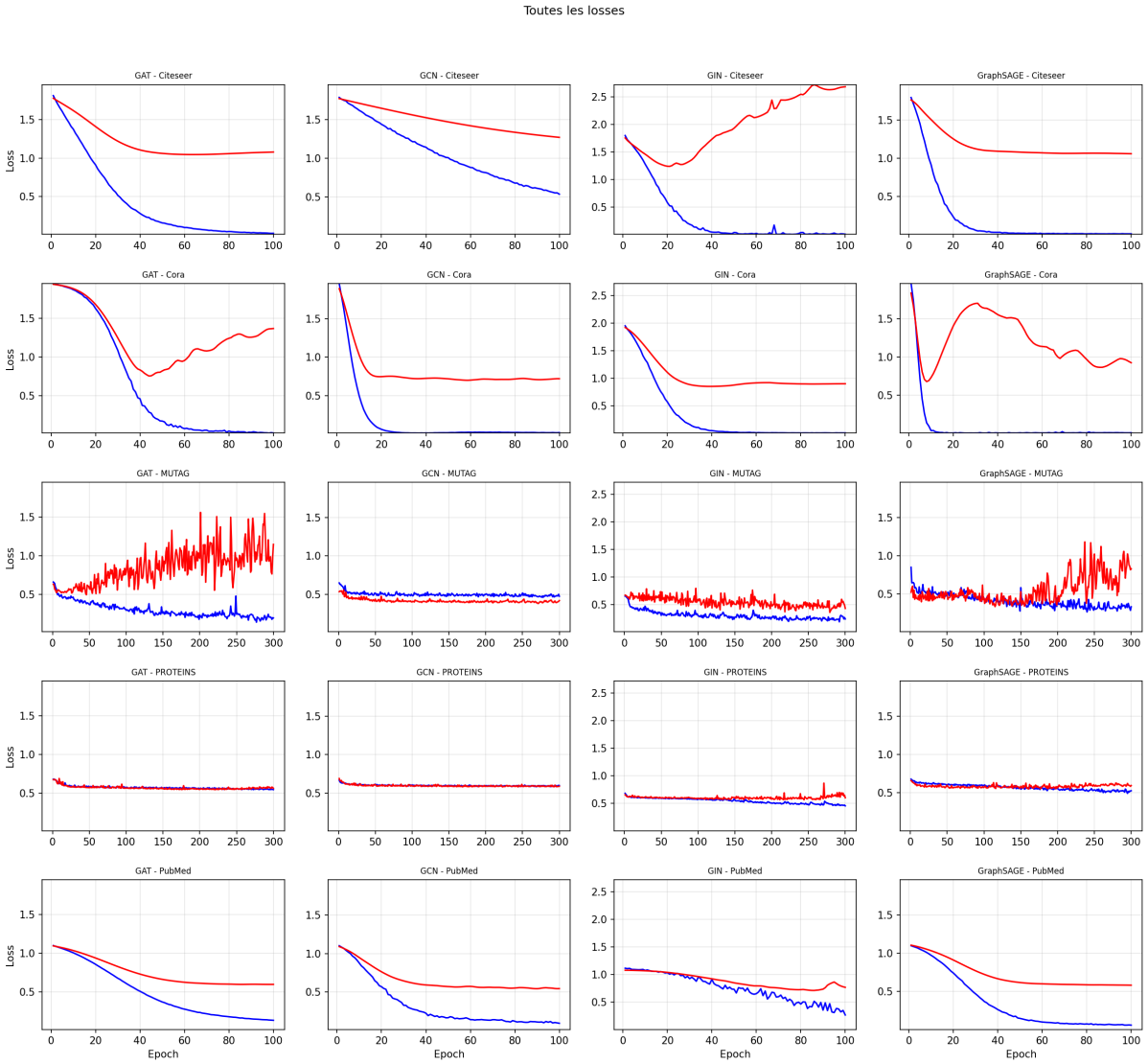


FIGURE 4.8 – Évolution des courbes de loss pour l'ensemble des modèles GNN testés sur les différents jeux de données.

4.5 Conclusion

Ce chapitre a présenté une évaluation expérimentale des performances de quatre modèles de GNN appliqués à des tâches de classification de noeuds et de graphes. L'analyse des résultats, complétée par des courbes de perte et des comparaisons avec l'état de l'art, a permis de mieux comprendre les avantages et les limites de chaque architecture. Des phénomènes comme le surapprentissage, le sous-apprentissage et l'instabilité ont été identifiés, soulignant l'importance du réglage des hyperparamètres.

