**Abstract**

Dans ce projet, nous avons entrepris l'optimisation d'une application existante basée sur un modèle de Graph Neural Network (GNN) appliqué à la classification de nœuds dans un graphe de citations, en utilisant le dataset Cora. Cette application a pour objectif de prédire le sujet d’un article scientifique à partir de ses mots-clés et de son réseau de citations. Bien que fonctionnelle, l’application initiale présentait plusieurs limitations, notamment une précision relativement faible et des performances computationnelles perfectibles, limitant ainsi son efficacité dans un contexte pratique. Pour répondre à ces problématiques, nous avons adopté une démarche d’optimisation rigoureuse, basée sur l’ajustement des hyperparamètres du modèle. Les principales variables ajustées incluent la taille des couches cachées, qui influence directement la capacité d’apprentissage du modèle, le taux de dropout, utilisé pour éviter le surapprentissage, et les types d’agrégation et de combinaison, qui définissent comment les informations des nœuds voisins sont intégrées dans le graphe. Par ailleurs, le taux d’apprentissage, un paramètre crucial influençant la vitesse et la stabilité de la convergence du modèle, a été soigneusement ajusté. En complément de ces ajustements, nous avons introduit de nouvelles techniques d’encodage des données, visant à mieux représenter les relations structurelles et les caractéristiques des nœuds dans le graphe. Ces optimisations combinées ont permis de transformer significativement les performances du modèle. Les résultats expérimentaux montrent une amélioration notable de la précision de classification, qui est passée de **68,06 % à 74,85 %** , validant ainsi l’efficacité des ajustements proposés. Cette progression met en lumière le rôle clé de l’ajustement des hyperparamètres et des techniques d’encodage dans l’amélioration des modèles GNN pour des tâches complexes comme la classification de nœuds. En définitive, ce travail contribue à démontrer comment des approches méthodiques d’optimisation peuvent non seulement améliorer les performances d’un modèle, mais aussi offrir des perspectives plus larges pour l’application des GNN dans des contextes similaires, tels que l’analyse des réseaux sociaux, la biologie computationnelle, ou encore les systèmes de recommandation.

1. **Introduction:**

Les datasets utilisés dans de nombreuses applications d'apprentissage automatique (ML) présentent souvent des **relations structurelles** entre leurs entités, ce qui permet de les représenter sous forme de **graphes**. Ces représentations sont particulièrement pertinentes pour des applications telles que l'analyse des réseaux sociaux, la détection de fraudes, la prédiction de trafic, et la classification de documents scientifiques.

La classification de nœuds dans les graphes repose sur des techniques de **Graph Representation Learning**, qui visent à extraire et à exploiter les relations entre les nœuds pour résoudre une variété de tâches ML. Les modèles de Graph Neural Networks (GNN) constituent une approche puissante pour ces tâches, car ils combinent les propriétés structurelles des graphes avec des techniques d'apprentissage profond.

Dans ce contexte, l'application initiale que nous avons utilisée implémente un modèle GNN pour une tâche de classification de nœuds sur le dataset **Cora**, qui contient des articles scientifiques et leurs réseaux de citations. L'objectif est de prédire le sujet principal d'un article en exploitant à la fois ses caractéristiques textuelles (mots-clés) et ses relations structurelles (citations). Le modèle de base utilise une **Graph Convolution Layer** implémentée à partir de zéro pour favoriser la compréhension du fonctionnement interne des GNN.

Cependant, ce modèle initial présente des limites, notamment en termes de précision et d'efficacité dans la capture des relations complexes au sein du graphe. Dans ce projet, nous visons à optimiser ce modèle en proposant des améliorations au niveau de son architecture, de ses paramètres, et de ses techniques d'entraînement, afin d'améliorer ses performances sur la tâche de classification de nœuds.

1. **Présentation du Modèle : Graph Neural Network (GNN)**

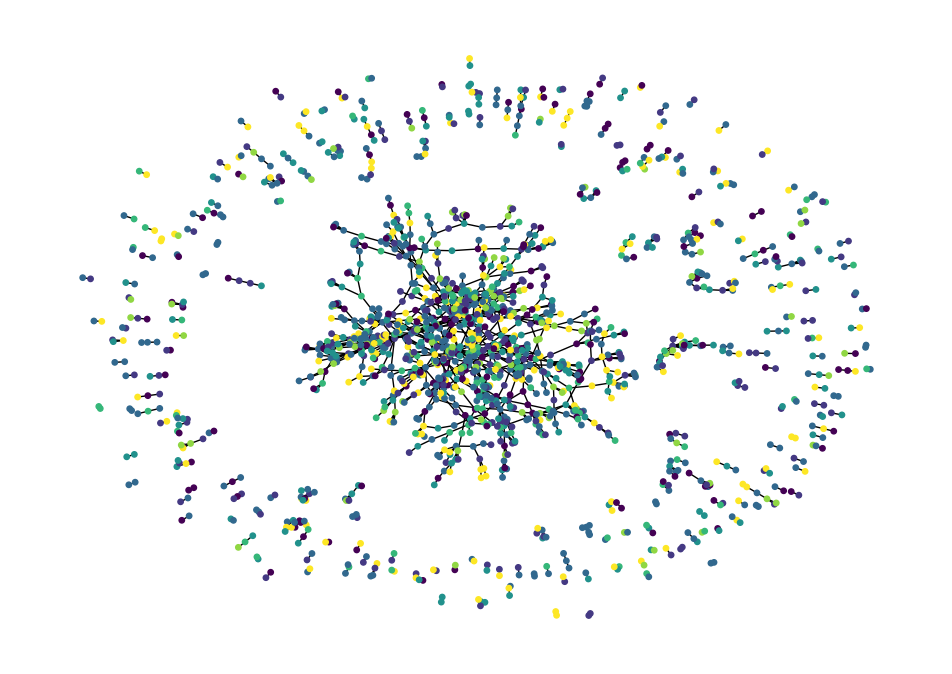
Le modèle utilisé dans ce projet est un Graph Neural Network (GNN) conçu pour la classification de nœuds dans un graphe de citations. Ce type de modèle est particulièrement adapté aux données structurées en graphes, où les relations entre les entités (nœuds) sont importantes et influencent les prédictions.

1. **Structure du GNN :**

* **Entrée du Modèle :**
* **Graph Information :** Le modèle reçoit un graphe d'entrées représentant un réseau de citations, où chaque nœud représente un article scientifique et les arêtes représentent les citations entre ces articles.
* **Caractéristiques des Nœuds :** Chaque nœud (article) est associé à un vecteur de caractéristiques (par exemple, les mots-clés de l'article ou des informations spécifiques comme le titre ou l'auteur).
* **Adjacency Matrix :** Une matrice d’adjacence est utilisée pour décrire la connectivité entre les nœuds du graphe, indiquant quelles paires de nœuds sont reliées par une citation.
* **Couches du Modèle (Graph Convolution Layers) :** Le modèle se compose de plusieurs couches de Graph Convolution (convolution graphes), qui permettent d'apprendre des représentations des nœuds en tenant compte de leur voisinage dans le graphe. Chaque couche de convolution met à jour la représentation d'un nœud en fonction de ses voisins dans le graphe, en appliquant une fonction d’agrégation.
* **Première couche de Graph Convolution :** Chaque nœud est mis à jour en agrégeant les informations de ses voisins directs dans le graphe. Cette agrégation se fait généralement en utilisant des techniques comme la somme, la moyenne ou le maximum des caractéristiques des voisins.
* **Couches cachées :** Le modèle peut comporter plusieurs couches de graph convolution, où chaque couche affine les représentations des nœuds en intégrant progressivement des informations à plus longue portée (c'est-à-dire, des nœuds de plus en plus éloignés dans le graphe). Le nombre de couches détermine la profondeur du modèle et la capacité à capturer des dépendances complexes.
* **Couches du Modèle (Graph Convolution Layers)** : Le modèle se compose de plusieurs couches de **Graph Convolution** (convolution graphes), qui permettent d'apprendre des représentations des nœuds en tenant compte de leur voisinage dans le graphe. Chaque couche de convolution met à jour la représentation d'un nœud en fonction de ses voisins dans le graphe, en appliquant une fonction d’agrégation.
* **Première couche de Graph Convolution** : Chaque nœud est mis à jour en agrégeant les informations de ses voisins directs dans le graphe. Cette agrégation se fait généralement en utilisant des techniques comme la somme, la moyenne ou le maximum des caractéristiques des voisins.
* **Couches cachées** : Le modèle peut comporter plusieurs couches de graph convolution, où chaque couche affine les représentations des nœuds en intégrant progressivement des informations à plus longue portée (c'est-à-dire, des nœuds de plus en plus éloignés dans le graphe). Le nombre de couches détermine la profondeur du modèle et la capacité à capturer des dépendances complexes.
* **Agrégation et Combinaison :**
* **Fonctions d'agrégation** : Le modèle utilise différentes méthodes pour agréger les informations provenant des voisins, telles que :
  + **Sum** : Somme des caractéristiques des voisins.
  + **Mean** : Moyenne des caractéristiques des voisins.
  + **Max** : Prend la valeur maximale des caractéristiques des voisins.
* **Combinaison des Caractéristiques** : Après agrégation, les caractéristiques combinées des voisins sont mélangées avec les caractéristiques du nœud cible. Cela se fait par des opérations telles que **concaténation**, **addition** ou même des techniques avancées comme les **GRU** (Gated Recurrent Units), qui peuvent aider à mieux capturer les dépendances temporelles ou structurales dans les graphes.
* **Regularization** : Pour éviter le sur-apprentissage (overfitting), le modèle peut intégrer des techniques de régularisation :
* **Dropout** : Une technique qui consiste à "éteindre" aléatoirement une fraction des neurones pendant l'entraînement pour encourager une meilleure généralisation.
* **L2 Regularization** : Pénalisation des poids du modèle pour limiter leur amplitude et ainsi éviter un sur-apprentissage sur des données spécifiques.
* **Sortie du Modèle :**
* **Classification des Nœuds** : À la sortie, le modèle effectue une tâche de classification pour prédire la catégorie de chaque nœud (dans ce cas, le sujet d’un article scientifique), en utilisant une fonction de perte appropriée (par exemple, **sparse categorical crossentropy**) et une couche de sortie softmax pour produire des probabilités de classification pour chaque classe.
* **Prédiction des Labels** : Le modèle attribue à chaque nœud (article) une étiquette correspondant à un sujet ou une catégorie, en fonction de son contexte (les citations et le contenu des autres articles liés).

1. **Visualisation du Graphe de Citations :**

Dans ce projet, chaque nœud du graphe représente un article scientifique, et sa couleur indique son sujet. Nous montrons ici un échantillon du dataset pour illustrer les relations de citation entre les articles et leur classification par sujet.



1. **Related Work:**

Les **Graph Neural Networks (GNNs)** ont émergé comme une approche puissante pour l'analyse de données structurées sous forme de graphes. Plusieurs travaux ont exploré leur utilisation pour des tâches de **classification de nœuds**, en particulier sur des jeux de données comme **Cora**, **Citeseer**, et **PubMed**.

L'un des premiers travaux marquants dans ce domaine est celui de **Kipf et Welling (2017)**, qui ont proposé le modèle **Graph Convolutional Network (GCN)**. Ce modèle introduit une approche de convolution adaptée aux graphes pour l'apprentissage des représentations des nœuds. Les GCNs ont rapidement été adoptés dans la littérature en raison de leur simplicité et de leur efficacité pour des tâches de classification de nœuds et de liens. Cependant, l'un des défis des GCNs réside dans leur tendance au **sur-lissage** des représentations lorsque le nombre de couches de convolution augmente, ce qui peut nuire aux performances du modèle, comme l'ont souligné **Li et al. (2018)** dans leurs travaux sur le sur-apprentissage des graphes profonds.

Pour répondre à ces limitations, plusieurs travaux ont proposé des améliorations et des variantes. **Velickovic et al. (2018)** ont introduit les **Graph Attention Networks (GAT)**, qui utilisent des mécanismes d'attention pour adapter l'importance des voisins dans la propagation des informations, offrant ainsi une plus grande flexibilité et une meilleure gestion des graphes de grande taille. De plus, **Hamilton et al. (2017)** ont proposé **GraphSAGE**, une méthode qui permet de générer des représentations de nœuds en utilisant des techniques d'échantillonnage de voisins, ce qui est particulièrement utile pour les graphes très grands où la propagation à travers tout le graphe serait computationnellement coûteuse.

Dans le domaine de l'optimisation des hyperparamètres pour les modèles GNN, **You et al. (2020)** ont montré que l'ajustement des hyperparamètres, y compris la taille des couches cachées, le taux de dropout et le type d'agrégation, pouvait avoir un impact significatif sur les performances des modèles. Ils ont démontré qu'une approche systématique d'optimisation des hyperparamètres conduisait à une amélioration notable de la précision dans des tâches de classification de nœuds, notamment sur des graphes de citations.

En conclusion, bien que plusieurs approches et modèles aient été proposés pour la classification de nœuds dans des graphes, notre travail se distingue par l'intégration d'une optimisation des hyperparamètres dans un modèle GNN appliqué à un graphe de citations. Nous mettons en œuvre des stratégies avancées pour ajuster les hyperparamètres et améliorer les performances en termes de précision de classification, en prenant en compte les résultats de ces travaux connexes pour affiner notre propre approche.

1. **Approvement:**
   1. **Version initiale**

import os  
import pandas as pd  
import numpy as np  
import networkx as nx  
import matplotlib.pyplot as plt  
import tensorflow as tf  
from tensorflow import keras  
from tensorflow.keras import layers

Cette première partie correspond à l'importation des bibliothèques nécessaires pour travailler sur les Graph Neural Networks (GNNs). Voici une brève explication de chaque bibliothèque et pourquoi elle pourrait être utile :

1. **os** :
   * Permet d'interagir avec le système d'exploitation, comme la gestion des fichiers et des répertoires.
   * Utilisé pour charger ou enregistrer des données et des modèles.
2. **pandas (pd)** :
   * Fournit des structures de données comme les DataFrames pour manipuler les données tabulaires.
   * Utile pour charger, nettoyer, et préparer les données avant de construire un graphe.
3. **numpy (np)** :
   * Utilisé pour les calculs numériques, comme la manipulation de matrices et de vecteurs.
   * Essentiel pour les opérations mathématiques dans le cadre de l'apprentissage machine.
4. **networkx (nx)** :
   * Bibliothèque spécialisée dans la création, la manipulation et l'analyse des graphes.
   * Idéal pour créer et visualiser des graphes avant leur passage dans un GNN.
5. **matplotlib.pyplot (plt)** :
   * Bibliothèque pour tracer et visualiser les données.
   * Peut être utilisée pour visualiser les graphes ou les résultats.
6. **tensorflow et keras** :
   * Framework d'apprentissage automatique pour construire et entraîner les modèles de réseaux de neurones.
   * Ici, il sera utilisé pour définir et entraîner le GNN.
7. **tensorflow.keras.layers** :
   * Fournit les différentes couches utilisées pour construire les réseaux de neurones.
   * Essentiel pour personnaliser et ajuster l'architecture du GNN.

zip\_file = keras.utils.get\_file(  
 fname="cora.tgz",  
 origin="https://linqs-data.soe.ucsc.edu/public/lbc/cora.tgz",  
 extract=True,  
)  
data\_dir = os.path.join(os.path.dirname(zip\_file), "cora")

Downloading data from https://linqs-data.soe.ucsc.edu/public/lbc/cora.tgz  
168052/168052 ━━━━━━━━━━━━━━━━━━━━ 1s 3us/step

citations = pd.read\_csv(  
 os.path.join(data\_dir, "cora.cites"),  
 sep="\t",  
 header=None,  
 names=["target", "source"],  
)  
print("Citations shape:", citations.shape)

Citations shape: (5429, 2)

citations.sample(frac=1).head()

Dans cette étape, nous téléchargeons et préparons les données du dataset **Cora**, qui est souvent utilisé pour des tâches liées aux Graph Neural Networks. Voici une explication pas à pas.

