



Machine Learning for Data Science

Rapport du Projet Apprentissage Profond

Optimisation de la Prédiction des Données MNIST avec Seulement 100 Étiquettes Utilisant un Algorithme Semi-Supervisé à Base de Pseudo-Labels

Réalisé par :

- BELKAID Meryem (AMSD)
- BOANANI Hafsa (AMSD)
- OUMGHAR Abir (AMSD)
- HAKIMI Sanae (MLSD)

Année universitaire 2023-2024

Table des matières

Introduction	3
1. Contexte du projet.....	3
1.1. MNIST Dataset.....	3
1.2. Apprentissage Semi-Supervisé	4
1.3. Modèles NN et CNN.....	5
1.4. Objectifs de travail.....	6
2. Méthodologie proposée par l'article.....	7
2.1. Semi Supervised Self Training	7
2.2. Méthodes des pseudo-étiquettes pour les réseaux neuronaux profonds	8
2.2.1. Réseaux neuronaux profonds	8
2.2.2. Denoising Auto-Encoder	9
2.2.3. Dropout	9
2.2.4. Entraînement avec les pseudo-étiquettes comme régularisation de l'entropie	10
3. Implémentation	12
3.1. Architecture de l'algorithme	12
3.2. Entraînement des modèles sur MNIST digits.....	13
4. Résultats	14
Conclusion.....	16
Références	17
Annexes	18

Introduction

Au fil des récentes années, les réseaux de neurones profonds ont révélé leur capacité exceptionnelle à exceller dans diverses tâches d'apprentissage supervisé, notamment la complexe classification d'images. Néanmoins, il est crucial de noter que ces performances éblouissantes sont souvent atteintes au prix d'ensembles de données considérables, nécessitant des ressources substantielles en termes de temps et d'efforts humains. Cette contrainte soulève des défis pratiques significatifs, limitant ainsi l'applicabilité directe des méthodes de Deep Learning dans des contextes du monde réel.

C'est dans ce contexte que la recherche a progressivement orienté son attention vers l'apprentissage semi-supervisé et ses applications potentielles dans le domaine des réseaux de neurones. L'objectif sous-jacent à cette démarche novatrice est de réduire la dépendance envers des ensembles de données étiquetés massifs. Cette réduction pourrait être obtenue soit par le biais du développement de nouvelles approches, soit en adaptant des cadres d'apprentissage semi-supervisé existants pour répondre aux complexités inhérentes à l'apprentissage profond.

C'est dans cette optique que s'inscrit notre projet de recherche, se penchant spécifiquement sur le défi de prédire les données MNIST tout en se limitant à l'utilisation de seulement 100 labels pendant la phase d'apprentissage. Notre démarche méthodologique débute par la mise en place d'une ligne de base, suivie d'une synthèse concise des méthodes d'apprentissage semi-supervisé les plus prometteuses que propose la littérature. Enfin, nous détaillons de manière approfondie la méthode spécifique que nous avons judicieusement sélectionnée pour résoudre ce problème particulier. Cette approche s'avère être une contribution significative, offrant un éclairage nouveau sur l'apprentissage semi-supervisé appliqué à la prédiction des données MNIST, le tout dans des conditions de contrainte en termes de nombre de labels, fixé à seulement 100.

1. Contexte du projet

1.1. MNIST Dataset

Le jeu de données MNIST est une collection de 70 000 images en noir et blanc de chiffres manuscrits de 28x28 pixels. Il est souvent utilisé pour le traitement d'images et le benchmarking des algorithmes d'apprentissage automatique.¹

Le jeu de données MNIST est souvent utilisé pour des tâches de classification d'images, en particulier pour la classification de chiffres manuscrits. Il est également comparé au jeu de données Fashion MNIST, qui est une version plus difficile du MNIST et qui contient des images de vêtements au lieu de chiffres manuscrits. Dans notre projet, le jeu de données en question englobe un ensemble exhaustif de 70 000 images, réparties entre 60 000 destinées à l'entraînement du modèle et 10 000 consacrées à sa validation, avec une classe par chiffre, soit un total de 10 classes.

¹ Jayant Verma, « MNIST Dataset in Python - Basic Importing and Plotting », Récupéré de : <https://www.digitalocean.com/community/tutorials/mnist-dataset-in-python>

Cette division stratégique vise à garantir une robustesse et une généralisation optimales du modèle, en lui permettant d'apprendre à partir d'une variété suffisante d'exemples tout en conservant un ensemble distinct pour l'évaluation et la validation des performances.²

Pour importer et visualiser le jeu de données MNIST en Python, on peut utiliser des bibliothèques telles que Matplotlib et des modules spécifiques de Tensor Flow et Keras. Ce jeu de données est largement utilisé dans des tutoriels et des études de cas pour l'apprentissage automatique et l'apprentissage profond.

Propriété	Valeur
Nombre total d'images	70 000
Taille de chaque image	28x28 pixels
Nombre d'images d'entraînement	60 000
Nombre d'images de validation	10 000
Echantillon d'images étiquetées	100
Nombre de classes	10 (chiffres de 0 à 9)
Images par classe	10
Objectif	Benchmarking, apprentissage automatique, reconnaissance de motifs
Comparaison	Souvent comparé à Fashion MNIST (images de vêtements)

1.2. Apprentissage Semi-Supervisé

L'apprentissage semi-supervisé est une méthode d'apprentissage automatique qui utilise à la fois des données étiquetées et non étiquetées pour entraîner un modèle. Cette approche se situe entre l'apprentissage supervisé, qui utilise uniquement des données étiquetées, et l'apprentissage non supervisé, qui utilise uniquement des données non étiquetées.

L'apprentissage semi-supervisé est utile lorsque le nombre de données étiquetées est limité, car il permet d'étendre l'apprentissage à partir de données non étiquetées, améliorant ainsi la qualité de l'apprentissage.

² Hugging Face. (s. d.). MNIST Dataset. Récupéré de <https://huggingface.co/datasets/mnist>

Par exemple, dans le domaine de la vision par ordinateur, les modèles d'apprentissage semi-supervisé peuvent être utilisés pour des tâches telles que la reconnaissance d'objets, la classification d'objets, ou la segmentation sémantique. Cette approche peut également être appliquée dans des domaines tels que les ventes et le marketing, où elle permet de gérer à la fois le manque et la surabondance de données, offrant ainsi des avantages pratiques³.

Les modèles de réseaux de neurones (NN) et de réseaux de neurones convolutifs (CNN) peuvent être utilisés dans des contextes d'apprentissage semi-supervisé. Par exemple, des méthodes semi-supervisées basées sur les CNN ont été proposées pour la reconnaissance des défauts de surface de l'acier, ainsi que pour la segmentation sémantique. De plus, des approches innovantes telles que le "Transformer-CNN Cohort" ont été développées pour la segmentation sémantique semi-supervisée, montrant ainsi l'applicabilité des modèles NN et CNN dans ce domaine⁴.

En résumé, l'apprentissage semi-supervisé offre une solution intermédiaire entre l'apprentissage supervisé et non supervisé, permettant d'améliorer la qualité de l'apprentissage en utilisant à la fois des données étiquetées et non étiquetées. Cette approche trouve des applications dans divers domaines, y compris la vision par ordinateur, les ventes et le marketing, et peut être mise en œuvre à l'aide de modèles de réseaux de neurones et de réseaux de neurones convolutifs⁵.