L'ensemble des scripts Python, modèles entraînés, notebooks d'expérimentation, ainsi que les fichiers de configuration utilisés dans le cadre de ce projet sont disponibles publiquement sur le dépôt GitHub suivant : `https://github.com/nawelgnn2025/Gnn-pfe`

Conclusion générale

Les Graph Neural Networks (GNN) représentent une avancée majeure dans le domaine de l'apprentissage profond, en permettant la modélisation efficace de données structurées sous forme de graphes. Là où les approches traditionnelles échouent à saisir la richesse des relations entre entités, les GNN se distinguent par leur capacité à agréger l'information à travers les nœuds, ce qui les rend particulièrement pertinents pour des tâches telles que la classification de nœuds ou de graphes.

Ce projet a permis la mise en œuvre, l'optimisation et l'évaluation de plusieurs architectures de GNN (GCN, GAT, GraphSAGE, GIN) sur divers ensembles de données de référence (Cora, Citeseer, PubMed, MUTAG, PROTEINS). L'étude a intégré l'optimisation des hyperparamètres à l'aide d'Optuna, la visualisation des résultats via un tableau de bord interactif, ainsi que l'analyse des courbes de perte pour mieux comprendre le comportement des modèles.

L'analyse approfondie des résultats a permis de dégager plusieurs enseignements clés. Elle a mis en évidence l'importance du choix de l'architecture en fonction de la nature des données et des objectifs visés. Par ailleurs, les performances des modèles se sont révélées étroitement dépendantes des hyperparamètres (nombre de couches, taille des embeddings, etc.), soulignant la nécessité d'une phase d'optimisation rigoureuse. L'examen des courbes de perte s'est également avéré instructif pour détecter des phénomènes tels que la convergence, le surapprentissage, le sous-apprentissage ou encore l'instabilité, offrant ainsi des leviers d'intervention pour améliorer l'apprentissage.

Ce travail constitue une première exploration des performances de quatre architectures classiques de GNN sur des jeux de données standards, avec un focus particulier sur la classification. Il ouvre néanmoins de nombreuses perspectives d'approfondissement, notamment l'étude de modèles plus récents (Graph Transformers, GIN+), l'extension à d'autres tâches (génération de graphes, détection d'anomalies), ou encore l'adaptation à des contextes plus complexes, industriels ou dynamiques.

Bibliographie

[1] Abien F. Agarap. Deep learning using rectified linear units (relu), 2018. Consulté le 18 mars 2025.

[2] Analytics Vidhya. Exploration of iris dataset using scikit-learn – part 1. https://medium.com/analytics-vidhya/exploration-of-iris-dataset-using-scikit-learn-part-1-8ac5604937f8, 2025. Consulté le 15 mars 2025.

[3] Réseaux de neurones artificiels (ann). https://itmag.tdsynnex.fr, 2025. Consulté le 15 février 2025.

[4] Michael Bernstein. A gentle introduction to graph convolutional networks. https://mbernste.github.io/posts/gcn/, 2021. Consulté le 22 mars 2025.

[5] Imen Bestani and Noureddine Benadjrouda. Développement des systèmes de recommandation de santé à l'aide des réseaux convolutifs. Mémoire de master, Université Ibn Khaldoun - Tiaret, 2023.

[6] Joyce Birkins. Comprehensive guide to gnn, gat, and gcn – a beginner's introduction to graph neural networks (after transformers), 2024. Consulté le 2 avril 2025.

[7] Karsten M Borgwardt and Hans-Peter Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21(suppl_1) :i47–i56, 2005.

[8] DataScientest. Graph neural networks : tout savoir. https://datascientest.com/graph-neural-networks-tout-savoir, 2023. Consulté le 4 avril 2025.

[9] Asim Debnath, Ricardo L Lopez de Compadre, Sudhir Debnath, Alan J Shusterman, and Corwin Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of medicinal chemistry*, 34(2) :786–797, 1991.

[10] Introduction au deep learning et aux réseaux de neurones, 2025. Consulté le 17 février 2025.

[11] Pantelis Elinas. Knowing your neighbours : Machine learning on graphs, 2019. Consulté le 20 mars 2025.

[12] Méthodes d'apprentissage supervisé et non supervisé. https://www.fastercapital.com, 2025. Consulté le 16 février 2025.