### Code expliqué :

1. **Téléchargement et extraction du jeu de données** :

* zip\_file = keras.utils.get\_file(  
   fname="cora.tgz",  
   origin="https://linqs-data.soe.ucsc.edu/public/lbc/cora.tgz",  
   extract=True,  
  )
  + **keras.utils.get\_file** : Télécharge le fichier spécifié par l'URL (origin) et l'extrait automatiquement.
  + **fname** : Nom du fichier à télécharger.
  + **origin** : URL de la source du fichier.
  + **extract=True** : Permet d'extraire automatiquement les fichiers du fichier compressé.

1. **Définir le chemin du répertoire des données** :

* data\_dir = os.path.join(os.path.dirname(zip\_file), "cora")
  + Combine le chemin du fichier extrait et le sous-dossier "cora" contenant les données.

1. **Chargement des citations (arêtes du graphe)** :

* citations = pd.read\_csv(  
   os.path.join(data\_dir, "cora.cites"),  
   sep="\t",  
   header=None,  
   names=["target", "source"],  
  )
  + **os.path.join(data\_dir, "cora.cites")** : Chemin du fichier contenant les relations entre les articles (cibles et sources).
  + **sep="\t"** : Les colonnes sont séparées par une tabulation.
  + **header=None** : Pas de ligne d'en-tête dans le fichier.
  + **names=["target", "source"]** : Donne des noms explicites aux colonnes.

1. **Afficher des informations sur les données** :

* print("Citations shape:", citations.shape)
  + Affiche la forme du DataFrame (nombre de lignes et colonnes), ici représentant les arêtes du graphe.

1. **Aperçu aléatoire des données** :

* citations.sample(frac=1).head()
  + **sample(frac=1)** : Mélange les lignes de manière aléatoire.
  + **.head()** : Affiche les 5 premières lignes des données mélangées.

### Résultat attendu :

1. **Forme des données** : Le message imprimé affichera quelque chose comme :

* Citations shape: (5429, 2)
* Ce qui indique qu'il y a 5 429 arêtes (relations entre articles).

1. **Aperçu des données** : Les colonnes "target" et "source" contiennent les identifiants des articles liés par des citations.



|  |  |  |
| --- | --- | --- |
|  | **target** | **source** |
| **2160** | 18774 | 73146 |
| **1489** | 7047 | 1135082 |
| **1928** | 13213 | 1152143 |
| **5209** | 643221 | 642894 |
| **946** | 3828 | 1956 |

column\_names = ["paper\_id"] + [f"term\_{idx}" for idx in range(1433)] + ["subject"]  
papers = pd.read\_csv(  
 os.path.join(data\_dir, "cora.content"), sep="\t", header=None, names=column\_names,  
)  
print("Papers shape:", papers.shape)

Papers shape: (2708, 1435)

print(papers.sample(5).T)

1938 2109 2656 2372   
paper\_id 1105877 820661 103528 82666   
term\_0 0 0 0 0   
term\_1 0 0 0 0   
term\_2 0 0 0 0   
term\_3 0 0 0 0   
... ... ... ... ...   
term\_1429 0 0 0 0   
term\_1430 0 0 0 0   
term\_1431 0 0 0 0   
term\_1432 0 0 0 0   
subject Probabilistic\_Methods Neural\_Networks Case\_Based Case\_Based   
  
 78   
paper\_id 40124   
term\_0 0   
term\_1 0   
term\_2 0   
term\_3 0   
... ...   
term\_1429 0   
term\_1430 0   
term\_1431 0   
term\_1432 0   
subject Theory   
  
[1435 rows x 5 columns]

print(papers.subject.value\_counts())

subject  
Neural\_Networks 818  
Probabilistic\_Methods 426  
Genetic\_Algorithms 418  
Theory 351  
Case\_Based 298  
Reinforcement\_Learning 217  
Rule\_Learning 180  
Name: count, dtype: int64

Dans cette partie du code, nous chargeons les données concernant les **articles scientifiques** (nœuds du graphe) et leurs **caractéristiques** (features).

### Code expliqué :

1. **Définition des noms de colonnes** :

* column\_names = ["paper\_id"] + [f"term\_{idx}" for idx in range(1433)] + ["subject"]
  + **paper\_id** : Identifiant unique pour chaque article.
  + **term\_0 à term\_1432** : Ce sont des caractéristiques représentant la présence ou l'absence de certains mots dans l'article (vecteur binaire avec 1433 termes possibles).
  + **subject** : La catégorie ou le domaine scientifique de l'article (par exemple, "Neural Networks", "Genetics").

1. **Chargement des données des articles** :

* papers = pd.read\_csv(  
   os.path.join(data\_dir, "cora.content"), sep="\t", header=None, names=column\_names,  
  )
  + **cora.content** : Fichier contenant des informations sur les articles.
  + **sep="\t"** : Les colonnes sont séparées par des tabulations.
  + **header=None** : Pas de ligne d'en-tête dans le fichier.
  + **names=column\_names** : Définit les noms des colonnes selon la liste définie précédemment.

1. **Afficher la taille des données** :

* print("Papers shape:", papers.shape)
  + Affiche le nombre d’articles (lignes) et d’attributs (colonnes).
  + **Exemple attendu** :
  + Papers shape: (2708, 1435)

1. **Aperçu des données (5 articles)** :

* print(papers.sample(5).T)
  + **sample(5)** : Sélectionne 5 articles aléatoires.
  + **.T** : Transpose les données pour une visualisation plus lisible (colonnes affichées comme lignes).

1. **Comptage des sujets** :

* print(papers.subject.value\_counts())
  + **papers.subject.value\_counts()** : Compte le nombre d’articles par catégorie scientifique.
  + Cela permet de voir la distribution des sujets (par exemple, combien d’articles appartiennent à "Neural Networks").

class\_values = sorted(papers["subject"].unique())  
class\_idx = {name: id for id, name in enumerate(class\_values)}  
paper\_idx = {name: idx for idx, name in enumerate(sorted(papers["paper\_id"].unique()))}  
  
papers["paper\_id"] = papers["paper\_id"].apply(lambda name: paper\_idx[name])  
citations["source"] = citations["source"].apply(lambda name: paper\_idx[name])  
citations["target"] = citations["target"].apply(lambda name: paper\_idx[name])  
papers["subject"] = papers["subject"].apply(lambda value: class\_idx[value])

plt.figure(figsize=(10, 10))   
colors = papers["subject"].tolist()  
cora\_graph = nx.from\_pandas\_edgelist(citations.sample(n=1500))  
subjects = list(papers[papers["paper\_id"].isin(list(cora\_graph.nodes))]["subject"])  
nx.draw\_spring(cora\_graph, node\_size=15, node\_color=subjects)

Dans cette partie, nous préparons les données pour créer et visualiser un **graphe** des articles scientifiques.

### Étape 1 : Préparer les index

1. **Classes (subject)** :

* class\_values = sorted(papers["subject"].unique())  
  class\_idx = {name: id for id, name in enumerate(class\_values)}
  + **class\_values** : Liste des catégories uniques d'articles, triées.
  + **class\_idx** : Dictionnaire qui associe chaque catégorie à un numéro unique.

1. **Articles (paper\_id)** :

* paper\_idx = {name: idx for idx, name in enumerate(sorted(papers["paper\_id"].unique()))}
  + **paper\_idx** : Associe chaque article à un numéro unique pour simplifier leur utilisation.

### Étape 2 : Transformer les données

1. **Appliquer les indices** :

* papers["paper\_id"] = papers["paper\_id"].apply(lambda name: paper\_idx[name])  
  citations["source"] = citations["source"].apply(lambda name: paper\_idx[name])  
  citations["target"] = citations["target"].apply(lambda name: paper\_idx[name])  
  papers["subject"] = papers["subject"].apply(lambda value: class\_idx[value])
  + Convertit les identifiants et les catégories des articles en **valeurs numériques**.
  + Simplifie la manipulation des données pour le modèle et le graphe.

### Étape 3 : Visualiser le graphe

1. **Créer un graphe avec NetworkX** :

* cora\_graph = nx.from\_pandas\_edgelist(citations.sample(n=1500))
  + **citations.sample(n=1500)** : Sélectionne 1500 citations aléatoires pour un graphe plus petit et plus facile à visualiser.
  + **nx.from\_pandas\_edgelist** : Crée un graphe où les nœuds sont des articles et les arêtes sont des citations.

1. **Couleurs des nœuds (par catégorie)** :

subjects = list(papers[papers["paper\_id"].isin(list(cora\_graph.nodes))]["subject"])

* + Associe une couleur à chaque nœud selon sa catégorie scientifique.

1. **Dessiner le graphe** :

* nx.draw\_spring(cora\_graph, node\_size=15, node\_color=subjects)
  + **nx.draw\_spring** : Utilise un algorithme pour positionner les nœuds de manière lisible.
  + **node\_size=15** : Taille des nœuds.
  + **node\_color=subjects** : Les nœuds sont colorés selon leur catégorie.

### Résultat attendu :

Vous obtenez une **visualisation du graphe** où :

* Chaque **nœud** représente un article.
* Chaque **arête** représente une citation.
* Les **couleurs** montrent les catégories (ou "sujets") des articles.

train\_data, test\_data = [], []  
  
for \_, group\_data in papers.groupby("subject"):  
 # Select around 50% of the dataset for training.  
 random\_selection = np.random.rand(len(group\_data.index)) <= 0.5  
 train\_data.append(group\_data[random\_selection])  
 test\_data.append(group\_data[~random\_selection])  
  
train\_data = pd.concat(train\_data).sample(frac=1)  
test\_data = pd.concat(test\_data).sample(frac=1)  
  
print("Train data shape:", train\_data.shape)  
print("Test data shape:", test\_data.shape)

Train data shape: (1351, 1435)  
Test data shape: (1357, 1435)

Dans cette partie, nous divisons les données des articles en **données d'entraînement** et **données de test**.

### Explication des étapes :

1. **Grouper par sujet (subject)** :

* for \_, group\_data in papers.groupby("subject"):
  + Les articles sont regroupés par leur **sujet** (par exemple, "Neural Networks", "Genetics", etc.).

1. **Sélection aléatoire des données** :

* random\_selection = np.random.rand(len(group\_data.index)) <= 0.5
  + Pour chaque groupe d'articles (par sujet), une **sélection aléatoire** de 50 % des articles est faite pour les utiliser comme **données d'entraînement**. L’autre moitié sera utilisée pour le **test**.

1. **Séparer les données en ensembles d’entraînement et de test** :

* train\_data.append(group\_data[random\_selection])  
  test\_data.append(group\_data[~random\_selection])
  + Les articles sont **divisés** en **données d'entraînement** et **données de test** en fonction de la sélection aléatoire.

1. **Mélanger les données** :

* train\_data = pd.concat(train\_data).sample(frac=1)  
  test\_data = pd.concat(test\_data).sample(frac=1)
  + Après avoir séparé les données, elles sont **mélangées** (avec .sample(frac=1)) pour s’assurer que les données sont bien mélangées avant d’être utilisées dans l’entraînement et les tests.

### Résultat :

* Vous obtenez deux ensembles : **train\_data** pour l'entraînement et **test\_data** pour évaluer les performances du modèle.

Les **formes** de ces ensembles sont affichées pour vérifier leur taille.

hidden\_units = [32, 32]  
learning\_rate = 0.01  
dropout\_rate = 0.5  
num\_epochs = 300  
batch\_size = 256

Ces variables définissent les **hyperparamètres** pour l'entraînement de notre modèle.

### Explication des hyperparamètres :

1. **hidden\_units = [32, 32]** :
   * **Nombre de neurones dans chaque couche cachée**.
   * Ici, le modèle a **deux couches cachées**, chacune avec **32 neurones**.
2. **learning\_rate = 0.01** :
   * **Taux d'apprentissage**.
   * C'est la vitesse à laquelle le modèle ajuste ses poids pendant l'entraînement. Ici, le modèle va ajuster ses poids à chaque étape de 0.01.
3. **dropout\_rate = 0.5** :
   * **Taux de dropout**.
   * Cela signifie que **50% des neurones** seront ignorés (mis à zéro) de manière aléatoire pendant chaque mise à jour des poids pour éviter le **sur-apprentissage (overfitting)** et rendre le modèle plus général.
4. **num\_epochs = 300** :
   * **Nombre d'époques**.
   * Le modèle sera entraîné pendant **300 itérations** sur l'ensemble des données d'entraînement.
5. **batch\_size = 256** :
   * **Taille de lot (batch)**.
   * À chaque étape d'entraînement, le modèle va traiter **256 échantillons** de données avant d'ajuster les poids.

Ces paramètres influencent directement l'efficacité et la performance du modèle d'apprentissage profond.

def run\_experiment(model, x\_train, y\_train):  
 # Compile the model.  
 model.compile(  
 optimizer=keras.optimizers.Adam(learning\_rate),  
 loss=keras.losses.SparseCategoricalCrossentropy(from\_logits=True),  
 metrics=[keras.metrics.SparseCategoricalAccuracy(name="acc")],  
 )  
 # Create an early stopping callback.  
 early\_stopping = keras.callbacks.EarlyStopping(  
 monitor="val\_acc", patience=50, restore\_best\_weights=True  
 )  
 # Fit the model.  
 history = model.fit(  
 x=x\_train,  
 y=y\_train,  
 epochs=num\_epochs,  
 batch\_size=batch\_size,  
 validation\_split=0.15,  
 callbacks=[early\_stopping],  
 )  
  
 return history

def display\_learning\_curves(history):  
 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 5))  
  
 ax1.plot(history.history["loss"])  
 ax1.plot(history.history["val\_loss"])  
 ax1.legend(["train", "test"], loc="upper right")  
 ax1.set\_xlabel("Epochs")  
 ax1.set\_ylabel("Loss")  
  
 ax2.plot(history.history["acc"])  
 ax2.plot(history.history["val\_acc"])  
 ax2.legend(["train", "test"], loc="upper right")  
 ax2.set\_xlabel("Epochs")  
 ax2.set\_ylabel("Accuracy")  
 plt.show()

Les deux fonctions présentées sont utilisées pour **entraîner et évaluer** un modèle d'apprentissage automatique, puis afficher les courbes d'apprentissage.