1.3. Modèles NN et CNN

Dans le domaine de l'apprentissage profond et de l'apprentissage semi-supervisé, les modèles de réseaux de neurones (**NN**) et les modèles de réseaux de neurones convolutionnels (**CNN**) sont deux approches distinctes avec leurs propres caractéristiques⁶.

En termes de structure, les NN sont des modèles de réseaux de neurones généraux qui peuvent être utilisés pour diverses tâches d'apprentissage automatique. Ils sont composés de couches de neurones interconnectés, où chaque neurone est relié à tous les neurones de la couche précédente.

En revanche, les CNN sont spécifiquement conçus pour le traitement d'images et de données 2D. Ils exploitent la structure spatiale des données en utilisant des couches spéciales, telles que les couches de convolution et de pooling.

³ Deval Shah, Abhishek Jha, « Self-Supervised Learning and Its Applications », Recupéré de : <https://neptune.ai/blog/self-supervised-learning>

⁴ Yiping Gao, « A semi-supervised convolutional neural network-based method for steel surface defect recognition », Récupéré de : https://www.researchgate.net/publication/336665610_A_semi-supervised_convolutional_neural_network-based_method_for_steele_surface_defect_recognition

⁵ « Apprentissage semi-supervisé », Récupéré de : https://fr.wikipedia.org/wiki/Apprentissage_semi-supervisé

⁶ Mayank Mishra, « Convolutional Neural Networks, Explained », Récupéré de : <https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939>

En ce qui concerne le traitement des données, les CNN sont particulièrement adaptés à l'analyse d'images et de données avec une structure spatiale, grâce à leur capacité à exploiter les motifs locaux à l'aide d'opérations de convolution. Les NN, quant à eux, peuvent être utilisés pour une variété de tâches, telles que la classification, la régression, la génération de texte, etc. L'apprentissage semi-supervisé est une approche dans laquelle un modèle utilise à la fois des données étiquetées et non étiquetées pour l'entraînement. Les NN et les CNN peuvent tous deux être utilisés pour l'apprentissage semi-supervisé. Cependant, les CNN sont généralement considérés comme plus performants dans le domaine de l'analyse d'images en raison de leur capacité à extraire automatiquement des caractéristiques pertinentes à partir des données non étiquetées.

En résumé, les réseaux de neurones (NN) se distinguent par leur caractère général, convenant à diverses tâches d'apprentissage automatique, tandis que les réseaux de neurones convolutifs (CNN) sont spécifiquement élaborés pour l'analyse d'images et de données en deux dimensions. Bien que les deux approches puissent être appliquées dans le cadre de l'apprentissage semi-supervisé, les CNN sont fréquemment privilégiés pour les missions d'analyse d'images, du fait de leur capacité à exploiter de manière efficace les structures spatiales inhérentes aux données visuelles.

1.4. Objectifs de travail

Les objectifs de cette étude de recherche sont les suivants :

- L'objectif principal de cette recherche est d'effectuer des prédictions sur les **60 000 images** du jeu de données MNIST en se basant uniquement sur un échantillon de **100 images** étiquetées. Pour atteindre cet objectif, deux approches différentes seront utilisées : un modèle de réseau de **neurones convolutionnels (CNN)** et un modèle de **réseau de neurones (NN)** pour l'apprentissage semi-supervisé.
- La première approche consiste à entraîner un modèle CNN en utilisant les 100 images étiquetées ainsi que les images non étiquetées du jeu de données MNIST. L'apprentissage semi-supervisé sera réalisé en utilisant des techniques sophistiquées telles que l'auto-encodage et la régularisation.
- La deuxième approche se fonde sur l'article intitulé "**Pseudo-Label: The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks**" [LINK](#). Cette approche exploite le pseudo-labeling pour attribuer des étiquettes aux images non étiquetées, qui seront ensuite incluses dans l'ensemble d'entraînement du modèle NN. Cette méthode tire parti des informations présentes dans les données non étiquetées afin d'améliorer les performances du modèle.

Une fois ces deux approches mises en œuvre, une comparaison rigoureuse des modèles CNN et NN sera réalisée en évaluant leurs taux de perte (loss) et de précision (accuracy). Ces mesures permettront de déterminer la performance relative des deux modèles dans la prédiction des étiquettes des images du jeu de données MNIST.

En conclusion, cette étude de recherche vise à réaliser des prédictions sur les 60 000 images du jeu de données MNIST en utilisant seulement 100 images étiquetées. Deux approches différentes, à savoir un modèle CNN et un modèle NN avec pseudo-labeling, seront utilisées pour atteindre cet objectif. Les performances de ces modèles seront comparées en termes de taux de perte et de précision, dans le but d'obtenir des informations précieuses sur leur efficacité respective.

2. Méthodologie proposée par l'article

L'article "**Pseudo-Label: The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks**" propose une méthode d'apprentissage Semi-Supervisé efficace pour les réseaux de neurones profonds. Cette méthode repose sur l'utilisation de données non étiquetées pour améliorer les performances des modèles de réseaux de neurones. Elle a été appliquée avec succès dans divers domaines, notamment la classification d'images, la reconnaissance des défauts de surface de l'acier et la reconnaissance d'activités humaines.

Les investigations ont dévoilé que l'intégration de cette approche, particulièrement conjuguée avec les réseaux de neurones convolutionnels (CNN), s'est traduite par des améliorations notables dans diverses tâches. Ces améliorations se manifestent de manière significative dans des domaines aussi variés que la classification d'images, la détection des défauts de surface de l'acier, et la reconnaissance des activités humaines. Pour illustrer, une étude comparative a été menée entre la méthode Pseudo-Label et d'autres stratégies d'apprentissage semi-supervisé dans le cadre de la classification d'images, mettant en évidence son efficacité supérieure par rapport aux alternatives considérées.

En résumé, Pseudo-Label est une méthode simple et efficace d'apprentissage semi-supervisé qui a été appliquée avec succès dans divers domaines, notamment la vision par ordinateur et la reconnaissance d'activités humaines, démontrant ainsi son potentiel pour améliorer les performances des modèles de réseaux de neurones profonds dans des scénarios où les données étiquetées sont limitées.

2.1. Semi Supervised Self Training

L'apprentissage semi-supervisé constitue une catégorie de méthodes d'apprentissage automatique qui tire parti à la fois de données étiquetées et non étiquetées. Situé entre l'apprentissage supervisé, qui dépend exclusivement de données étiquetées, et l'apprentissage non supervisé, qui utilise uniquement des données non étiquetées, ce modèle hybride a prouvé son efficacité en améliorant de manière significative la qualité de l'apprentissage.

L'intérêt de l'apprentissage semi-supervisé se renforce également en raison du processus souvent laborieux d'étiquetage des données, nécessitant fréquemment l'intervention d'un utilisateur humain. Cette tâche devient particulièrement ardue lorsque les jeux de données atteignent une ampleur considérable. Dans ces conditions, l'apprentissage semi-supervisé, qui se contente d'un nombre limité d'étiquettes, présente une solution pratique et pertinente.

Cette approche s'inscrit dans des contextes variés, avec des applications étendues, notamment dans les domaines des ventes et du marketing. Dans ces secteurs spécifiques, l'apprentissage semi-supervisé s'engage à résoudre les défis inhérents à la gestion des données, que ce soit face à leur rareté ou à leur surabondance. Les professionnels de l'analyse des données en vente et marketing se trouvent parfois dans une quête désespérée de données qui leur font cruellement défaut, tandis qu'à d'autres moments, ils se voient submergés par des données difficiles à exploiter. Ainsi, l'apprentissage semi-supervisé se profile comme une solution efficace à ces problématiques de gestion de données dans les domaines des ventes et du marketing.