[13] Ronald A Fisher. The use of multiple measurements in taxonomic problems. *Annals of eugenics*, 7(2) :179–188, 1936.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. Disponible en ligne : `http://www.deeplearningbook.org` (Consulté le 4 mai 2025).

[15] Geoffrey Hinton. The forward–forward algorithm : Some preliminary investigations, 2022. Consulté le 13 mars 2025.

[16] Qu'est-ce que l'apprentissage par renforcement. `https://www.ibm.com/fr-fr/think/topics/reinforcement-learning`, 2025. Consulté le 16 février 2025.

[17] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11) :2278–2324, 1998.

[18] Wai Weng Lo, Siamak Layeghy, Mohanad Sarhan, Marcus Gallagher, and Marius Portmann. E-graphsage : A graph neural network based intrusion detection system for iot. In *2022 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2022.

[19] Ghalia Merzougui. Support de cours : Chapitre 3 – réseaux de neurones convolutifs (cnn). Technical report, Université Mustapha Benboulaid de Batna, Département de Mathématiques et Informatique, 2021. Disponible en ligne.

[20] Galileo Namata, Ben London, Lise Getoor, Bert Huang, and UMD EDU. Query-driven active surveying for collective classification. In *10th International Workshop on Mining and Learning with Graphs*, 2012.

[21] Samba Ndojh Ndiaye. Applications des graphes. `https://perso.liris.cnrs.fr/samba-ndojh.ndiaye/fichiers/App_Graphes.pdf`, 2018. Consulté le 22 mars 2025.

[22] nugg.ad. Réseaux de neurones sur graphes, 2025.

[23] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. Activation functions : Comparison of trends in practice and research for deep learning, 2018. Consulté le 10 mars 2025.

[24] N. Pancino, P. Bongini, F. Scarselli, and M. Bianchini. Gnnkeras : A keras-based library for graph neural networks and homogeneous and heterogeneous graph processing. *SoftwareX*, 18 :101061, 2022.

[25] Perceptrons multicouches dans l'apprentissage automatique. `https://www.datacamp.com`, 2025. Consulté le 18 février 2025.

[26] Molkhir Rabia and Siham Saidani. Réseaux de neurones pour les jeux de plateau. Mémoire de master, Université Akli Mohand Oulhadj - Bouira, 2020. Consulté le 15 février 2025.

[27] Guilhem Renton. *Analyzing and improving graph neural networks*. Thèse de doctorat, Normandie Université, 2021. [En ligne] : `https://theses.hal.science/tel-03346018/`.

[28] Qu'est-ce que les recurrent neural networks ? `https://www.intelligence-artificielle-school.com`, 2025. Consulté le 19 février 2025.

[29] Frank Rosenblatt. The perceptron : A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6) :386–408, 1958. Consulté le 16 mars 2025.

[30] Sebastian Ruder. An overview of gradient descent optimization algorithms, 2016. `https://arxiv.org/pdf/1609.04747.pdf`, Consulté le 16 mars 2025.

[31] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088) :533–536, 1986.

[32] Prithviraj Sen, Galileo Namata, Mustafa Bilgic, Lise Getoor, Brian Galligher, and Tina Eliassi-Rad. Collective classification in network data. *AI magazine*, 29(3) :93–93, 2008.

[33] TechiesCamp. Semi-supervised machine learning, 2025. Consulté le 15 mars 2025.

[34] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv :1710.10903*, 2017.

[35] Cédric Vert. Graphml - partie 2 : Attributs typés, graphes mixtes, structures hiérarchiques. `https://cedric.cnam.fr/vertigo/cours/RCP217/docs/RCP217-GraphML2.pdf`, 2021. Consulté le 2 avril 2025.

[36] Cédric Vert. Introduction à la représentation des graphes en xml avec graphml. `https://cedric.cnam.fr/vertigo/cours/RCP217/docs/RCP217-GraphML1.pdf`, 2021. Consulté le 23 mars 2025.

[37] Cheng Wu. *Graph Representation Learning : from Kernel to Neural Networks*. Thèse de doctorat, Institut Polytechnique de Paris, 2021.

[38] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks ? In *International Conference on Learning Representations (ICLR)*, 2019.

[39] Yash. Cnn for mnist handwritten dataset. `https://medium.com/@yash.4198/cnn-for-mnist-handwritten-dataset-cc94d61f6c94`, 2025. Consulté le 15 mars 2025.