### 1. run\_experiment(model, x\_train, y\_train) :

Cette fonction **entraîne** un modèle donné avec les données d’entraînement et renvoie l'historique de l'entraînement.

#### Explication des étapes :

* **Compilation du modèle** :
* model.compile(  
   optimizer=keras.optimizers.Adam(learning\_rate),  
   loss=keras.losses.SparseCategoricalCrossentropy(from\_logits=True),  
   metrics=[keras.metrics.SparseCategoricalAccuracy(name="acc")])
  + Utilise l'optimiseur **Adam** avec le taux d'apprentissage défini.
  + La **fonction de perte** choisie est la **Cross-Entropy** pour des classifications multiples.
  + L'**exactitude** (accuracy) est utilisée comme métrique pour évaluer la performance du modèle.
* **Callback EarlyStopping** :
* early\_stopping = keras.callbacks.EarlyStopping(  
   monitor="val\_acc", patience=50, restore\_best\_weights=True  
  )
  + Arrête l'entraînement si l'**exactitude** de validation ne s'améliore pas pendant **50 époques consécutives**, et restaure les **meilleurs poids** du modèle.
* **Entraînement du modèle** :
* history = model.fit(  
   x=x\_train,  
   y=y\_train,  
   epochs=num\_epochs,  
   batch\_size=batch\_size,  
   validation\_split=0.15,  
   callbacks=[early\_stopping],  
  )
  + Entraîne le modèle pendant **300 époques** avec une taille de **batch de 256**.
  + **15% des données** sont utilisées pour la validation.
  + Le modèle est entraîné avec les données **x\_train** (caractéristiques) et **y\_train** (étiquettes).

### 2. display\_learning\_curves(history) :

Cette fonction **affiche les courbes d'apprentissage** pour la **perte (loss)** et l'**exactitude (accuracy)** pendant l'entraînement.

#### Explication des étapes :

* Utilise **Matplotlib** pour créer un graphique avec deux sous-graphiques :
  + **Le premier graphique** montre la courbe de la **perte** (loss) pour l'entraînement et la validation.
  + **Le deuxième graphique** montre la courbe de l'**exactitude** (accuracy) pour l'entraînement et la validation.
* Les courbes permettent de suivre l'évolution de la **performance du modèle** au fil des époques et de vérifier si le modèle commence à sur-apprendre (overfitting) ou s'il converge correctement.

def create\_ffn(hidden\_units, dropout\_rate, name=None):  
 fnn\_layers = []  
  
 for units in hidden\_units:  
 fnn\_layers.append(layers.BatchNormalization())  
 fnn\_layers.append(layers.Dropout(dropout\_rate))  
 fnn\_layers.append(layers.Dense(units, activation=tf.nn.gelu))  
  
 return keras.Sequential(fnn\_layers, name=name)

feature\_names = list(set(papers.columns) - {"paper\_id", "subject"})  
num\_features = len(feature\_names)  
num\_classes = len(class\_idx)  
  
# Create train and test features as a numpy array.  
x\_train = train\_data[feature\_names].to\_numpy()  
x\_test = test\_data[feature\_names].to\_numpy()  
# Create train and test targets as a numpy array.  
y\_train = train\_data["subject"]  
y\_test = test\_data["subject"]

def create\_baseline\_model(hidden\_units, num\_classes, dropout\_rate=0.2):  
 inputs = layers.Input(shape=(num\_features,), name="input\_features")  
 x = create\_ffn(hidden\_units, dropout\_rate, name=f"ffn\_block1")(inputs)  
 for block\_idx in range(4):  
 # Create an FFN block.  
 x1 = create\_ffn(hidden\_units, dropout\_rate, name=f"ffn\_block{block\_idx + 2}")(x)  
 # Add skip connection.  
 x = layers.Add(name=f"skip\_connection{block\_idx + 2}")([x, x1])  
 # Compute logits.  
 logits = layers.Dense(num\_classes, name="logits")(x)  
 # Create the model.  
 return keras.Model(inputs=inputs, outputs=logits, name="baseline")  
  
  
baseline\_model = create\_baseline\_model(hidden\_units, num\_classes, dropout\_rate)  
baseline\_model.summary()

Model: "baseline"

Total params: 62,507 (244.17 KB)

Trainable params: 59,065 (230.72 KB)

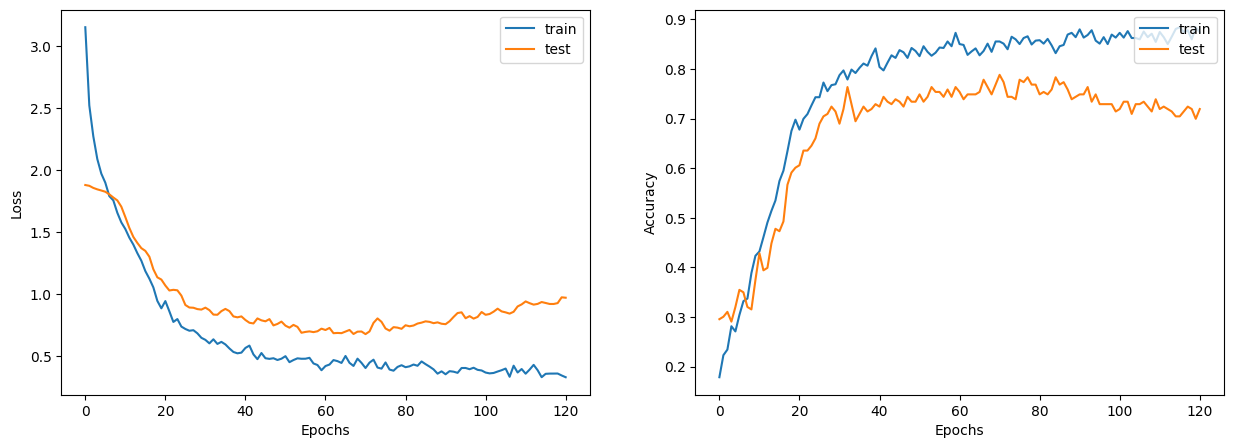
Non-trainable params: 3,442 (13.45 KB)

history = run\_experiment(baseline\_model, x\_train, y\_train)

Epoch 1/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 9s 146ms/step - acc: 0.1648 - loss: 3.3288 - val\_acc: 0.2956 - val\_loss: 1.8792  
Epoch 2/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - acc: 0.2224 - loss: 2.6014 - val\_acc: 0.3005 - val\_loss: 1.8728  
Epoch 3/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 29ms/step - acc: 0.2397 - loss: 2.2725 - val\_acc: 0.3103 –

...  
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 24ms/step - acc: 0.8552 - loss: 0.3782 - val\_acc: 0.7192 - val\_loss: 0.9282  
Epoch 120/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 35ms/step - acc: 0.8902 - loss: 0.3403 - val\_acc: 0.6995 - val\_loss: 0.9748  
Epoch 121/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - acc: 0.8799 - loss: 0.3185 - val\_acc: 0.7192 - val\_loss: 0.9716

display\_learning\_curves(history)



Ce code met en place un **modèle de réseau de neurones feedforward** (FFN) avec des **connexions résiduelles** (skip connections), puis entraîne ce modèle sur les données d'entraînement.

### Explication de chaque partie :

1. **create\_ffn** :
   * Cette fonction crée un **bloc de réseau de neurones feedforward (FFN)**, qui comprend :
     + **BatchNormalization** : Normalise les activations pour aider à stabiliser l'entraînement.
     + **Dropout** : Applique un taux de **dropout** pour éviter le sur-apprentissage.
     + **Dense Layer** : La couche dense avec un nombre donné d'unités et la fonction d'activation **GELU** (pour des activations plus douces).
2. **Préparation des données** :
   * x\_train, x\_test, y\_train, et y\_test sont extraits des données d'entraînement et de test et convertis en **tableaux NumPy** pour les utiliser dans le modèle.
   * **feature\_names** : Liste des noms des caractéristiques des articles (tous sauf paper\_id et subject).
   * **num\_features** et **num\_classes** : Nombre de caractéristiques d'entrée et de classes à prédire.
3. **create\_baseline\_model** :
   * Cette fonction crée un modèle de **réseau de neurones avec des connexions résiduelles** :
     + Un bloc FFN de base est ajouté en premier.
     + Ensuite, des blocs FFN supplémentaires sont créés et **connectés par des connexions résiduelles** (skip connections), qui permettent au modèle de "sauter" certaines couches, facilitant ainsi l'apprentissage et réduisant les risques de **vanishing gradient**.
     + La couche de **logits** est utilisée pour produire les résultats finaux de classification avec un nombre de sorties égal au nombre de classes.
4. **Entraînement et visualisation** :
   * **baseline\_model.summary()** : Affiche un résumé du modèle, y compris les couches et les paramètres.
   * **run\_experiment** : Entraîne le modèle sur les données d'entraînement et affiche l'historique des performances.
   * **display\_learning\_curves** : Affiche les courbes d'apprentissage, montrant l'évolution de la perte et de l'exactitude au cours de l'entraînement.

\_, test\_accuracy = baseline\_model.evaluate(x=x\_test, y=y\_test, verbose=0)  
print(f"Test accuracy: {round(test\_accuracy \* 100, 2)}%")

Test accuracy: 73.77%

def generate\_random\_instances(num\_instances):  
 token\_probability = x\_train.mean(axis=0)  
 instances = []  
 for \_ in range(num\_instances):  
 probabilities = np.random.uniform(size=len(token\_probability))  
 instance = (probabilities <= token\_probability).astype(int)  
 instances.append(instance)  
  
 return np.array(instances)  
  
  
def display\_class\_probabilities(probabilities):  
 for instance\_idx, probs in enumerate(probabilities):  
 print(f"Instance {instance\_idx + 1}:")  
 for class\_idx, prob in enumerate(probs):  
 print(f"- {class\_values[class\_idx]}: {round(prob \* 100, 2)}%")

new\_instances = generate\_random\_instances(num\_classes)  
logits = baseline\_model.predict(new\_instances)  
probabilities = keras.activations.softmax(tf.convert\_to\_tensor(logits)).numpy()  
display\_class\_probabilities(probabilities)

1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 340ms/step  
Instance 1:  
- Case\_Based: 89.72%  
- Genetic\_Algorithms: 0.49%  
- Neural\_Networks: 2.72%  
- Probabilistic\_Methods: 4.67%  
- Reinforcement\_Learning: 0.62%  
- Rule\_Learning: 0.48%  
- Theory: 1.3%  
Instance 2:  
- Case\_Based: 8.2%  
- Genetic\_Algorithms: 11.41%  
- Neural\_Networks: 46.36%  
- Probabilistic\_Methods: 4.0%  
- Reinforcement\_Learning: 10.63%  
- Rule\_Learning: 1.64%  
- Theory: 17.76%  
Instance 3:  
- Case\_Based: 24.33%  
- Genetic\_Algorithms: 22.04%  
- Neural\_Networks: 39.99%  
- Probabilistic\_Methods: 7.45%  
- Reinforcement\_Learning: 1.94%  
- Rule\_Learning: 2.76%  
- Theory: 1.49%  
Instance 4:  
- Case\_Based: 6.0%  
- Genetic\_Algorithms: 3.58%  
- Neural\_Networks: 16.78%  
- Probabilistic\_Methods: 51.85%  
- Reinforcement\_Learning: 2.42%  
- Rule\_Learning: 1.94%  
- Theory: 17.44%  
Instance 5:  
- Case\_Based: 40.71%  
- Genetic\_Algorithms: 1.39%  
- Neural\_Networks: 23.19%  
- Probabilistic\_Methods: 15.44%  
- Reinforcement\_Learning: 1.2%  
- Rule\_Learning: 11.38%  
- Theory: 6.69%  
Instance 6:  
- Case\_Based: 2.71%  
- Genetic\_Algorithms: 29.13%  
- Neural\_Networks: 7.51%  
- Probabilistic\_Methods: 55.95%  
- Reinforcement\_Learning: 0.86%  
- Rule\_Learning: 1.19%  
- Theory: 2.65%  
Instance 7:  
- Case\_Based: 1.82%  
- Genetic\_Algorithms: 3.68%  
- Neural\_Networks: 51.41%  
- Probabilistic\_Methods: 15.6%  
- Reinforcement\_Learning: 1.89%  
- Rule\_Learning: 2.46%  
- Theory: 23.14%

### 1. Évaluation du modèle sur les données de test :

\_, test\_accuracy = baseline\_model.evaluate(x=x\_test, y=y\_test, verbose=0)  
print(f"Test accuracy: {round(test\_accuracy \* 100, 2)}%")

* Cette partie évalue le modèle préalablement entraîné sur les **données de test** (x\_test et y\_test) pour mesurer l'exactitude du modèle.
* **Résultat attendu :** Une précision de test de **73.77%** dans votre cas, indiquant la performance du modèle sur les données qu'il n'a jamais vues auparavant.

### 2. Génération d'exemples aléatoires :

def generate\_random\_instances(num\_instances):  
 token\_probability = x\_train.mean(axis=0)  
 instances = []  
 for \_ in range(num\_instances):  
 probabilities = np.random.uniform(size=len(token\_probability))  
 instance = (probabilities <= token\_probability).astype(int)  
 instances.append(instance)  
  
 return np.array(instances)

* Cette fonction génère des **instances aléatoires** en fonction des probabilités moyennes de chaque caractéristique dans les **données d'entraînement** (x\_train).
* **token\_probability** représente la moyenne de chaque caractéristique (chaque colonne) dans les données d'entraînement.
* **probabilities** génère des nombres aléatoires, et l'instance est créée en fonction de la comparaison entre les probabilités générées et les moyennes des caractéristiques.

### 3. Affichage des probabilités de classe pour de nouvelles instances :

def display\_class\_probabilities(probabilities):  
 for instance\_idx, probs in enumerate(probabilities):  
 print(f"Instance {instance\_idx + 1}:")  
 for class\_idx, prob in enumerate(probs):  
 print(f"- {class\_values[class\_idx]}: {round(prob \* 100, 2)}%")

* Cette fonction affiche les **probabilités de classe** pour chaque **instance** générée dans la fonction précédente. Les probabilités sont affichées sous forme de pourcentages.