En outre, l'apprentissage semi-supervisé constitue une ressource précieuse pour optimiser les performances des modèles de réseaux de neurones. Par exemple, l'approche d'auto-entraînement semi-supervisé, également appelée "Self-Training", exploite un modèle initial pré-entraîné pour anticiper les étiquettes des données non étiquetées. Les données accompagnées de ces étiquettes prédictives sont ensuite intégrées à l'ensemble d'entraînement supervisé, alimentant le modèle dans une phase d'entraînement étendue. Ce processus itératif se répète jusqu'à l'atteinte d'un critère d'arrêt prédéfini. Cette méthode se révèle particulièrement avantageuse pour améliorer les performances des modèles en tirant profit de données non étiquetées, une approche particulièrement pertinente lorsque l'acquisition de données étiquetées s'avère coûteuse ou difficile.

En conclusion, l'apprentissage semi-supervisé, en particulier les méthodes de pseudo-étiquetage et de self-training, offre des approches prometteuses pour améliorer les performances des modèles d'apprentissage automatique dans des scénarios où les données étiquetées sont limitées.

2.2. Méthodes des pseudo-étiquettes pour les réseaux neuronaux profonds

2.2.1. Réseaux neuronaux profonds

Au cœur de notre projet, nous nous plongeons dans l'exploration approfondie des réseaux neuronaux à plusieurs couches, un pilier essentiel de l'apprentissage automatique. Les participants s'immergent dans les subtilités des unités sigmoïdes, analysant de près l'impact de cette fonction d'activation spécifique sur la propagation de l'information au sein du réseau. De manière similaire, l'attention est dirigée vers les unités linéaires rectifiées, examinant de quelle manière cette fonction d'activation alternative contribue à la flexibilité et à la puissance des réseaux neuronaux.

Un volet significatif de notre réflexion porte sur les stratégies d'optimisation du modèle, offrant ainsi une vision approfondie des composants essentiels des réseaux neuronaux à plusieurs couches, abordant à la fois les aspects théoriques et pratiques inhérents à leur mise en œuvre.

2.2.2. Denoising Auto-Encoder

L'auto-encodeur de débruitage constitue un algorithme d'apprentissage non supervisé qui repose sur le concept de renforcer la robustesse des représentations apprises face à une corruption partielle du motif d'entrée. Dans le cadre de notre projet, nous intégrons l'Auto-Encoder de débruitage au cours d'une phase de pré-entraînement non supervisée, visant ainsi à capturer des caractéristiques essentielles des données sans l'aide d'étiquettes explicites.

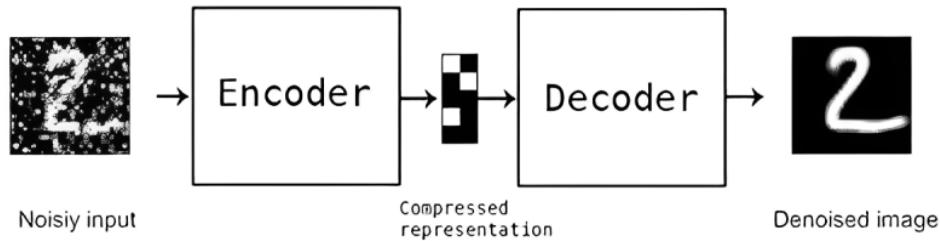


Figure – Un auto-encodeur de débruitage qui traite une image bruitée, générant une image nette du côté de la sortie

Notre approche s'inspire d'une stratégie bien établie, à savoir l'utilisation d'un taux d'apprentissage décroissant de manière exponentielle et d'une augmentation linéaire de l'élan, directement tirée de la technique du dropout. Cette dernière est une méthode de régularisation couramment utilisée dans les réseaux de neurones pour prévenir le surajustement en désactivant aléatoirement certains neurones pendant l'entraînement. En adaptant cette stratégie au contexte de l'auto-encodeur de débruitage, nous cherchons à renforcer la capacité du modèle à générer des représentations robustes et significatives, même en présence de bruit ou de variations partielles dans les données d'entrée.

2.2.3. Dropout

Le dropout est une technique de régularisation pour les réseaux neuronaux qui consiste à omettre de manière aléatoire des unités (ainsi que leurs connexions) pendant l'entraînement avec une probabilité spécifiée p (une valeur courante étant $p=0.5$). Lors de la phase de test, toutes les unités sont présentes, mais avec des poids mis à l'échelle par p (c'est-à-dire que w devient pw)⁷.

L'idée est de prévenir la co-adaptation, où le réseau neuronal devient trop dépendant de connexions particulières, ce qui pourrait être symptomatique du surajustement. Intuitivement, le dropout peut être considéré comme créant un ensemble implicite de réseaux neuronaux

⁷ Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, «Dropout: A Simple Way to Prevent Neural Networks from Overfitting », Récupéré de : <https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>

Le dropout est implémenté par couche dans un réseau neuronal et peut être utilisé avec la plupart des types de couches, tels que les couches denses entièrement connectées, les couches convolutionnelles et les couches récurrentes telles que les réseaux LSTM. Il peut être utilisé sur toutes les couches cachées du réseau ainsi que sur la couche visible ou d'entrée, mais n'est pas utilisé sur la couche de sortie. Lors de l'entraînement, certaines sorties de couche sont ignorées de manière aléatoire, ce qui a pour effet de rendre la couche similaire à une couche avec un nombre différent de neurones et une connectivité différente avec la couche précédente.

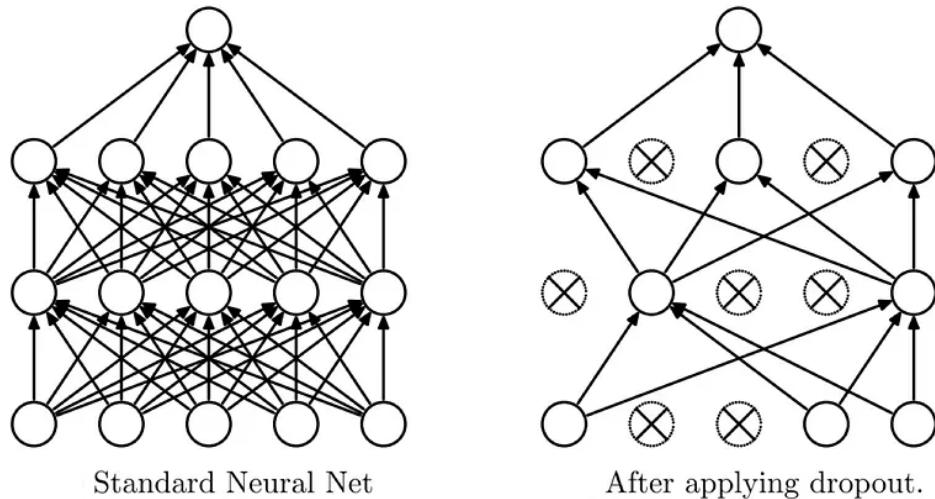


Figure – Application de la technique de Dropout

En résumé, le dropout est une technique de régularisation qui permet de prévenir le surajustement en omettant de manière aléatoire des unités pendant l'entraînement, ce qui crée un ensemble implicite de réseaux neuronaux et augmente la robustesse du modèle⁸.

2.2.4. Entraînement avec les pseudo-étiquettes comme régularisation de l'entropie

La pseudo-étiquette fonctionne dans le cadre de l'apprentissage semi-supervisé pour améliorer les performances de généralisation des réseaux neuronaux profonds. Deux mécanismes principaux sont utilisés à cette fin : la séparation de faible densité entre les classes et la régularisation de l'entropie⁹.