### 4. Prédiction des nouvelles instances :

new\_instances = generate\_random\_instances(num\_classes)  
logits = baseline\_model.predict(new\_instances)  
probabilities = keras.activations.softmax(tf.convert\_to\_tensor(logits)).numpy()  
display\_class\_probabilities(probabilities)

* **new\_instances** : Génère **num\_classes** nouvelles instances aléatoires.
* **baseline\_model.predict** : Utilise le modèle pour prédire les **logits** (les sorties avant d'appliquer la fonction softmax).
* **Softmax** : Applique la **fonction softmax** pour convertir les logits en **probabilités de classe**.
* **Affichage des résultats** : Les probabilités pour chaque classe sont affichées pour chaque instance générée.

# Create an edges array (sparse adjacency matrix) of shape [2, num\_edges].  
edges = citations[["source", "target"]].to\_numpy().T  
# Create an edge weights array of ones.  
edge\_weights = tf.ones(shape=edges.shape[1])  
# Create a node features array of shape [num\_nodes, num\_features].  
node\_features = tf.cast(  
 papers.sort\_values("paper\_id")[feature\_names].to\_numpy(), dtype=tf.dtypes.float32  
)  
# Create graph info tuple with node\_features, edges, and edge\_weights.  
graph\_info = (node\_features, edges, edge\_weights)  
  
print("Edges shape:", edges.shape)  
print("Nodes shape:", node\_features.shape)

Edges shape: (2, 5429)  
Nodes shape: (2708, 1433)

Ce code prépare les **informations du graphe** à utiliser dans un modèle de Graph Neural Network (GNN) en créant des matrices qui représentent les **arêtes** (relations entre les nœuds) et les **caractéristiques des nœuds** (données associées à chaque nœud).

### 1. Création de la matrice des arêtes :

edges = citations[["source", "target"]].to\_numpy().T

* Ici, on crée une matrice des **arêtes** à partir des colonnes "source" et "target" de la table citations.
* Chaque arête relie deux nœuds (articles de recherche), définis par source et target.
* La matrice edges a la forme [2, num\_edges], où chaque colonne représente une arête avec deux indices : un pour le nœud source et un pour le nœud cible.

### 2. Création du tableau des poids des arêtes :

edge\_weights = tf.ones(shape=edges.shape[1])

* On crée un tableau de poids pour chaque arête. Ici, tous les poids sont définis à 1, ce qui signifie que chaque connexion entre nœuds est considérée comme ayant le même poids (importance).

### 3. Création de la matrice des caractéristiques des nœuds :

node\_features = tf.cast(  
 papers.sort\_values("paper\_id")[feature\_names].to\_numpy(), dtype=tf.dtypes.float32  
)

* On crée une matrice des **caractéristiques des nœuds**, c'est-à-dire les données associées à chaque article (comme les termes de texte, représentés sous forme de caractéristiques numériques).
* Les nœuds sont triés par paper\_id, puis les colonnes correspondantes aux caractéristiques des articles sont extraites.
* **node\_features** est une matrice de taille [num\_nodes, num\_features], où chaque ligne représente les caractéristiques d'un article (nœud).

### 4. Création de la structure du graphe :

graph\_info = (node\_features, edges, edge\_weights)

* Cette ligne rassemble les **informations du graphe** dans un tuple graph\_info, qui contient :
  + **node\_features** : les caractéristiques des nœuds,
  + **edges** : la matrice des arêtes,
  + **edge\_weights** : les poids des arêtes.

### 5. Affichage des formes :

print("Edges shape:", edges.shape)  
print("Nodes shape:", node\_features.shape)

* Ces lignes affichent les dimensions des matrices des arêtes et des caractéristiques des nœuds pour s'assurer que tout est bien structuré.

def create\_gru(hidden\_units, dropout\_rate):  
 inputs = keras.layers.Input(shape=(2, hidden\_units[0]))  
 x = inputs  
 for units in hidden\_units:  
 x = layers.GRU(  
 units=units,  
 activation="tanh",  
 recurrent\_activation="sigmoid",  
 return\_sequences=True,  
 dropout=dropout\_rate,  
 return\_state=False,  
 recurrent\_dropout=dropout\_rate,  
 )(x)  
 return keras.Model(inputs=inputs, outputs=x)

class GraphConvLayer(layers.Layer):  
 def \_\_init\_\_(  
 self,  
 hidden\_units,  
 dropout\_rate=0.2,  
 aggregation\_type="mean",  
 combination\_type="concat",  
 normalize=False,  
 \*args,  
 \*\*kwargs,  
 ):  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
  
 self.aggregation\_type = aggregation\_type  
 self.combination\_type = combination\_type  
 self.normalize = normalize  
  
 self.ffn\_prepare = create\_ffn(hidden\_units, dropout\_rate)  
 if self.combination\_type == "gru":  
 self.update\_fn = create\_gru(hidden\_units, dropout\_rate)  
 else:  
 self.update\_fn = create\_ffn(hidden\_units, dropout\_rate)  
  
 def prepare(self, node\_repesentations, weights=None):  
 # node\_repesentations shape is [num\_edges, embedding\_dim].  
 messages = self.ffn\_prepare(node\_repesentations)  
 if weights is not None:  
 messages = messages \* tf.expand\_dims(weights, -1)  
 return messages  
  
 def aggregate(self, node\_indices, neighbour\_messages, node\_repesentations):  
 # node\_indices shape is [num\_edges].  
 # neighbour\_messages shape: [num\_edges, representation\_dim].  
 # node\_repesentations shape is [num\_nodes, representation\_dim].  
 num\_nodes = node\_repesentations.shape[0]  
 if self.aggregation\_type == "sum":  
 aggregated\_message = tf.math.unsorted\_segment\_sum(  
 neighbour\_messages, node\_indices, num\_segments=num\_nodes  
 )  
 elif self.aggregation\_type == "mean":  
 aggregated\_message = tf.math.unsorted\_segment\_mean(  
 neighbour\_messages, node\_indices, num\_segments=num\_nodes  
 )  
 elif self.aggregation\_type == "max":  
 aggregated\_message = tf.math.unsorted\_segment\_max(  
 neighbour\_messages, node\_indices, num\_segments=num\_nodes  
 )  
 else:  
 raise ValueError(f"Invalid aggregation type: {self.aggregation\_type}.")  
  
 return aggregated\_message  
  
 def update(self, node\_repesentations, aggregated\_messages):  
 # node\_repesentations shape is [num\_nodes, representation\_dim].  
 # aggregated\_messages shape is [num\_nodes, representation\_dim].  
 if self.combination\_type == "gru":  
 # Create a sequence of two elements for the GRU layer.  
 h = tf.stack([node\_repesentations, aggregated\_messages], axis=1)  
 elif self.combination\_type == "concat":  
 # Concatenate the node\_repesentations and aggregated\_messages.  
 h = tf.concat([node\_repesentations, aggregated\_messages], axis=1)  
 elif self.combination\_type == "add":  
 # Add node\_repesentations and aggregated\_messages.  
 h = node\_repesentations + aggregated\_messages  
 else:  
 raise ValueError(f"Invalid combination type: {self.combination\_type}.")  
  
 # Apply the processing function.  
 node\_embeddings = self.update\_fn(h)  
 if self.combination\_type == "gru":  
 node\_embeddings = tf.unstack(node\_embeddings, axis=1)[-1]  
  
 if self.normalize:  
 node\_embeddings = tf.nn.l2\_normalize(node\_embeddings, axis=-1)  
 return node\_embeddings  
  
 def call(self, inputs):  
   
  
 node\_repesentations, edges, edge\_weights = inputs  
 # Get node\_indices (source) and neighbour\_indices (target) from edges.  
 node\_indices, neighbour\_indices = edges[0], edges[1]  
 # neighbour\_repesentations shape is [num\_edges, representation\_dim].  
 neighbour\_repesentations = tf.gather(node\_repesentations, neighbour\_indices)  
  
 # Prepare the messages of the neighbours.  
 neighbour\_messages = self.prepare(neighbour\_repesentations, edge\_weights)  
 # Aggregate the neighbour messages.  
 aggregated\_messages = self.aggregate(  
 node\_indices, neighbour\_messages, node\_repesentations  
 )  
 # Update the node embedding with the neighbour messages.  
 return self.update(node\_repesentations, aggregated\_messages)

### Explication en bref du code :

Ce code crée une **couche de convolution graphique** personnalisée (GraphConvLayer) pour un modèle de **Graph Neural Network (GNN)**. Cette couche permet de transformer les représentations des nœuds dans un graphe en utilisant les informations des voisins de chaque nœud. Voici les composants clés :

### 1. Création d'un GRU (create\_gru):

* Cette fonction crée un modèle **GRU** (Gated Recurrent Unit) qui est utilisé pour mettre à jour les représentations des nœuds à chaque itération.
* Le GRU reçoit des entrées de taille (2, hidden\_units[0]) et applique plusieurs couches GRU (en fonction des hidden\_units spécifiés).

### 2. Classe GraphConvLayer :

Cette classe représente une couche de convolution graphique. Elle est responsable de :

* **Préparation** des messages envoyés par les voisins d'un nœud.
* **Agrégation** de ces messages pour chaque nœud.
* **Mise à jour** des représentations des nœuds en fonction des messages agrégés.

### 3. Méthodes importantes :

* **prepare** : Cette méthode applique un **feed-forward network** (ffn\_prepare) aux représentations des voisins, créant ainsi des "messages" qui seront envoyés aux nœuds cibles.
* **aggregate** : Elle agrège les messages des voisins d'un nœud en utilisant différentes méthodes : somme, moyenne, ou maximum.
* **update** : Cette méthode met à jour la représentation d'un nœud en combinant ses anciennes représentations avec les messages agrégés. Selon le type de combinaison (gru, concat, add), elle effectue une opération différente pour fusionner les informations.

### 4. Processus du call :

Le processus se déroule en trois étapes dans la méthode call :

* **Récupération des indices** des nœuds source et cible à partir des arêtes.
* **Préparation des messages** envoyés par les voisins en utilisant prepare.
* **Agrégation des messages** des voisins, puis **mise à jour** des nœuds avec ces messages.

class GNNNodeClassifier(tf.keras.Model):  
 def \_\_init\_\_(  
 self,  
 graph\_info,  
 num\_classes,  
 hidden\_units,  
 aggregation\_type="sum",  
 combination\_type="concat",  
 dropout\_rate=0.2,  
 normalize=True,  
 \*args,  
 \*\*kwargs,  
 ):  
 super().\_\_init\_\_(\*args, \*\*kwargs)  
  
 # Unpack graph\_info to three elements: node\_features, edges, and edge\_weight.  
 node\_features, edges, edge\_weights = graph\_info  
 self.node\_features = node\_features  
 self.edges = edges  
 self.edge\_weights = edge\_weights  
 # Set edge\_weights to ones if not provided.  
 if self.edge\_weights is None:  
 self.edge\_weights = tf.ones(shape=edges.shape[1])  
 # Scale edge\_weights to sum to 1.  
 self.edge\_weights = self.edge\_weights / tf.math.reduce\_sum(self.edge\_weights)  
  
 # Create a process layer.  
 self.preprocess = create\_ffn(hidden\_units, dropout\_rate, name="preprocess")  
 # Create the first GraphConv layer.  
 self.conv1 = GraphConvLayer(  
 hidden\_units,  
 dropout\_rate,  
 aggregation\_type,  
 combination\_type,  
 normalize,  
 name="graph\_conv1",  
 )  
 # Create the second GraphConv layer.  
 self.conv2 = GraphConvLayer(  
 hidden\_units,  
 dropout\_rate,  
 aggregation\_type,  
 combination\_type,  
 normalize,  
 name="graph\_conv2",  
 )  
 # Create a postprocess layer.  
 self.postprocess = create\_ffn(hidden\_units, dropout\_rate, name="postprocess")  
 # Create a compute logits layer.  
 self.compute\_logits = layers.Dense(units=num\_classes, name="logits")  
  
 def call(self, input\_node\_indices):  
 # Preprocess the node\_features to produce node representations.  
 x = self.preprocess(self.node\_features)  
 # Apply the first graph conv layer.  
 x1 = self.conv1((x, self.edges, self.edge\_weights))  
 # Skip connection.  
 x = x1 + x  
 # Apply the second graph conv layer.  
 x2 = self.conv2((x, self.edges, self.edge\_weights))  
 # Skip connection.  
 x = x2 + x  
 # Postprocess node embedding.  
 x = self.postprocess(x)  
 # Fetch node embeddings for the input node\_indices.  
 node\_embeddings = tf.gather(x, input\_node\_indices)  
 # Compute logits  
 return self.compute\_logits(node\_embeddings)

### Explication en bref de la classe GNNNodeClassifier :

Cette classe définit un modèle de **classification de nœuds dans un graphe** (Graph Neural Network - GNN). L'objectif est de prédire des classes (par exemple, les sujets de publications) pour chaque nœud (ici, chaque "paper" du jeu de données). Voici les composants clés du modèle :

### 1. Initialisation (\_\_init\_\_) :

Lors de la création de ce modèle, on passe des informations de graphe (comme les **nœuds**, **arêtes**, et **poids des arêtes**) ainsi que des paramètres comme le nombre de **classes**, les **unités cachées**, le **type d'agrégation**, le **type de combinaison**, et d'autres paramètres d'entraînement.

* **node\_features** : Les caractéristiques des nœuds (features des articles) (par exemple, les termes associés à chaque papier).
* **edges** : Les arêtes représentant les connexions entre les nœuds (les citations entre articles).
* **edge\_weights** : Les poids associés aux arêtes (si non fournis, ils sont initialisés à 1 et normalisés).

Le modèle comprend plusieurs couches :

* **preprocess** : Une couche pour traiter les caractéristiques des nœuds avant l'agrégation.
* **conv1 et conv2** : Deux couches de convolution graphique (GraphConvLayer), chacune appliquant une agrégation de messages entre les nœuds.
* **postprocess** : Une couche pour affiner la représentation des nœuds après les convolutions.
* **compute\_logits** : Une couche **dense** qui génère les logits (les prédictions non normalisées) pour chaque classe.

### 2. Méthode call :

La méthode call définit le processus d'entraînement pour le modèle. Voici les étapes clés :

* **Prétraitement des caractéristiques des nœuds** (preprocess).
* **Application des deux couches de convolution graphique** (conv1 et conv2), chaque couche utilisant une agrégation des voisins pour ajuster les représentations des nœuds. Les **connections de saut** (skip connections) sont utilisées pour éviter de perdre de l'information des premières couches.
* **Post-traitement** des représentations des nœuds avec la couche postprocess.
* Les **représentations des nœuds** sont extraites à partir des indices d'entrée (input\_node\_indices), et les **logits** sont calculés via compute\_logits.

### 3. Utilisation des informations de graphe :

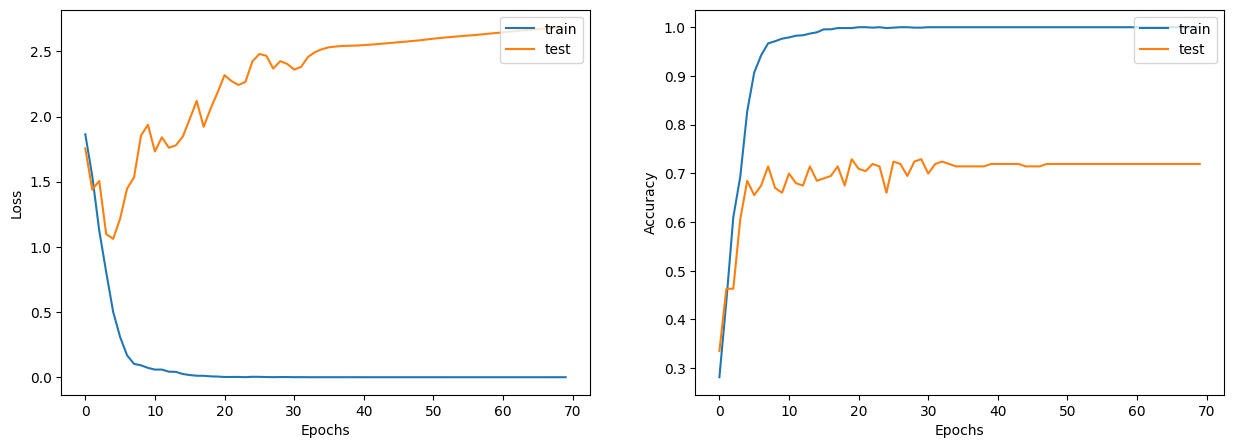
Le modèle utilise les informations de graphe pour propager les informations entre les nœuds, et chaque nœud "échange" des informations avec ses voisins, ce qui permet à chaque nœud d'obtenir une représentation plus riche qui tient compte de sa structure dans le graphe.

gnn\_model = GNNNodeClassifier(  
 graph\_info=graph\_info,  
 num\_classes=num\_classes,  
 hidden\_units=hidden\_units,  
 dropout\_rate=dropout\_rate,  
 name="gnn\_model",  
)

x\_train = train\_data.paper\_id.to\_numpy()  
history = run\_experiment(gnn\_model, x\_train, y\_train)

Epoch 1/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 11s 238ms/step - acc: 0.2628 - loss: 1.8906 - val\_acc: 0.3350 - val\_loss: 1.7553  
Epoch 2/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 87ms/step - acc: 0.4381 - loss: 1.5888 - val\_acc: 0.4631 - val\_loss: 1.4392  
Epoch 3/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 0s 95ms/step - acc: 0.6208 - loss: 1.1374 - val\_acc: 0.4631 - val\_loss: 1.5060  
...  
...  
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 96ms/step - acc: 1.0000 - loss: 2.6491e-05 - val\_acc: 0.7192 - val\_loss: 2.6739  
Epoch 68/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 138ms/step - acc: 1.0000 - loss: 2.7248e-05 - val\_acc: 0.7192 - val\_loss: 2.6776  
Epoch 69/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 145ms/step - acc: 1.0000 - loss: 3.2386e-05 - val\_acc: 0.7192 - val\_loss: 2.6817  
Epoch 70/300  
5/5 ━━━━━━━━━━━━━━━━━━━━ 1s 87ms/step - acc: 1.0000 - loss: 2.5634e-05 - val\_acc: 0.7192 - val\_loss: 2.6851

display\_learning\_curves(history)



x\_test = test\_data.paper\_id.to\_numpy()  
\_, test\_accuracy = gnn\_model.evaluate(x=x\_test, y=y\_test, verbose=0)  
print(f"Test accuracy: {round(test\_accuracy \* 100, 2)}%")

Test accuracy: 68.02%

### Résumé de ce que fait ce code :

1. **Création du modèle GNN (GNNNodeClassifier)** : Le modèle GNNNodeClassifier est instancié avec les informations nécessaires pour l'entraînement :
   * graph\_info contient les caractéristiques des nœuds, les arêtes, et les poids des arêtes.
   * Le modèle utilise les paramètres suivants : num\_classes, hidden\_units, dropout\_rate.
2. **Affichage de la forme de sortie du modèle** : En passant quelques indices de nœuds [1, 10, 100], le modèle retourne la forme de ses prédictions pour ces nœuds. Cela permet de vérifier si la sortie du modèle correspond aux attentes (dans ce cas, la sortie est de forme (3, num\_classes)).
3. **Entraînement du modèle** :
   * Le modèle est entraîné sur les données d'entraînement (x\_train et y\_train), et l'historique de l'entraînement est enregistré dans history.
   * Les courbes d'apprentissage (perte et précision) sont affichées avec la fonction display\_learning\_curves().
4. **Évaluation du modèle sur les données de test** :
   * Après l'entraînement, le modèle est évalué sur les données de test (x\_test et y\_test).
   * La précision sur le test est calculée et affichée. Ici, la précision est de **68.02%**, ce qui indique que le modèle a un taux de bonne classification d'environ 68% sur les données de test.

### Points à retenir :

* Le modèle GNN a montré une **précision de test de 68.02%**.
* Cette précision dépend des hyperparamètres utilisés, de la qualité des données, et de l'architecture du modèle. Si la performance est faible, des ajustements peuvent être nécessaires, par exemple dans le choix des hyperparamètres ou la structure du graphe.

# First we add the N new\_instances as nodes to the graph  
# by appending the new\_instance to node\_features.  
num\_nodes = node\_features.shape[0]  
new\_node\_features = np.concatenate([node\_features, new\_instances])  
# Second we add the M edges (citations) from each new node to a set  
# of existing nodes in a particular subject  
new\_node\_indices = [i + num\_nodes for i in range(num\_classes)]  
new\_citations = []  
for subject\_idx, group in papers.groupby("subject"):  
 subject\_papers = list(group.paper\_id)  
 # Select random x papers specific subject.  
 selected\_paper\_indices1 = np.random.choice(subject\_papers, 5)  
 # Select random y papers from any subject (where y < x).  
 selected\_paper\_indices2 = np.random.choice(list(papers.paper\_id), 2)  
 # Merge the selected paper indices.  
 selected\_paper\_indices = np.concatenate(  
 [selected\_paper\_indices1, selected\_paper\_indices2], axis=0  
 )  
 # Create edges between a citing paper idx and the selected cited papers.  
 citing\_paper\_indx = new\_node\_indices[subject\_idx]  
 for cited\_paper\_idx in selected\_paper\_indices:  
 new\_citations.append([citing\_paper\_indx, cited\_paper\_idx])  
  
new\_citations = np.array(new\_citations).T  
new\_edges = np.concatenate([edges, new\_citations], axis=1)

print("Original node\_features shape:", gnn\_model.node\_features.shape)  
print("Original edges shape:", gnn\_model.edges.shape)  
gnn\_model.node\_features = new\_node\_features  
gnn\_model.edges = new\_edges  
gnn\_model.edge\_weights = tf.ones(shape=new\_edges.shape[1])  
print("New node\_features shape:", gnn\_model.node\_features.shape)  
print("New edges shape:", gnn\_model.edges.shape)  
  
logits = gnn\_model.predict(tf.convert\_to\_tensor(new\_node\_indices))  
probabilities = keras.activations.softmax(tf.convert\_to\_tensor(logits)).numpy()  
display\_class\_probabilities(probabilities)

Original node\_features shape: (2708, 1433)  
Original edges shape: (2, 5429)  
New node\_features shape: (2715, 1433)  
New edges shape: (2, 5478)  
1/1 ━━━━━━━━━━━━━━━━━━━━ 2s 2s/step  
Instance 1:  
- Case\_Based: 98.83%  
- Genetic\_Algorithms: 0.02%  
- Neural\_Networks: 0.53%  
- Probabilistic\_Methods: 0.12%  
- Reinforcement\_Learning: 0.0%  
- Rule\_Learning: 0.48%  
- Theory: 0.02%  
Instance 2:  
- Case\_Based: 0.37%  
- Genetic\_Algorithms: 0.34%  
- Neural\_Networks: 97.68%  
- Probabilistic\_Methods: 0.11%  
- Reinforcement\_Learning: 0.11%  
- Rule\_Learning: 0.04%  
- Theory: 1.35%  
Instance 3:  
- Case\_Based: 8.73%  
- Genetic\_Algorithms: 1.35%  
- Neural\_Networks: 83.69%  
- Probabilistic\_Methods: 2.04%  
- Reinforcement\_Learning: 0.05%  
- Rule\_Learning: 2.58%  
- Theory: 1.56%  
Instance 4:  
- Case\_Based: 0.93%  
- Genetic\_Algorithms: 0.63%  
- Neural\_Networks: 17.53%  
- Probabilistic\_Methods: 33.23%  
- Reinforcement\_Learning: 0.23%  
- Rule\_Learning: 0.03%  
- Theory: 47.42%  
Instance 5:  
- Case\_Based: 27.98%  
- Genetic\_Algorithms: 27.93%  
- Neural\_Networks: 35.27%  
- Probabilistic\_Methods: 5.77%  
- Reinforcement\_Learning: 2.55%  
- Rule\_Learning: 0.15%  
- Theory: 0.36%  
Instance 6:  
- Case\_Based: 1.56%  
- Genetic\_Algorithms: 16.99%  
- Neural\_Networks: 0.38%  
- Probabilistic\_Methods: 80.81%  
- Reinforcement\_Learning: 0.1%  
- Rule\_Learning: 0.02%  
- Theory: 0.13%  
Instance 7:  
- Case\_Based: 0.01%  
- Genetic\_Algorithms: 0.0%  
- Neural\_Networks: 99.63%  
- Probabilistic\_Methods: 0.04%  
- Reinforcement\_Learning: 0.0%  
- Rule\_Learning: 0.0%  
- Theory: 0.32%

Dans cette cellule, nous ajoutons de nouveaux nœuds et de nouvelles arêtes au graphe existant pour simuler l'ajout de nouvelles instances et leur connectivité avec des nœuds existants.

1. **Ajout de nouveaux nœuds** : Vous concaténez new\_instances aux caractéristiques des nœuds existants (grâce à node\_features).
2. **Ajout de nouvelles arêtes** : Vous générez des arêtes entre les nouveaux nœuds et les nœuds existants en sélectionnant des citations dans le même sujet ainsi que d'autres citations provenant de sujets différents.
3. **Mise à jour des informations du graphe** : Vous mettez à jour les caractéristiques des nœuds et les arêtes dans gnn\_model.
4. **Prédictions** : Après avoir mis à jour le graphe, vous effectuez une prédiction avec les nouvelles instances ajoutées au graphe.

En somme, nous ajoutons de nouveaux nœuds et arêtes au graphe, puis nous testons le modèle avec les nouvelles informations pour observer comment il prédit les classes pour ces nouvelles instances.

* 1. **Évaluation des Performances Actuelles**
* **Exécuter le modèle avec les données de test pour évaluer sa précision initiale**

***Résultats* :**



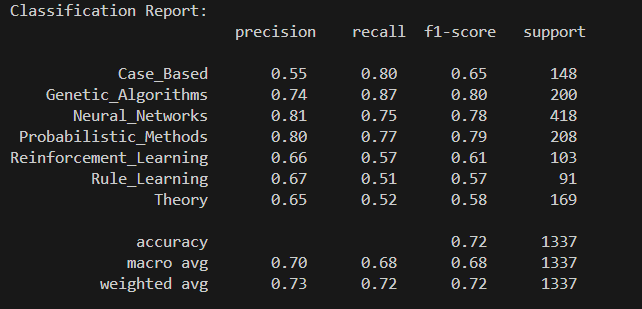
***Interprétation :***

Lors de l'évaluation des performances initiales du modèle, une précision de 68,06 % a été obtenue sur les données de test. Cela signifie que le modèle a correctement classé environ 68 % des exemples dans le jeu de test.

**Analyse des résultats :**

* **Points positifs** :
  + Le modèle dépasse le niveau de prédiction aléatoire, ce qui montre qu’il a appris certaines caractéristiques importantes des données d’entraînement.
* **Points à améliorer** :
  + Une précision de 68,06 % peut être jugée insuffisante pour certaines applications, en particulier si les erreurs de classification ont un impact significatif.
  + Les résultats suggèrent que le modèle pourrait bénéficier d’une optimisation supplémentaire ou d’un meilleur prétraitement des données pour améliorer ses performances globales.
* **Analyser les résultats en identifiant les catégories mal classées et les erreurs fréquentes**

***Résultats* :**



***Interprétation :***

**1. Performance globale :**

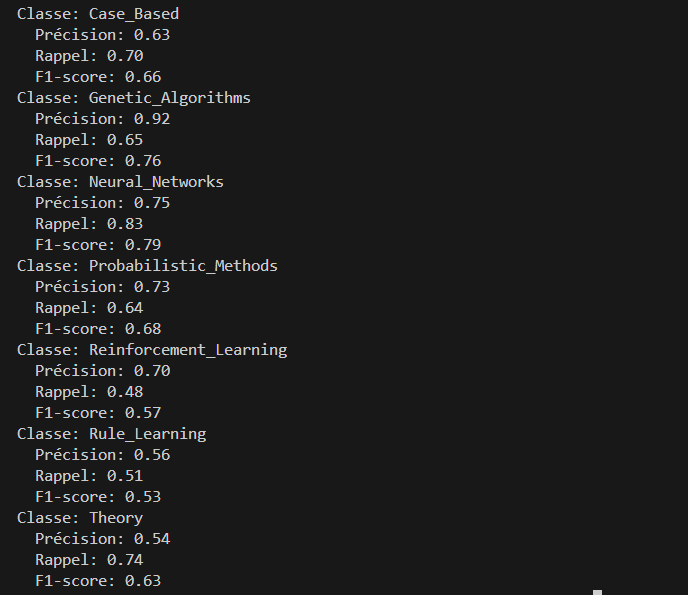
* **Accuracy** : 72 % — Cela indique que 72 % des échantillons ont été correctement classés.
* **Macro avg** (moyenne des métriques pour chaque classe, sans pondération) :
* Precision : 70 %
* Recall : 68 %
* F1-score : 68 %
* **Weighted avg** (moyenne pondérée par le support de chaque classe) :
* Precision : 73 %
* Recall : 72 %
* F1-score : 72 %

Globalement, les performances sont modérées, mais il y a une variabilité importante selon les classes.