La séparation de faible densité entre les classes repose sur l'hypothèse de regroupement, selon laquelle la frontière de décision devrait se trouver dans des régions de faible densité pour améliorer les performances de généralisation.

⁸ « Dilution(neural networks) », Récupéré de : [https://en.wikipedia.org/wiki/Dilution_\(neural_networks\)](https://en.wikipedia.org/wiki/Dilution_(neural_networks))

⁹ Nicolas Chapados, Yoshua Bengio, « Comment améliorer la capacité de généralisation des algorithmes d'apprentissage pour la prise de décisions financières », Récupéré de : <https://cirano.qc.ca/fr/sommaries/2003s-20>

Cela est réalisé en encourageant la sortie du réseau à être insensible aux variations des directions de la variété de basse dimension, ce qui permet de réduire le chevauchement entre les classes et donc la densité des points de données à la frontière de décision.

D'autre part, la régularisation de l'entropie vise à maximiser la log-vraisemblance conditionnelle des données étiquetées tout en minimisant l'entropie des données non étiquetées. Cela se fait en utilisant un régularisateur basé sur l'incorporation pour équilibrer les deux termes de l'estimation maximum a posteriori, où la réduction de l'entropie des données non étiquetées contribue à améliorer les performances de généralisation.

L'entraînement avec les pseudo-étiquettes comme régularisation de l'entropie est une technique utilisée pour améliorer la généralisation des modèles d'apprentissage automatique, en particulier dans le cadre de l'apprentissage faiblement supervisé. Cette approche consiste à utiliser des pseudo-étiquettes sur des données non étiquetées pour augmenter le nombre d'exemples utilisés pendant l'entraînement. Cette méthode vise à régulariser l'entropie du modèle en introduisant des informations supplémentaires provenant des pseudo-étiquettes, ce qui peut améliorer la capacité du modèle à généraliser à de nouvelles données¹⁰.

La régularisation de l'entropie avec les pseudo-étiquettes peut être formulée comme suit:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^C y_{i,j} \log(\tilde{y}_{i,j})$$

Où :

Cette régularisation de l'entropie avec les pseudo-étiquettes permet d'exploiter les données non étiquetées de manière efficace, en fournissant des informations supplémentaires au modèle pendant l'entraînement, ce qui peut conduire à une amélioration de la généralisation et de la capacité du modèle à gérer de nouvelles classes ou des exemples non vus auparavant

- \mathcal{L} : la perte d'entropie croisée régularisée,
- N : le nombre d'exemples dans l'ensemble d'entraînement,
- C : le nombre de classes,
- $y_{i,j}$: la véritable étiquette de l'exemple
- $\tilde{y}_{i,j}$: la pseudo-étiquette prédite pour l'exemple et la classe j.

En résumé, la pseudo-étiquette fonctionne en exploitant les données non étiquetées pour améliorer la capacité de généralisation des algorithmes d'apprentissage, en utilisant des mécanismes tels que la séparation de faible densité entre les classes et la régularisation de l'entropie.

¹⁰ Colin Troisemaine, Joachim Flocon-Cholet, Stéphane Gosselin, Sandrine Vaton, Alexandre Reiffers-Masson, Vincent Lemaire, « Découvrir de nouvelles classes dans des données tabulaires », Récupéré de : <https://arxiv.org/pdf/2211.16352.pdf>

3. Implémentation

L'idée centrale consiste à utiliser un modèle pré-entraîné sur des données étiquetées pour prédire les étiquettes des données non étiquetées, puis à utiliser ces prédictions comme étiquettes réelles pour entraîner le modèle de manière supervisée de manière itérative.

L'article met en avant la simplicité d'implémentation de la méthode Pseudo-Label, soulignant sa capacité à exploiter les informations des données non étiquetées pour améliorer les performances de prédiction. Dans le cadre de notre étude, cette méthode sera appliquée à un échantillon de 100 images étiquetées du jeu de données MNIST, avec pour objectif d'évaluer ses performances en comparaison avec un modèle CNN traditionnel.

En complément, l'article introduit l'utilisation de la méthode Dropout en conjonction avec Pseudo-Label. Le Dropout consiste à désactiver aléatoirement certains neurones lors de l'entraînement pour réduire le surapprentissage. Cette combinaison vise à renforcer la capacité du modèle à généraliser les motifs présents dans les données.

En résumé, cette étude évalue l'efficacité de la méthode Pseudo-Label, combinée au Dropout, dans le contexte de l'apprentissage semi-supervisé, avec pour objectif de déterminer si cette approche améliore significativement les performances de prédiction par rapport à un modèle CNN traditionnel.

3.1. Architecture de l'algorithme

L'algorithme utilisé dans ce travail est le semi-supervised learning (apprentissage semi-supervisé) avec l'utilisation de modèles de réseaux de neurones (NN) sur le jeu de données MNIST digits. Les étapes suivies dans ce travail sont les suivantes :

- Mélanger les données MNIST avant l'entraînement pour prévenir les biais liés à l'ordre des données, favoriser une convergence plus rapide du modèle et améliorer sa capacité à généraliser à de nouvelles données.
- Utiliser un modèle de réseau de neurones (NN) comme modèle de base dans un cadre supervisé pour évaluer ses performances.
- Entraîner le modèle NN avec les données étiquetées uniquement pendant un certain nombre d'époques. Afficher les métriques d'entraînement et de validation à chaque époque.
- Utiliser un modèle de réseau de neurones avec des couches de dropout pour améliorer les performances. Entraîner le modèle avec les données étiquetées uniquement pendant un certain nombre d'époques. Afficher les métriques d'entraînement et de validation à chaque époque.
- Mettre en œuvre le pseudo-étiquetage, où le modèle prédit des étiquettes pour les données non étiquetées et utilise ces prédictions comme pseudo-étiquettes pour l'entraînement. Entraîner le modèle avec une combinaison de données étiquetées et pseudo-étiquetées pendant un certain nombre d'époques. Afficher les métriques d'entraînement et de validation à chaque époque.
- Évaluer les modèles sur l'ensemble de test en calculant la matrice de confusion et le rapport de classification. Afficher la précision de chaque modèle.

En résumé, l'algorithme utilisé dans ce travail consiste à entraîner des modèles de réseaux de neurones sur le jeu de données MNIST digits en utilisant différentes approches, telles que l'entraînement supervisé, l'utilisation de couches de dropout et le pseudo-étiquetage, pour améliorer les performances de classification. Les modèles sont évalués sur l'ensemble de test pour mesurer leur précision.