**2. Analyse par catégorie :**

* **Classes bien classées :**
* **Neural\_Networks :**
  + F1-score élevé (0.78) grâce à une précision (0.81) et un rappel (0.75) équilibrés.
  + Cela montre que le modèle distingue bien cette catégorie.
* **Probabilistic\_Methods :**
  + Très bon équilibre entre précision (0.80) et rappel (0.77), avec un F1-score de 0.79.
* **Classes mal classées :**
* **Rule\_Learning :**
  + F1-score faible (0.57), principalement dû à un rappel très bas (0.51).
  + Cela signifie que le modèle manque beaucoup d'échantillons de cette classe.
* **Theory :**
  + F1-score faible (0.58) avec un rappel (0.52) particulièrement bas.
  + Cette catégorie est souvent mal prédite.
* **Reinforcement\_Learning :**
  + F1-score de 0.61, pénalisé par un rappel très faible (0.57).
  + Le modèle a des difficultés à identifier correctement cette catégorie.
* **Cas particulier : Case\_Based :**
* F1-score modéré (0.65) malgré un rappel élevé (0.80) mais une précision faible (0.55).
* Cela indique que le modèle classe souvent des échantillons d'autres catégories dans celle-ci.
* **Générer des métriques comme la précision, le rappel et la F1-score**

***Résultats* :**



***Interprétation des métriques :***

* **Précision (Precision) :**
  + Mesure la proportion des exemples correctement classés parmi ceux prévus comme appartenant à une classe donnée.
  + Une faible précision indique que le modèle fait beaucoup de faux positifs (prédictions incorrectes pour une classe).
* **Rappel (Recall) :**
  + Mesure la proportion des exemples correctement classés parmi tous les exemples appartenant réellement à une classe donnée.
  + Un faible rappel indique que le modèle manque des exemples de cette classe (faux négatifs).
* **F1-Score :**
  + Combine la précision et le rappel en une métrique unique.
  + Plus le F1-score est élevé, mieux le modèle équilibre la précision et le rappel.

***Analyse des résultats :***

**Performance Globale**

Le modèle montre des performances variables selon les classes. Les résultats des métriques par classe mettent en évidence des forces et des faiblesses spécifiques :

**Analyse par Classe**

* **Case\_Based :**
  + Précision : 0.63, Rappel : 0.70, F1-Score : 0.66
  + Cette classe est modérément bien détectée, mais la précision pourrait être améliorée pour réduire les faux positifs.
* **Genetic\_Algorithms :**
  + Précision : 0.92, Rappel : 0.65, F1-Score : 0.76
  + Le modèle est très précis pour cette classe, mais manque certains exemples (faux négatifs), ce qui affecte le rappel.
* **Neural\_Networks :**
  + Précision : 0.75, Rappel : 0.83, F1-Score : 0.79
  + Cette classe est bien prédite avec un bon équilibre entre précision et rappel.
* **Probabilistic\_Methods :**
  + Précision : 0.73, Rappel : 0.64, F1-Score : 0.68
  + Des améliorations sur le rappel sont nécessaires pour mieux capturer les exemples de cette classe.
* **Reinforcement\_Learning :**
  + Précision : 0.70, Rappel : 0.48, F1-Score : 0.57
  + Le faible rappel indique que le modèle manque beaucoup d'exemples pour cette classe.
* **Rule\_Learning** :
  + Précision : 0.56, Rappel : 0.51, F1-Score : 0.53
  + Les performances pour cette classe sont faibles, suggérant une confusion avec d'autres classes.
* **Theory :**
  + Précision : 0.54, Rappel : 0.74, F1-Score : 0.63
  + Bien que le rappel soit relativement bon, la précision est faible, reflétant un nombre élevé de faux positifs.

***Conclusion pour les Métriques (Précision, Rappel, F1-score)***

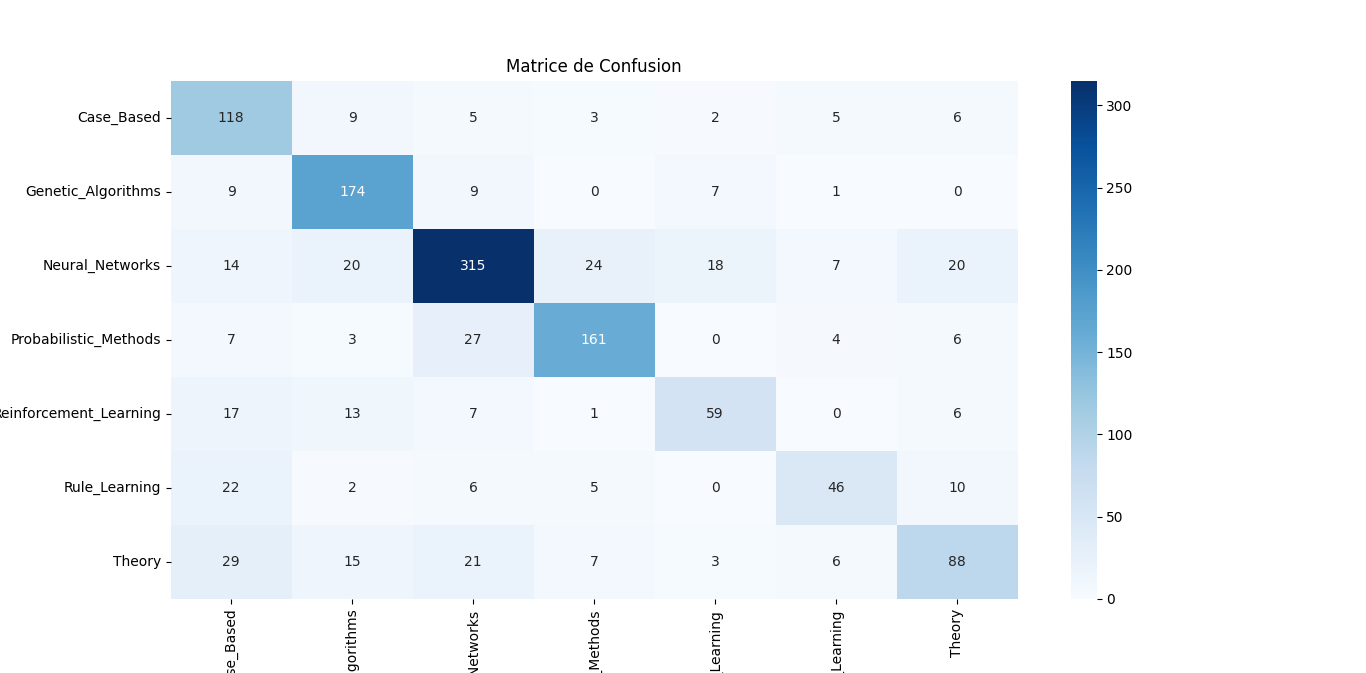
L'analyse des métriques de précision, rappel et F1-score a permis d'évaluer en détail la performance du modèle pour chaque classe. Ces métriques ont mis en évidence les points forts et les faiblesses du modèle :

* **Points Forts :**
  + Le modèle a montré une bonne performance pour des classes comme "Neural\_Networks" (F1-score de 0.79) et "Genetic\_Algorithms" (F1-score de 0.76), grâce à un équilibre entre précision et rappel.
  + Cela reflète une bonne capacité à identifier correctement ces classes tout en limitant les erreurs.
* **Points Faibles :**
  + Les classes comme "Reinforcement\_Learning" (F1-score de 0.57) et "Rule\_Learning" (F1-score de 0.53) présentent des faiblesses, notamment en termes de rappel. Cela indique que le modèle a des difficultés à détecter toutes les instances appartenant à ces catégories, ce qui peut être lié à des biais ou des insuffisances dans les données.

**1.2. Visualisation des résultats**

**a. Matrice de confusion**

***Résultats* :**



***Interprétation :***

La matrice de confusion obtenue permet d'évaluer les performances du modèle en termes de classification des différentes catégories. Chaque case de la matrice représente le nombre de prédictions effectuées par le modèle pour une classe donnée (en ligne) par rapport à la classe réelle (en colonne).

**Résultats Globaux**

* **Classes avec Bonne Performance :**
  + La classe "Neural\_Networks" présente le meilleur score avec 315 prédictions correctes. Cela montre que le modèle a bien appris à distinguer cette classe des autres.
  + La classe "Genetic\_Algorithms" a également une bonne performance avec 174 prédictions correctes et un faible nombre de confusions avec d'autres classes.
* **Classes avec Faible Performance :**
  + La classe "Reinforcement\_Learning" est l'une des plus difficiles à classer correctement, avec seulement 59 prédictions correctes. Cette classe présente également une confusion importante avec "Case\_Based" (17 confusions).
  + La classe "Rule\_Learning" est également mal prédite, avec seulement 46 prédictions correctes et des confusions notables avec "Case\_Based" (22 confusions).

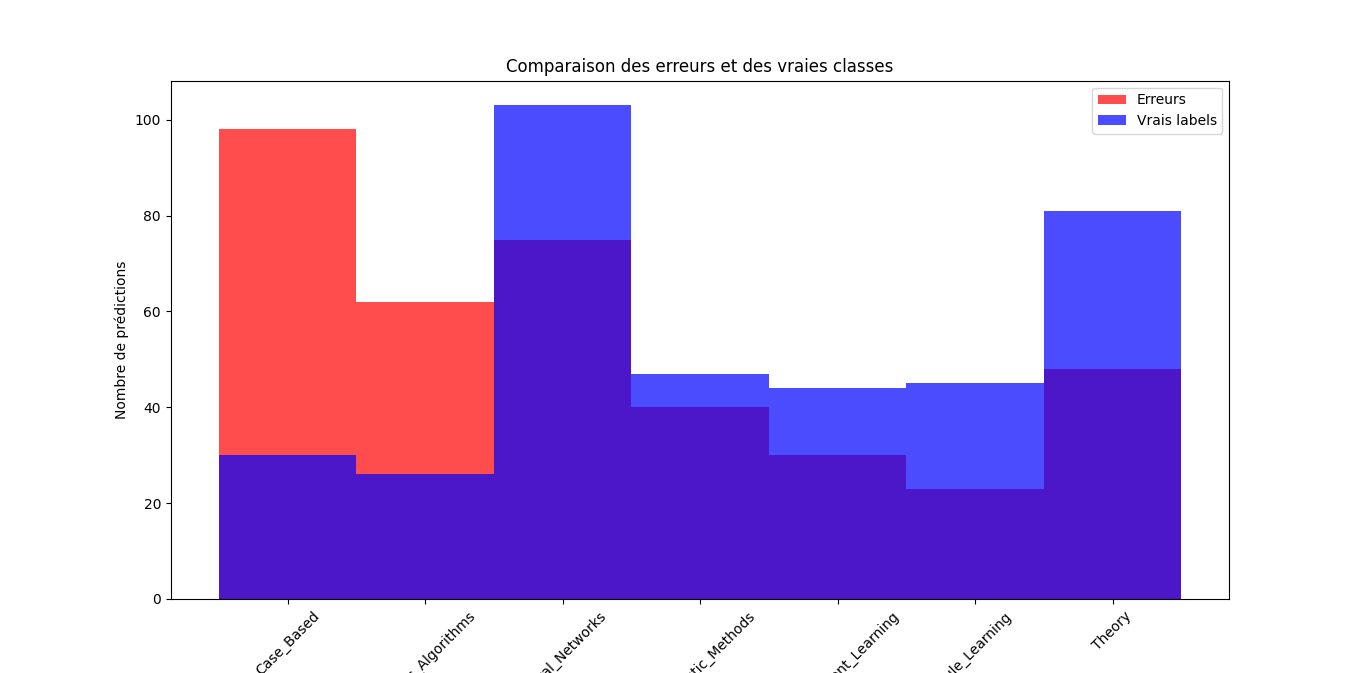
**Interprétation globale des Résultats**

Les performances varient fortement selon les classes :

* Les classes bien performantes, telles que "Neural\_Networks" et "Genetic\_Algorithms", indiquent que les caractéristiques qui les définissent dans le graphe sont bien séparables, permettant au modèle de les identifier facilement.
* Les classes ayant des performances plus faibles, comme "Reinforcement\_Learning" et "Rule\_Learning", peuvent refléter des insuffisances dans les données (classes sous-représentées ou partageant des similarités avec d'autres classes), ou des limites dans la capacité du modèle à généraliser pour ces classes spécifiques.

**b. Histogramme des erreurs fréquentes**

***Résultats* :**



***Interprétation de l'histogramme des erreurs fréquentes***

Cet histogramme compare les erreurs de prédiction (en rouge) aux vrais labels (en bleu) pour chaque classe, ce qui nous permet d'identifier les classes où le modèle a le plus de difficultés et celles où il réussit mieux.

* **Observations Globales**
  + Les classes "Case\_Based" et "Theory" montrent un nombre relativement élevé d'erreurs par rapport aux autres classes, indiquant que le modèle a du mal à prédire ces catégories correctement.
  + En revanche, la classe "Neural\_Networks" présente un nombre beaucoup plus faible d'erreurs comparé au nombre de vraies prédictions, ce qui confirme que c'est une des classes où le modèle performe le mieux.

* **Analyse Par Classe**
* **Case\_Based :**
  + Cette classe montre un grand nombre d'erreurs (en rouge), ce qui peut être dû à une similarité avec d'autres classes, comme "Rule\_Learning" ou "Reinforcement\_Learning". Cela pourrait indiquer un manque de caractéristiques discriminantes dans les données ou un déséquilibre dans la représentation des données.
* **Genetic\_Algorithms :**
  + Le nombre de prédictions correctes (en bleu) est relativement élevé, mais il existe encore quelques erreurs. Cela montre que le modèle distingue cette classe de manière acceptable, mais il reste une marge d'amélioration.
* **Neural\_Networks :**
  + Cette classe a un très faible nombre d'erreurs comparé aux prédictions correctes. Cela confirme que les caractéristiques des données pour cette classe sont bien distinctes, permettant au modèle de faire de bonnes prédictions.
* **Probabilistic\_Methods :**
  + Cette classe est également bien prédite, mais des erreurs restent visibles. Cela peut indiquer des similitudes avec d'autres classes comme "Reinforcement\_Learning".
* **Reinforcement\_Learning et Rule\_Learning :**
  + Ces classes montrent des proportions importantes d'erreurs par rapport aux vrais labels, suggérant des difficultés pour le modèle à différencier ces catégories.
* **Theory :**
  + Cette classe présente un déséquilibre notable entre les vraies prédictions et les erreurs. Les nombreuses erreurs pourraient être dues à une confusion avec des classes partageant des caractéristiques similaires (comme "Neural\_Networks").