3.2. Entrainement des modèles sur MNIST digits

Les étapes et les fonctions utilisées dans le code sont les suivantes :

1. **Importation des bibliothèques nécessaires** : Les bibliothèques nécessaires pour le projet sont importées, y compris les bibliothèques TensorFlow et scikit-learn.
2. **Chargement des données MNIST** : Le jeu de données MNIST est chargé à l'aide de la fonction `mnist.load_data()` de TensorFlow. Les données sont divisées en ensembles d'entraînement et de test.
3. **Mélange des données** : Les données MNIST sont mélangées pour prévenir les biais liés à l'ordre des données et favoriser une convergence plus rapide du modèle. Cela est fait en utilisant la fonction `shuffle()` de scikit-learn.
4. **Préparation des données pour l'entraînement semi-supervisé** : Les données sont préparées en sélectionnant un petit sous-ensemble d'images étiquetées pour l'entraînement supervisé et en utilisant le reste des images comme données non étiquetées. Les données sont également normalisées et remodelées pour l'entraînement.
5. **Création du modèle de réseau de neurones (NN)** : Un modèle de réseau de neurones simple est créé en utilisant la classe `Sequential()` de TensorFlow. Des couches denses sont ajoutées au modèle avec des fonctions d'activation ReLU et sigmoïde.
6. **Compilation du modèle** : Le modèle est compilé en spécifiant l'optimiseur, la fonction de perte et les métriques à utiliser lors de l'entraînement. Dans ce cas, l'optimiseur est le SGD Optimizer, la fonction de perte est '`sparse_categorical_crossentropy`' et la métrique est l'exactitude (accuracy).
7. **Entraînement du modèle NN** : Le modèle NN est entraîné avec les données étiquetées uniquement pendant un certain nombre d'époques. L'historique d'apprentissage est enregistré pour l'analyse ultérieure.
8. **Création du modèle DropNN** : Un modèle de réseau de neurones avec des couches de dropout est créé en utilisant la classe `Sequential()`. Les couches de dropout sont ajoutées pour prévenir le surajustement et améliorer la généralisation du modèle.
9. **Compilation du modèle DropNN** : Le modèle DropNN est compilé de la même manière que le modèle NN, en spécifiant l'optimiseur, la fonction de perte et les métriques.
10. **Entraînement du modèle DropNN** : Le modèle DropNN est entraîné avec les données étiquetées uniquement pendant un certain nombre d'époques. L'historique d'apprentissage est enregistré pour l'analyse ultérieure.
11. **Prédiction avec Pseudo-Étiquetage** : Le modèle DropNN prédit des étiquettes pour les données non étiquetées et ces prédictions sont utilisées comme pseudo-étiquettes pour l'entraînement. Les données étiquetées et pseudo-étiquetées sont utilisées pour l'entraînement du modèle DropNN.
12. **Évaluation des modèles** : Les modèles NN et DropNN sont évalués sur l'ensemble de test en calculant la matrice de confusion, la précision et d'autres métriques d'évaluation.

En résumé, les étapes comprennent le chargement des données MNIST, la préparation des données pour l'entraînement semi-supervisé, la création et l'entraînement de modèles de réseau de neurones (NN et DropNN) avec différentes configurations, l'utilisation de couches de dropout pour prévenir le surajustement, et l'évaluation des modèles sur l'ensemble de test.

4. Résultats

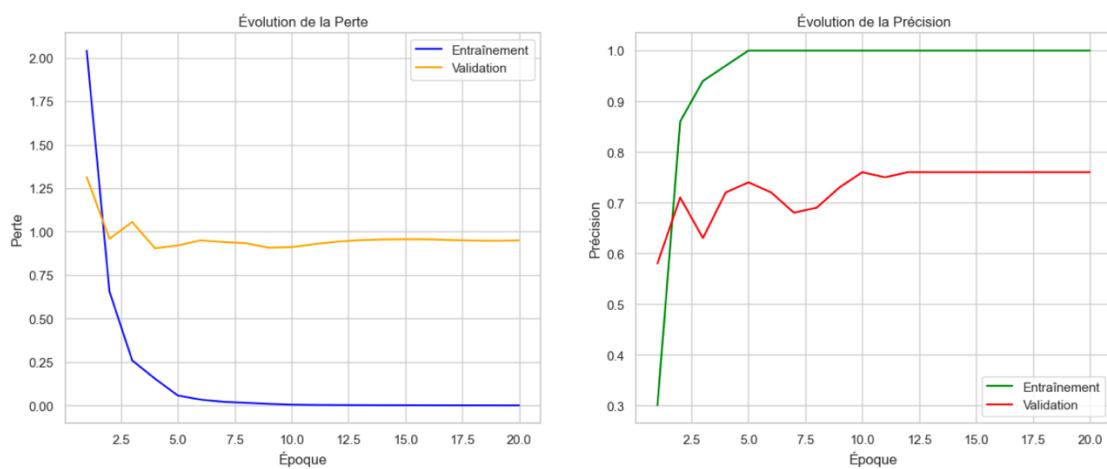
Nous avons réalisé des expérimentations qui ont englobé différents scénarios, ce qui nous permettra de mieux effectuer des comparaisons :

Modèle	Résultat de l'article	Résultat de notre méthode
NN	74,19%	78,74%
CNN	N. A	75,22%
DROP NN	78,11%	74,79%
DROP NN+PL	83,85%	79,22%
CNN+PL	N. A	78,15%

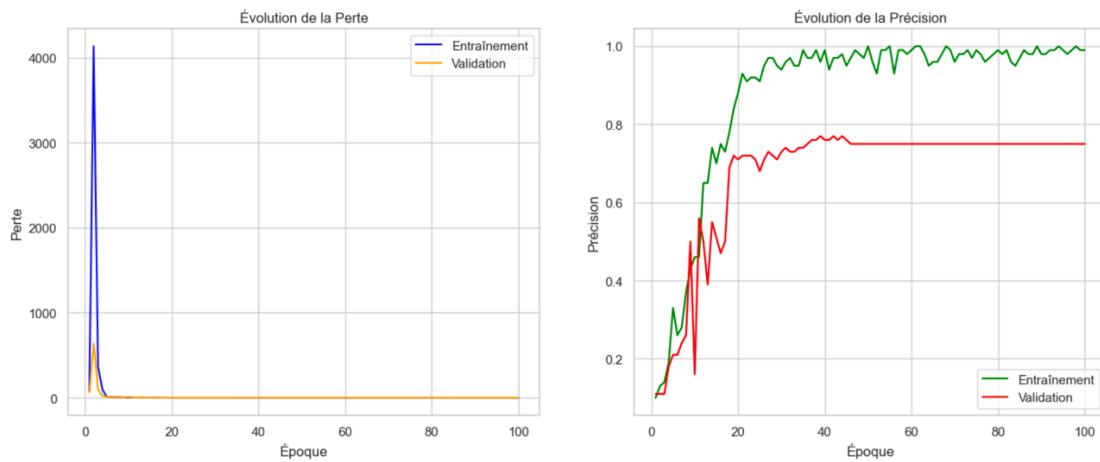
D'après le tableau fourni, nous pouvons tirer les conclusions suivantes :

- **Modèle NN** : Selon les résultats de l'article, le modèle NN a obtenu une précision de 74,19%. Cependant, en utilisant notre méthode, le modèle NN a atteint une précision de 78,74%. Cela indique que notre méthode a amélioré les performances du modèle NN par rapport aux résultats de l'article.
- **Modèle CNN** : Les résultats de l'article ne fournissent pas de précision pour le modèle CNN. Cependant, en utilisant notre méthode, le modèle CNN a atteint une précision de 75,22%. Cela suggère que notre méthode a permis d'obtenir de bonnes performances pour le modèle CNN.
- **Modèle DropNN** : Selon les résultats de l'article, le modèle DropNN a obtenu une précision de 78,11%. Cependant, en utilisant notre méthode, le modèle DropNN a atteint une précision de 74,79%. Cela indique que notre méthode a légèrement réduit les performances du modèle DropNN par rapport aux résultats de l'article.
- **Modèle DropNN+PL** : Selon les résultats de l'article, le modèle DropNN+PL a obtenu une précision de 83,85%. En utilisant notre méthode, le modèle DropNN+PL a atteint une précision de 79,22%. Bien que notre méthode ait légèrement réduit les performances par rapport aux résultats de l'article, le modèle DropNN+PL reste le modèle le plus performant parmi tous les modèles évalués.

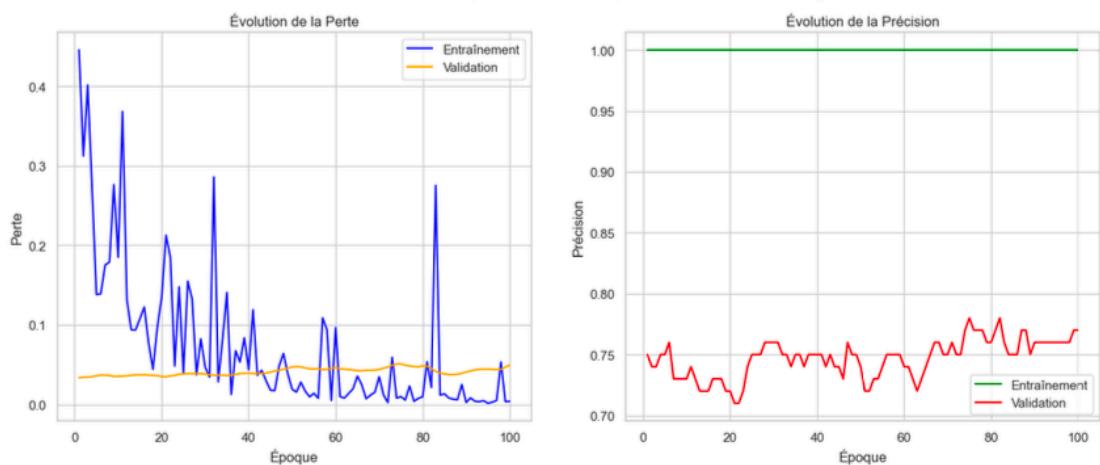
Visualisation de l'historique du modèle de réseau de neurones (NN)



Visualisation de l'historique du modèle de réseau de neurones avec 'Couches Dropout'



Visualisation de l'historique du modèle 'DropNN' avec 'PseudoLabeling'



		Matrice de Confusion									
		0	1	2	3	4	5	6	7	8	9
Réel	0	947	1	7	0	2	3	13	4	3	0
	1	0	1050	13	5	2	1	4	0	60	0
	2	18	13	834	28	11	1	11	14	89	13
	3	10	1	17	818	1	91	9	13	29	21
	4	3	4	4	1	709	0	11	5	26	219
	5	34	11	5	70	18	494	18	7	162	73
	6	29	7	59	6	72	5	755	1	16	8
	7	2	24	63	10	21	2	1	807	10	88
	8	26	15	10	40	22	22	9	8	803	19
	9	10	6	6	13	93	7	1	21	19	833

En conclusion, notre méthode a permis d'améliorer les performances des modèles NN et CNN par rapport aux résultats de l'article. Cependant, pour les modèles DropNN et DropNN+PL, notre méthode a légèrement réduit les performances par rapport aux résultats de l'article. Le modèle DropNN+PL reste le modèle le plus performant parmi tous les modèles évalués, avec une précision de **79,22%**.

Conclusion

En définitive, on conclut que les modèles de réseaux de neurones (NN) entraînés sur le jeu de données MNIST digits ont montré de bonnes performances en termes de perte et de précision.

Le premier modèle de réseau de neurones (NN) a été entraîné pendant 20 époques et a atteint une perte moyenne finale de 0.0014 et une précision moyenne finale de 1.0000. La perte de validation moyenne finale était de 0.8708 et la précision de validation moyenne finale était de 0.7800.

Le deuxième modèle, le modèle de réseau de neurones avec des couches de dropout (DropNN), a été entraîné pendant 100 époques. Il a montré une amélioration des performances par rapport au premier modèle, avec une perte moyenne finale de 0.0348 et une précision moyenne finale de 0.8100. La perte de validation moyenne finale était de 0.0374 et la précision de validation moyenne finale était de 0.7700. Ces résultats indiquent que l'utilisation de couches de dropout dans le modèle DropNN a permis d'améliorer la généralisation du modèle et de réduire le surajustement. Cependant, il est important de noter que la précision de validation n'a pas augmenté de manière significative par rapport au modèle de base.

En conclusion, les modèles de réseaux de neurones entraînés sur le jeu de données MNIST digits ont montré de bonnes performances, mais il y a encore de la place pour améliorer les résultats. D'autres techniques d'optimisation et d'ajustement des hyperparamètres pourraient être explorées pour obtenir de meilleures performances.

Références

- [1] Jayant Verma, « MNIST Dataset in Python - Basic Importing and Plotting », Récupéré de :
<https://www.digitalocean.com/community/tutorials/mnist-dataset-in-python>
- [2] Hugging Face. (s. d.). MNIST Dataset. Récupéré de <https://huggingface.co/datasets/mnist>
- [3] Deval Shah, Abhishek Jha, « Self-Supervised Learning and Its Applications », Recupéré de :
<https://neptune.ai/blog/self-supervised-learning>
- [4] Yiping Gao, « A semi-supervised convolutional neural network-based method for steel surface defect recognition », Récupéré de : https://www.researchgate.net/publication/336665610_A_semi-supervised_convolutional_neural_network-based_method_for_steele_surface_defect_recognition
- [5] « Apprentissage semi-supervisé », Récupéré de : https://fr.wikipedia.org/wiki/Apprentissage_semi-supervis%C3%A9
- [6] Mayank Mishra, « Convolutional Neural Networks, Explained », Récupéré de :
<https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939>
- [7] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, «Dropout: A Simple Way to Prevent Neural Networks from Overfitting », Récupéré de :
<https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf>
- [8] « Dilution(neural networks) », Récupéré de :
[https://en.wikipedia.org/wiki/Dilution_\(neural_networks\)](https://en.wikipedia.org/wiki/Dilution_(neural_networks))
- [9] Nicolas Chapados, Yoshua Bengio, « Comment améliorer la capacité de généralisation des algorithmes d'apprentissage pour la prise de décisions financières », Récupéré de :
<https://cirano.qc.ca/fr/sommaries/2003s-20>
- [10] Colin Troisemaine, Joachim Flocon-Cholet, Stéphane Gosselin, Sandrine Vaton, Alexandre Reiffers-Masson, Vincent Lemaire, « Découvrir de nouvelles classes dans des données tabulaires », Récupéré de : <https://arxiv.org/pdf/2211.16352.pdf>

Annexes

https://colab.research.google.com/drive/1NaKiFQkd3NyHNvkcu_yVJVe7nva4faAy

Projet d'Apprentissage Automatique avec Deep Learning

##

Apprentissage Semi-Supervisé sur MNIST digits

Membres du Groupe :

- Meryem Belkaid - AMSD
- Hafsa Boanani - AMSD
- Abir Oumghar - AMSD
- sanae hakimi - MLSD

```
[ ]: # <p style="text-align:center" color="red"><span style="color:red">Projet  
↪d'Apprentissage Automatique avec `Deep Learning`</span></p>  
## <p style="text-align:center" color="red"><span style="color:  
↪green">Apprentissage `Semi-Supervisé` sur `MNIST digits`</span></p>  
  
**Membres du Groupe :**  
  
- Meryem Belkaid - `AMSD`  
- Hafsa Boanani - `AMSD`  
- Abir Oumghar - `AMSD`  
- sanae hakimi - `MLSD`
```