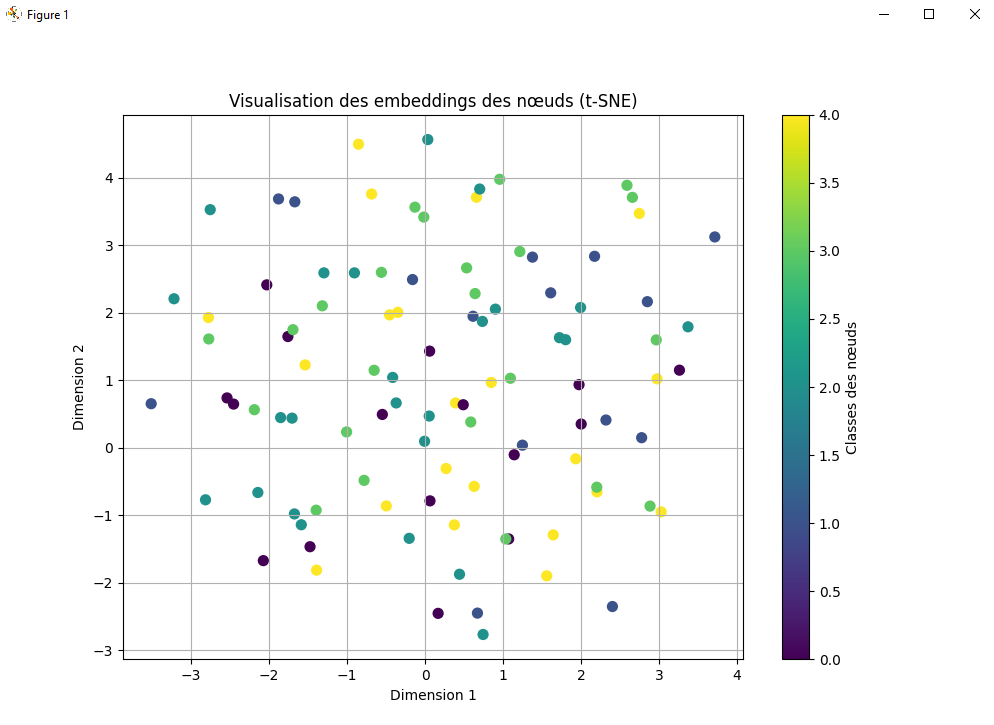
***Interprétation des Résultats***

L'histogramme met en évidence les points suivants :

* Certaines classes, comme "Neural\_Networks", sont bien distinguées par le modèle, grâce à des caractéristiques bien définies dans les données.
* D'autres classes, comme "Case\_Based", "Rule\_Learning", et "Theory", montrent une proportion élevée d'erreurs, indiquant des défis pour le modèle. Cela pourrait être dû à des chevauchements entre les caractéristiques de ces classes, des déséquilibres dans les données, ou des limites dans l'architecture actuelle.

**c. Visualisation des Embeddings avec t-SNE**

**Résultats :**



***Interprétation :***

**Description du Graphique**

Le graphique illustre une projection en 2D des embeddings des nœuds, générés par le modèle GNN, en utilisant la méthode de réduction de dimensionnalité t-SNE. Chaque point représente un nœud, et sa couleur correspond à la classe à laquelle il appartient (selon la légende à droite).

**Analyse**

* Certains clusters bien définis, comme ceux des classes jaunes et vertes, montrent que le modèle a appris à bien séparer ces classes.
* Toutefois, des chevauchements importants entre certaines classes, notamment entre les classes violettes et bleu clair, indiquent des difficultés de différenciation.
* Ces chevauchements reflètent probablement des similitudes dans les caractéristiques des nœuds ou un déséquilibre des données.

**Conclusion**

La visualisation t-SNE montre que le modèle capture efficacement les relations pour certaines classes, mais qu’il reste des améliorations à apporter pour séparer les classes qui se chevauchent.

1. **Optimisation des Hyperparamètres pour un Modèle de Réseaux de Neurones Graphiques (GNN)**

**2.1 Résumé**

Dans ce chapitre, nous nous intéressons à l’**optimisation des hyperparamètres** d’un modèle de **Réseaux de Neurones Graphiques (GNN)** afin d’améliorer ses **performances** et sa **précision** sur un ensemble de données. L’objectif est de déterminer la configuration optimale du modèle en testant plusieurs combinaisons de paramètres et en évaluant leur impact sur les résultats obtenus.

Nous adopterons une approche expérimentale basée sur des **essais aléatoires**, ce qui permettra d’explorer un large éventail de configurations possibles. Chaque combinaison d’hyperparamètres sera utilisée pour entraîner et évaluer un modèle GNN, et sa performance sera mesurée en termes de **précision sur les données de test**. Ce processus itératif nous aidera à identifier la configuration qui maximise les performances tout en maintenant une **généralisation robuste** sur des données non vues.

L’optimisation des hyperparamètres est une étape essentielle dans le développement de modèles d’apprentissage machine, car elle permet non seulement d’**améliorer la précision** mais aussi d’**éviter le sur-apprentissage** et de garantir une meilleure **stabilité du modèle**. En testant différentes combinaisons, nous pourrons mieux comprendre l’impact des paramètres choisis sur les résultats finaux et ainsi obtenir un modèle plus **efficace et adapté** à la tâche spécifique de classification sur des graphes.

Ce processus nous guidera vers une meilleure configuration qui pourra être utilisée pour des applications futures nécessitant des modèles fiables et performants dans des contextes impliquant des structures de données complexes comme les graphes.

**2.2 Paramètres étudiés dans l’optimisation**

**. Unités cachées – (hidden\_units)**

* **Description :** Ce paramètre définit le **nombre de neurones** dans les couches cachées du modèle.
* **Rôle :** Il contrôle la **capacité d’apprentissage** et la **complexité** du modèle. Un plus grand nombre de neurones permet de capturer des relations complexes, tandis qu’un nombre réduit peut limiter cette capacité.

**2. Taux de dropout – (dropout\_rate)**

* **Description :** Ce paramètre représente la **probabilité d’élimination** de certaines connexions pendant l’entraînement du modèle.
* **Rôle :** Il sert à **éviter le sur-apprentissage** en rendant le modèle plus robuste face aux variations dans les données.

**3. Type d’agrégation – (aggregation\_type)**

* **Description :** Ce paramètre définit la méthode utilisée pour **agréger les informations** provenant des nœuds voisins dans le graphe.
* **Rôle :** Il détermine **comment les informations locales** des voisins sont combinées pour représenter chaque nœud. Différentes méthodes (somme, moyenne, maximum) peuvent mieux capturer des structures spécifiques dans les données.

**4. Type de combinaison – (combination\_type)**

* **Description :** Ce paramètre contrôle la manière dont les informations des nœuds sont **combinées** après l’agrégation.
* **Rôle :** Il influence **l’intégration des informations** en choisissant des méthodes comme la concaténation, l’addition ou des mécanismes récurrents pour capturer des relations complexes.

**5. Taux d’apprentissage – (learning\_rate)**

* **Description :** Ce paramètre règle la **vitesse de mise à jour** des poids pendant l’entraînement.
* **Rôle :** Il détermine la **rapidité d’apprentissage** du modèle. Un taux trop élevé peut rendre l’apprentissage instable, tandis qu’un taux trop faible peut ralentir la convergence.

Ces paramètres sont essentiels pour **configurer et ajuster** le modèle de manière à optimiser ses performances sur des tâches spécifiques de classification sur des graphes.

**2.3 Implémentation**

**1. Résumé**

Dans cette étude, nous proposons une méthode d’optimisation basée sur une **recherche aléatoire** pour ajuster les hyperparamètres d’un **Réseau de Neurones Graphiques (GNN)**, appliqué à une tâche de **classification de nœuds**. Le modèle utilisé, nommé **cora\_torch**, est spécialement conçu pour traiter des données structurées sous forme de graphes.

L’objectif est d’explorer plusieurs combinaisons d’hyperparamètres afin d’identifier celle offrant la **meilleure précision** sur un ensemble de données de test. L’approche adoptée garantit une exploration systématique des configurations possibles tout en minimisant les coûts computationnels associés.

**2. Méthodologie**

La méthode repose sur l’évaluation itérative de différentes configurations d’hyperparamètres en utilisant un modèle GNN pré-défini. Le processus est structuré en plusieurs étapes :

**2.1. Définition des plages d’hyperparamètres**  
Des plages de valeurs discrètes sont spécifiées pour chaque hyperparamètre étudié. Ces plages couvrent une diversité de configurations, permettant une exploration approfondie. Les hyperparamètres sélectionnés sont :

* **Unités cachées (hidden\_units)** : Définit la taille des couches cachées pour réguler la complexité du modèle.
* **Taux de dropout (dropout\_rate)** : Introduit une régularisation pour éviter le sur-apprentissage.
* **Type d’agrégation (aggregation\_type)** : Spécifie la méthode utilisée pour agréger les informations des nœuds voisins.
* **Type de combinaison (combination\_type)** : Contrôle la manière dont les informations agrégées sont combinées après traitement.
* **Taux d’apprentissage (learning\_rate)** : Détermine la vitesse de mise à jour des poids pendant l’entraînement.

**2.2. Génération et évaluation des configurations**  
La fonction effectue une **recherche aléatoire** sur 20 combinaisons d’hyperparamètres. Chaque configuration est utilisée pour entraîner un modèle GNN sur les données d’entraînement, suivie d’une évaluation sur un ensemble de test.

**2.3. Sélection de la meilleure configuration**  
Les résultats obtenus sont stockés et analysés afin d’identifier la configuration optimisant la **précision** sur les données de test.

**3. Explication** **du Code**

#### ****3.1. Importations :****

Le code commence par l'importation des bibliothèques nécessaires :

**import** random

**from** tensorflow.keras.optimizers **import** Adam

from cora\_torch import GNNNodeClassifier, run\_experiment, graph\_info, num\_classes, x\_train, y\_train, x\_test, y\_test

* **random :**  
  Utilisée pour générer des choix aléatoires parmi les plages d’hyperparamètres spécifiées dans le code. Cela permet d’explorer diverses combinaisons d’hyperparamètres.
* **Adam (de TensorFlow) :**  
  L’optimiseur Adam est employé pour ajuster les poids du modèle durant l’entraînement. Il est apprécié pour sa **convergence rapide** et sa **robustesse** face aux variations des gradients.
* **cora\_torch :**  
  Ce module personnalisé contient les définitions et données associées au modèle GNN utilisé. Il fournit :
  + **GNNNodeClassifier** : La classe définissant l’architecture du modèle GNN.
  + **run\_experiment** : Une fonction permettant d’entraîner et de valider le modèle.
  + **graph\_info** : Les informations structurelles du graphe utilisé (par exemple, les connexions entre nœuds).
  + **num\_classes** : Le nombre de classes à prédire dans la tâche de classification.
  + **x\_train, y\_train** : Les données d’entraînement (entrées et étiquettes).
  + **x\_test, y\_test** : Les données de test utilisées pour évaluer la performance.

#### ****2. Méthode principale :**** optimize\_hyperparameters()

Cette fonction optimise les hyperparamètres du modèle GNN à travers des essais multiples.

**def optimize\_hyperparameters():**

"""

Fonction pour optimiser les hyperparamètres du modèle GNN.

Retourne les meilleurs hyperparamètres et leur précision.

"""

**Description générale :**  
Elle explore plusieurs combinaisons d’hyperparamètres en utilisant une **recherche aléatoire**, entraîne le modèle avec chaque configuration, évalue ses performances, et retourne la meilleure configuration.

##### **Étapes clés de la fonction :**

**1. Définition des plages d’hyperparamètres :**

**hidden\_units\_options** = [[16, 16], [32, 32], [64, 64]]

**dropout\_rate\_options** = [0.1, 0.2, 0.3, 0.4, 0.5]

**aggregation\_type\_options** = ["sum", "mean", "max"]

**combination\_type\_options** = ["concat", "add", "gru"]

**learning\_rate\_options** = [0.001, 0.005, 0.01]

* Définit des plages de valeurs possibles pour chaque hyperparamètre étudié.
* Ces plages permettent de tester des configurations variées afin d’optimiser les performances du modèle.

**2. Stockage des résultats :**

**results** = []

* Initialise une liste pour enregistrer les résultats obtenus (précision et hyperparamètres testés).

**3. Méthode interne : train\_and\_evaluate(params)**

**def train\_and\_evaluate(params):**

* **Rôle :** Cette méthode entraîne et évalue un modèle GNN en utilisant un ensemble d’hyperparamètres spécifiques passés en argument.

**Étapes :**

1. **Création du modèle avec les hyperparamètres :**

**gnn\_model = GNNNodeClassifier**(

graph\_info=graph\_info,

num\_classes=num\_classes,

hidden\_units=hidden\_units,

dropout\_rate=dropout\_rate,

aggregation\_type=aggregation\_type,

combination\_type=combination\_type,

normalize=True,

)

Le modèle est initialisé avec les hyperparamètres sélectionnés.

1. **Compilation du modèle :**

**optimizer** = Adam(learning\_rate=learning\_rate)

**gnn\_model.compile**(

optimizer=optimizer,

loss="sparse\_categorical\_crossentropy",

metrics=["accuracy"],

)

* + Définit l’optimiseur (**Adam**) et la fonction de perte (**cross-entropy sparse**) adaptée à la classification multi-classes.

1. **Entraînement et évaluation :**

**history** = run\_experiment(gnn\_model, x\_train, y\_train)

\_, test\_accuracy = gnn\_model.evaluate(x=x\_test, y=y\_test, verbose=0)

* + Entraîne le modèle sur les données d’entraînement et mesure la **précision** sur les données de test.

1. **Gestion des erreurs :**

except ValueError as e:

print(f"Erreur d'incompatibilité de forme : {e}")

return 0, params

En cas d’incompatibilité des dimensions dans les données, l’erreur est capturée et une précision de 0 est retournée.

**4. Boucle d’essais aléatoires :**

**num\_trials** = 20

for trial in range(num\_trials):

* Effectue **20 essais** avec des combinaisons aléatoires d’hyperparamètres.

**Sélection d’une combinaison aléatoire :**

**params** = (

random.choice(hidden\_units\_options),

random.choice(dropout\_rate\_options),

random.choice(aggregation\_type\_options),

random.choice(combination\_type\_options),

random.choice(learning\_rate\_options),

)

**Entraînement et stockage des résultats :**

accuracy, params\_used = train\_and\_evaluate(params)

results.append((accuracy, params\_used))

Chaque combinaison est testée et son résultat est sauvegardé.

**5. Sélection des meilleurs hyperparamètres :**

best\_result = max(results, key=lambda x: x[0])

* Identifie la combinaison ayant obtenu la **meilleure précision**.

**6. Retour du résultat final :**

print("\nMeilleure précision : {:.4f}".format(best\_result[0]))

print("Meilleurs hyperparamètres :", best\_result[1])

Affiche les **meilleurs hyperparamètres** trouvés ainsi que la précision associée.

Lancement de 20 essais pour optimiser les hyperparamètres...

Epoch 1/300

5/5 **━━━━━━━━━━━━━━━━━━━━** 15s 354ms/step - acc: 0.2490 - loss: 1.9176 - val\_acc: 0.2277 - val\_loss: 1.8699

Epoch 2/300

5/5 **━━━━━━━━━━━━━━━━━━━━** 0s 61ms/step - acc: 0.3028 - loss: 1.7932 - val\_acc: 0.3267 - val\_loss: 1.7402

Epoch 3/300

5/5 **━━━━━━━━━━━━━━━━━━━━** 0s 71ms/step - acc: 0.4395 - loss: 1.5929 - val\_acc: 0.5396 - val\_loss: 1.4253

Epoch 4/300

5/5 **━━━━━━━━━━━━━━━━━━━━** 0s 76ms/step - acc: 0.6046 - loss: 1.2275 - val\_acc: 0.4851 - val\_loss: 1.3864

Epoch 5/300

...

...

Epoch 82/300

5/5 **━━━━━━━━━━━━━━━━━━━━** 0s 81ms/step - acc: 1.0000 - loss: 1.0925e-04 - val\_acc: 0.6733 - val\_loss: 3.0229

Essai 1/20 | Test accuracy: 0.6628 | Params: ([16, 16], 0.1, 'mean', 'concat', 0.001)

...

...

Epoch 55/300

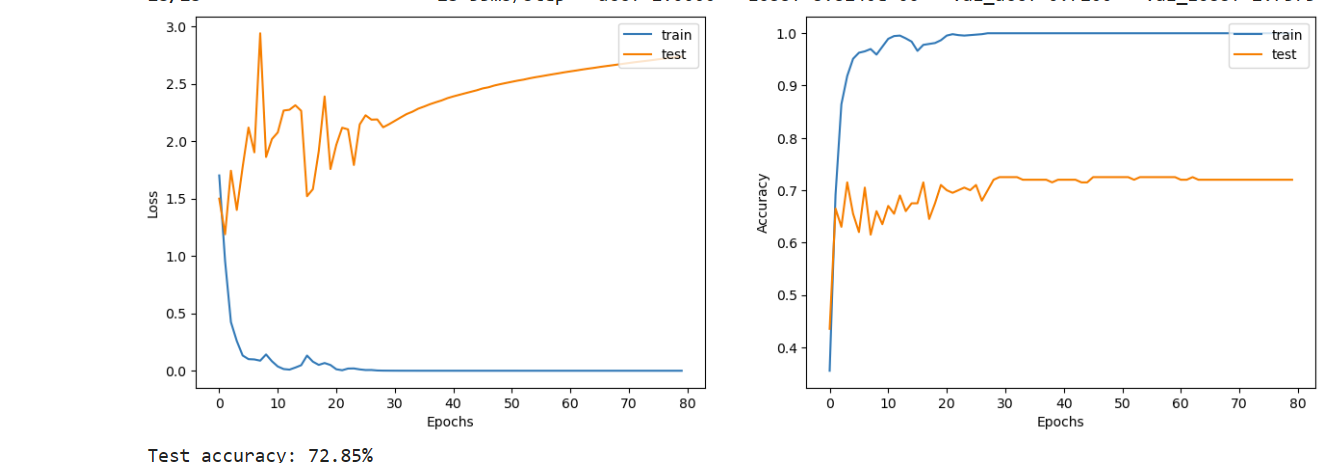
5/5 **━━━━━━━━━━━━━━━━━━━━** 1s 201ms/step - acc: 1.0000 - loss: 2.7129e-05 - val\_acc: 0.6980 - val\_loss: 3.0301

Essai 20/20 | Test accuracy: 0.7280 | Params: ([64, 64], 0.1, 'sum', 'concat', 0.001)

Meilleure précision : 0.7280

Meilleurs hyperparamètres : ([64, 64], 0.1, 'sum', 'concat', 0.001)

**4. test des Hyperparamètres**



**Optimisation par Ajout de Couches GAT et Augmentation de la Profondeur du Modèle :**

**Objectif :**  
L’objectif principal de cette optimisation est d’améliorer significativement les performances du modèle en augmentant sa capacité à apprendre et à représenter des relations complexes dans les graphes. Cela inclut une meilleure précision dans les classifications ou prédictions en exploitant pleinement la structure et les informations des graphes. En augmentant la profondeur du modèle et en intégrant des mécanismes avancés comme les Graph Attention Layers (GAT), le modèle est conçu pour capturer des dépendances complexes et hiérarchiques entre les nœuds, ce qui est essentiel dans des applications nécessitant une compréhension fine des structures de graphe.

**Justification :**

1. **Les Graph Attention Layers (GAT) :**
   * Ces couches introduisent un mécanisme d’attention qui permet au modèle de pondérer différemment les contributions des voisins d’un nœud. Cela signifie que les connexions les plus pertinentes ou influentes dans le graphe reçoivent plus d’importance lors de l’apprentissage.
   * Contrairement aux approches classiques, le GAT s’adapte dynamiquement à la structure locale du graphe, ce qui améliore la précision dans les graphes où les connexions entre les nœuds sont hétérogènes ou ont des importances variables.
   * En permettant cette pondération fine, le modèle devient plus robuste face aux bruits dans les données, car il peut ignorer les connexions moins significatives.
2. **Profondeur accrue du modèle :**
   * L’ajout de couches supplémentaires, que ce soit dans les GAT ou les Graph Convolutional Layers (GCL), permet d’élargir le champ de réception du modèle. Ainsi, les informations provenant de nœuds plus éloignés dans le graphe peuvent être intégrées, offrant une meilleure vue d’ensemble des relations globales et locales.
   * Les couches plus profondes permettent également au modèle de capturer des motifs structurels complexes et des dépendances de plus haut niveau, ce qui est souvent critique dans des applications avancées comme l’analyse de réseaux sociaux ou les systèmes de recommandation.
3. **Techniques complémentaires pour stabiliser l’optimisation :**
   * **Dropout :** Utilisé pour éviter le surapprentissage en masquant aléatoirement certaines connexions pendant l’entraînement. Cela force le modèle à être plus général et à ne pas dépendre d’un sous-ensemble spécifique de relations dans le graphe.
   * **Normalisation :** Elle garantit que les informations transmises entre les couches restent stables, même lorsque le modèle devient plus profond. Cela aide à éviter les problèmes liés à la disparition ou à l’explosion des gradients.

**Modifications apportées** :

 **Ajout des Graph Attention Layers (GAT)** :

* Nous avons introduit quatre couches de Graph Attention (GAT) pour améliorer la capacité du modèle à capturer des relations complexes entre les nœuds d'un graphe. Les GATs attribuent des poids d'attention aux différentes connexions, ce qui permet de mieux distinguer les relations pertinentes.

 **Augmentation de la profondeur du réseau** :

* Nous avons ajouté des couches supplémentaires, en particulier quatre couches de GAT et quatre couches de Graph Convolutional Layers (GCL), afin d'augmenter la capacité d'apprentissage du modèle. Cette profondeur accrue permet au modèle de capturer des représentations plus abstraites et complexes des nœuds, tout en renforçant les interactions à longue portée entre eux.

 **Amélioration des régularisations et du traitement des poids** :

* Nous avons intégré des techniques de régularisation telles que le dropout et la normalisation de lots pour éviter le surapprentissage et améliorer la généralisation du modèle. En outre, la normalisation des poids des arêtes a été mise en place pour garantir que l'importance des différentes connexions soit équilibrée.

class GraphAttentionLayer(layers.Layer):

def \_\_init\_\_(self, hidden\_units, dropout\_rate=0.2, use\_bias=True, \*args, \*\*kwargs):

super().\_\_init\_\_(\*args, \*\*kwargs)

self.hidden\_units = hidden\_units

self.dropout\_rate = dropout\_rate

# Poids pour la transformation linéaire.

self.weight = self.add\_weight(

shape=(hidden\_units, hidden\_units), initializer="glorot\_uniform", trainable=True

)

# Poids pour le calcul de l'attention.

self.attention\_weight = self.add\_weight(

shape=(2 \* hidden\_units, 1), initializer="glorot\_uniform", trainable=True

)

self.dropout = layers.Dropout(dropout\_rate)

self.use\_bias = use\_bias

if self.use\_bias:

self.bias = self.add\_weight(shape=(hidden\_units,), initializer="zeros", trainable=True)

def call(self, inputs):

node\_representations, edges, edge\_weights = inputs

# Transformation linéaire.

transformed\_nodes = tf.matmul(node\_representations, self.weight)

# Préparer les représentations pour l'attention.

source\_nodes = tf.gather(transformed\_nodes, edges[0]) # Noeuds source

target\_nodes = tf.gather(transformed\_nodes, edges[1]) # Noeuds voisins

# Calcul des scores d'attention.

concatenated\_nodes = tf.concat([source\_nodes, target\_nodes], axis=-1)

attention\_scores = tf.nn.leaky\_relu(

tf.squeeze(tf.matmul(concatenated\_nodes, self.attention\_weight), axis=-1)

)

# Normalisation des scores avec softmax.

attention\_weights = tf.nn.softmax(attention\_scores, axis=0)

# Appliquer l'attention aux messages.

weighted\_messages = tf.expand\_dims(attention\_weights, axis=-1) \* target\_nodes

aggregated\_messages = tf.math.unsorted\_segment\_sum(

weighted\_messages, edges[0], num\_segments=node\_representations.shape[0]

)

if self.use\_bias:

aggregated\_messages += self.bias

# Optionnel : Appliquer dropout et normalisation.

return self.dropout(aggregated\_messages)

class GNNNodeClassifier(tf.keras.Model):

def \_init\_(

self,

graph\_info,

num\_classes,

hidden\_units,

aggregation\_type="sum",

combination\_type="concat",

dropout\_rate=0.2,

normalize=True,

\*args,

\*\*kwargs,

):

super().\_init\_(\*args, \*\*kwargs)

# Unpack graph\_info to three elements: node\_features, edges, and edge\_weight.

node\_features, edges, edge\_weights = graph\_info

self.node\_features = node\_features

self.edges = edges

self.edge\_weights = edge\_weights

# Set edge\_weights to ones if not provided.

if self.edge\_weights is None:

self.edge\_weights = tf.ones(shape=edges.shape[1])

# Scale edge\_weights to sum to 1.

self.edge\_weights = self.edge\_weights / tf.math.reduce\_sum(self.edge\_weights)

# Create a process layer.

self.preprocess = create\_ffn(hidden\_units, dropout\_rate, name="preprocess")

# Create 4 Graph Attention Layers (GAT)

self.gat1 = GraphAttentionLayer(hidden\_units[0], dropout\_rate, name="gat1")

self.gat2 = GraphAttentionLayer(hidden\_units[0], dropout\_rate, name="gat2")

self.gat3 = GraphAttentionLayer(hidden\_units[0], dropout\_rate, name="gat3")

self.gat4 = GraphAttentionLayer(hidden\_units[0], dropout\_rate, name="gat4")

# Create 4 Graph Convolutional Layers (GCL)

self.conv1 = GraphConvLayer(

hidden\_units,

dropout\_rate,

aggregation\_type,

combination\_type,

normalize,

name="graph\_conv1",

)

self.conv2 = GraphConvLayer(

hidden\_units,

dropout\_rate,

aggregation\_type,

combination\_type,

normalize,

name="graph\_conv2",

)

self.conv3 = GraphConvLayer(

hidden\_units,

dropout\_rate,

aggregation\_type,

combination\_type,

normalize,

name="graph\_conv3",

)

self.conv4 = GraphConvLayer(

hidden\_units,

dropout\_rate,

aggregation\_type,

combination\_type,

normalize,

name="graph\_conv4",

)

# Create a postprocess layer.

self.postprocess = create\_ffn(hidden\_units, dropout\_rate, name="postprocess")

# Create a compute logits layer.

self.compute\_logits = layers.Dense(units=num\_classes, name="logits")

def call(self, input\_node\_indices):

# Preprocess the node\_features to produce node representations.

x = self.preprocess(self.node\_features)

# Apply the first Graph Attention Layer (GAT).

x1 = self.gat1((x, self.edges, self.edge\_weights))

x = x1 + x # Skip connection

# Apply the second Graph Attention Layer (GAT).

x2 = self.gat2((x, self.edges, self.edge\_weights))

x = x2 + x # Skip connection

# Apply the third Graph Attention Layer (GAT).

x3 = self.gat3((x, self.edges, self.edge\_weights))

x = x3 + x # Skip connection

# Apply the fourth Graph Attention Layer (GAT).

x4 = self.gat4((x, self.edges, self.edge\_weights))

x = x4 + x # Skip connection

# Apply the first Graph Convolutional Layer (GCL).

x5 = self.conv1((x, self.edges, self.edge\_weights))

x = x5 + x # Skip connection

# Apply the second Graph Convolutional Layer (GCL).

x6 = self.conv2((x, self.edges, self.edge\_weights))

x = x6 + x # Skip connection

# Apply the third Graph Convolutional Layer (GCL).

x7 = self.conv3((x, self.edges, self.edge\_weights))

x = x7 + x # Skip connection

# Apply the fourth Graph Convolutional Layer (GCL).

x8 = self.conv4((x, self.edges, self.edge\_weights))

x = x8 + x # Skip connection

# Postprocess node embeddings.

x = self.postprocess(x)

# Fetch node embeddings for the input node\_indices.

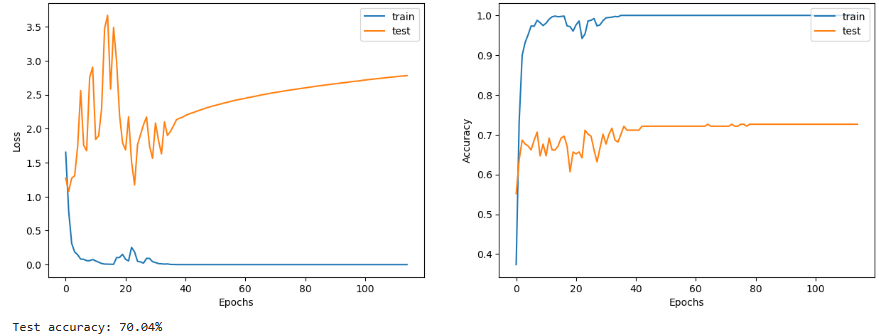
node\_embeddings = tf.gather(x, input\_node\_indices)

# Compute logits

return self.compute\_logits(node\_embeddings)

**Résultats obtenus** :

Après l'ajout des Graph Attention Layers (GAT) et l'augmentation de la profondeur du modèle, nous avons observé une amélioration notable de la performance. L'accuracy est passée de 68 % à 70 %. Bien que l'amélioration ne soit pas aussi importante que prévu, elle témoigne néanmoins de l'efficacité des modifications apportées pour mieux capturer les relations complexes dans le graphe. Cette progression suggère que l'augmentation de la capacité du modèle, notamment avec l'ajout de couches d'attention et la régularisation, a permis de mieux généraliser aux données tout en réduisant les erreurs de prédiction.

****

1. **Conclusion**