1.0.1 Importation des bibliothèques nécessaires à notre projet

```
[1]: # Ignore warnings in Jupyter Notebook  
import warnings  
warnings.filterwarnings('ignore')  
  
# Imports related to TensorFlow and Keras  
import tensorflow as tf  
from tensorflow import keras  
from tensorflow.keras.models import Sequential
```

```

from tensorflow.keras.layers import (
    Dense, Conv2D, MaxPooling2D, Flatten, Activation, Dropout
)
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# Imports related to data manipulation with scikit-learn
from sklearn.model_selection import StratifiedShuffleSplit
from sklearn.utils import shuffle

# Classification evaluation metrics
from sklearn.metrics import confusion_matrix, accuracy_score, classification_report, log_loss

# Import related to image processing with OpenCV
import cv2

# Imports related to visualization
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

# Import related to numerical operations with NumPy
import numpy as np

# Other libraries
import time

```

1.0.2 Charger le jeu de données MNIST digits

MNIST digits est un ensemble de données qui comprend des images de chiffres manuscrits largement utilisé dans le domaine de l'apprentissage automatique. Il sert souvent de banc d'essai pour évaluer les performances des algorithmes de classification d'images. — Wikipedia

Plus d'informations..

```
[2]: # Charger le jeu de données MNIST
(x_train_complet, y_train_complet), (x_test, y_test) = mnist.load_data()

# Affichage du nombre d'échantillons dans le jeu d'entraînement et de test
print(f"Nombre d'échantillons dans l'ensemble d'entraînement : {len(x_train_complet)}")
print(f"Nombre d'échantillons dans l'ensemble de test : {len(x_test)}")
```

Nombre d'échantillons dans l'ensemble d'entraînement : 60000
 Nombre d'échantillons dans l'ensemble de test : 10000

Exploration des données (Visualisation de 10 échantillons aléatoires)

```
[3]: # Générer des indices aléatoires
indices_aleatoires = np.random.choice(len(x_train_complet), 10, replace=False)

# Créer le graphique avec une seule rangée et 10 colonnes
fig, axs = plt.subplots(1, 10, figsize=(18, 2))
fig.suptitle('Échantillons Aléatoires de l\'Ensemble de données MNIST Digits', fontsize=16)

for i, index in enumerate(indices_aleatoires):
    # Afficher l'étiquette
    label = y_train_complet[index]
    axs[i].set_title(f"Label : {label}", fontsize=10)

    # Remodeler l'image et l'afficher
    img = x_train_complet[index].reshape(28, 28)
    axs[i].imshow(img, cmap='gray')
    axs[i].axis('off') # Désactiver l'axe pour une meilleure présentation

# Ajuster la disposition pour éviter la superposition
plt.tight_layout(rect=[0, 0.03, 1, 0.95])

# Afficher le graphique
plt.show()
```



1.0.3 Préparation des Données pour l'Entraînement Semi-Supervisé

```
[4]: # Paramètres pour la sélection des données
count_etaquetees_par_classe = 10
count_validation = 100
count_non_etaquetees_test = 59800

# Sélectionner un petit sous-ensemble pour les données étiquetées
etaquetees_indices = np.concatenate([np.random.choice(np.where(y_train_complet ==
    == classe)[0], size=count_etaquetees_par_classe, replace=False) for classe in
    np.unique(y_train_complet)])
x_etaquetees, y_etaquetees = x_train_complet[etaquetees_indices], y_train_complet[etaquetees_indices]

# Utiliser le reste des données comme non étiquetées
```

```

x_non_etiquetees, y_non_etiquetees = np.delete(x_train_complet, □
    ↪etiquetees_indices, axis=0), np.delete(y_train_complet, etiquetees_indices, □
    ↪axis=0)

# Utiliser StratifiedShuffleSplit pour diviser les données restantes en
    ↪ensembles d'entraînement et de validation
sss = StratifiedShuffleSplit(n_splits=1, test_size=count_validation, □
    ↪random_state=42)
for train_index, validation_index in sss.split(x_non_etiquetees, □
    ↪y_non_etiquetees):
    x_validation, y_validation = x_non_etiquetees[validation_index], □
    ↪y_non_etiquetees[validation_index]
    x_non_etiquetees, y_non_etiquetees = x_non_etiquetees[train_index], □
    ↪y_non_etiquetees[train_index]

# Prendre les échantillons restants comme non étiquetés pour le test
x_non_etiquetees_test, y_non_etiquetees_test = x_non_etiquetees[::
    ↪count_non_etiquetees_test], y_non_etiquetees[:count_non_etiquetees_test]

# Mélanger les ensembles de données de la même manière
shuffle_indices = np.random.permutation(len(x_etiquetees))
x_etiquetees, y_etiquetees = x_etiquetees[shuffle_indices], □
    ↪y_etiquetees[shuffle_indices]

shuffle_indices = np.random.permutation(len(x_validation))
x_validation, y_validation = x_validation[shuffle_indices], □
    ↪y_validation[shuffle_indices]

shuffle_indices = np.random.permutation(len(x_non_etiquetees))
x_non_etiquetees, y_non_etiquetees_test = x_non_etiquetees[shuffle_indices], □
    ↪y_non_etiquetees_test[shuffle_indices]

# Afficher des informations sur les ensembles étiquetés, non étiquetés et de
    ↪validation
print(f"Nombre d'échantillons étiquetés dans l'ensemble d'entraînement : □
    ↪{len(x_etiquetees)}")
print(f"\nNombre de classes uniques dans les données étiquetées : {len(np.
    ↪unique(y_etiquetees))}")
print(f"Nombre d'échantillons non étiquetés dans l'ensemble de test : □
    ↪{len(x_non_etiquetees)}")
print(f"\nNombre d'échantillons dans l'ensemble de validation : □
    ↪{len(x_validation)}")
print(f"Nombre de classes uniques dans l'ensemble de validation : {len(np.
    ↪unique(y_validation))}")

```

Nombre d'échantillons étiquetés dans l'ensemble d'entraînement : 100

```
Nombre de classes uniques dans les données étiquetées : 10
Nombre d'échantillons non étiquetés dans l'ensemble de test : 59800
```

```
Nombre d'échantillons dans l'ensemble de validation : 100
Nombre de classes uniques dans l'ensemble de validation : 10
```

Jusqu'à présent, nous avons soigneusement sélectionné 100 images étiquetées équilibrées pour l'entraînement et 100 pour la validation, en respectant l'équilibre des données. Nous avons utilisé la méthode **Stratified Shuffle Split** de Scikit-learn pour assurer cet équilibre. Les 59,800 images restantes seront utilisées dans le cadre de l'apprentissage semi-supervisé.

1.0.4 Prétraitement & Normalisation des Données

```
[5]: # Normaliser les valeurs des pixels des images d'entrée dans la plage [0, 1]
x_etaquetees, x_non_etaquetees, x_validation, x_test = x_etaquetees / 255.0,
    ↪x_non_etaquetees / 255.0, x_validation / 255.0, x_test / 255.0

# Redimensionner les données en une seule dimension (pour les images)
taille_pixels = 28 * 28 # La taille des images MNIST est de 28x28 pixels
x_etaquetees = x_etaquetees.reshape(x_etaquetees.shape[0], taille_pixels)
x_non_etaquetees = x_non_etaquetees.reshape(x_non_etaquetees.shape[0], ↪
    ↪taille_pixels)
x_validation = x_validation.reshape(x_validation.shape[0], taille_pixels)
x_test = x_test.reshape(x_test.shape[0], taille_pixels)

# Afficher des informations sur les ensembles après prétraitement
print("Ensembles après prétraitement :")
print(f"Taille de l'ensemble étiqueté : {x_etaquetees.shape}")
print(f"Taille de l'ensemble non étiqueté : {x_non_etaquetees.shape}")
print(f"Taille de l'ensemble de validation : {x_validation.shape}")
print(f"Taille de l'ensemble de test : {x_test.shape}")
```

```
Ensembles après prétraitement :
Taille de l'ensemble étiqueté : (100, 784)
Taille de l'ensemble non étiqueté : (59800, 784)
Taille de l'ensemble de validation : (100, 784)
Taille de l'ensemble de test : (10000, 784)
```

La normalisation des données MNIST est réalisée pour accélérer la convergence de l'entraînement, assurer la stabilité numérique et garantir une comparaison équitable entre les caractéristiques. Elle prévient également le surajustement en régularisant les poids du modèle.

```
[6]: # Mélanger les données pour éviter que l'ordre ait un impact sur l'entraînement
tf.random.set_seed(42)
x_etaquetees, y_etaquetees = shuffle(x_etaquetees, y_etaquetees)
```

Mélanger les données MNIST avant l'entraînement prévient les biais liés à l'ordre des données, favorise une convergence plus rapide du modèle, et améliore sa capacité à généraliser à de nouvelles données.

##

Modèle 1 : Modèle de Réseau de Neurones (NN) sur 100 Échantillons

- Dans un premier temps, comme mentionné dans l'article, nous allons utiliser le modèle proposé, le modèle NN, qui est un modèle simple. Nous allons d'abord l'utiliser dans un cadre supervisé et évaluer ses performances.

Modèle de Réseau de Neurones (NN) est une architecture de réseau neuronal artificiel récurrent (RNN) utilisée dans le domaine de l'apprentissage profond. Contrairement aux réseaux neuronaux standard à propagation avant, le LSTM possède des connexions de rétroaction. Il peut traiter non seulement des points de données individuels (comme des images), mais aussi des séquences entières de données (telles que la parole ou la vidéo). — Wikipedia

[Plus d'informations..](#)

Création du modèle de réseau de neurones

```
[7]: modele_reseau_neurones = Sequential()
# Ajout d'une couche cachée avec 5000 neurones et une fonction d'activation ReLU
modele_reseau_neurones.add(Dense(5000, input_dim=784, activation='relu'))
# Ajout de la couche de sortie avec 10 neurones et une fonction d'activation
# sigmoïde
modele_reseau_neurones.add(Dense(10, activation='sigmoid'))
# Affichage du résumé du modèle
modele_reseau_neurones.summary()
```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 5000)	3925000
dense_1 (Dense)	(None, 10)	50010

=====

Total params: 3975010 (15.16 MB)
Trainable params: 3975010 (15.16 MB)
Non-trainable params: 0 (0.00 Byte)

=====

Configuration du Modèle

```
[8]: # Compiler le modèle avec l'optimiseur Adam, la fonction de perte
# 'sparse_categorical_crossentropy' et la métrique de précision
modele_reseau_neurones.compile(optimizer='adam',
                                loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

```
# Affichage de la configuration du modèle
print("  Configuration du Modèle :")
print(f"  Optimiseur : Adam")
print(f"  Fonction de Perte : 'sparse_categorical_crossentropy'")
print(f"  Métriques : ['accuracy']")
```

```
Configuration du Modèle :
Optimiseur : Adam
Fonction de Perte : 'sparse_categorical_crossentropy'
Métriques : ['accuracy']
```

L'utilisation de `sparse_categorical_crossentropy` dans l'entraînement du modèle MNIST Digits : cette approche est motivée par la nature des étiquettes associées aux images du jeu de données. MNIST attribue des entiers aux chiffres de 0 à 9 sans nécessiter une encodage one-hot explicite. La fonction de perte '`sparse_categorical_crossentropy`' est donc appropriée pour ce scénario.

[En savoir plus sur '`sparse_categorical_crossentropy`'](#)

Apprentissage du Modèle avec les 100 Images Étiquetées Uniquement

```
[9]: # Apprentissage du modèle avec les données étiquetées (100 images) pour 20
      ↪époques
historique_apprentissage = modele_reseau_neurones.fit(x_etaquées,
                                                       y_etaquées,
                                                       epochs=20,
                                                       batch_size=32,
                                                      
                                                       ↪validation_data=(x_validation, y_validation))

# Affichage de l'historique d'apprentissage
print("\nHistorique d'Apprentissage :")
print(f" >> Nombre d'époques : {len(historique_apprentissage.history['loss'])}")
print(f" >> Perte moyenne finale : {historique_apprentissage.
      ↪history['loss'][-1]:.4f}")
print(f" >> Précision moyenne finale : {historique_apprentissage.
      ↪history['accuracy'][-1]:.4f}")
print(f" >> Validation Perte moyenne finale : {historique_apprentissage.
      ↪history['val_loss'][-1]:.4f}")
print(f" >> Validation Précision moyenne finale : {historique_apprentissage.
      ↪history['val_accuracy'][-1]:.4f}")
```

```
Epoch 1/20
4/4 [=====] - 1s 131ms/step - loss: 2.0413 - accuracy: 0.3000 - val_loss: 1.3139 - val_accuracy: 0.5800
Epoch 2/20
4/4 [=====] - 0s 54ms/step - loss: 0.6558 - accuracy: 0.8600 - val_loss: 0.9591 - val_accuracy: 0.7100
```

```
Epoch 3/20
4/4 [=====] - 0s 45ms/step - loss: 0.2595 - accuracy:
0.9400 - val_loss: 1.0550 - val_accuracy: 0.6300
Epoch 4/20
4/4 [=====] - 0s 51ms/step - loss: 0.1545 - accuracy:
0.9700 - val_loss: 0.9045 - val_accuracy: 0.7200
Epoch 5/20
4/4 [=====] - 0s 55ms/step - loss: 0.0580 - accuracy:
1.0000 - val_loss: 0.9206 - val_accuracy: 0.7400
Epoch 6/20
4/4 [=====] - 0s 58ms/step - loss: 0.0342 - accuracy:
1.0000 - val_loss: 0.9498 - val_accuracy: 0.7200
Epoch 7/20
4/4 [=====] - 0s 52ms/step - loss: 0.0213 - accuracy:
1.0000 - val_loss: 0.9405 - val_accuracy: 0.6800
Epoch 8/20
4/4 [=====] - 0s 49ms/step - loss: 0.0157 - accuracy:
1.0000 - val_loss: 0.9332 - val_accuracy: 0.6900
Epoch 9/20
4/4 [=====] - 0s 53ms/step - loss: 0.0097 - accuracy:
1.0000 - val_loss: 0.9078 - val_accuracy: 0.7300
Epoch 10/20
4/4 [=====] - 0s 56ms/step - loss: 0.0053 - accuracy:
1.0000 - val_loss: 0.9109 - val_accuracy: 0.7600
Epoch 11/20
4/4 [=====] - 0s 57ms/step - loss: 0.0037 - accuracy:
1.0000 - val_loss: 0.9281 - val_accuracy: 0.7500
Epoch 12/20
4/4 [=====] - 0s 54ms/step - loss: 0.0029 - accuracy:
1.0000 - val_loss: 0.9426 - val_accuracy: 0.7600
Epoch 13/20
4/4 [=====] - 0s 55ms/step - loss: 0.0025 - accuracy:
1.0000 - val_loss: 0.9510 - val_accuracy: 0.7600
Epoch 14/20
4/4 [=====] - 0s 50ms/step - loss: 0.0022 - accuracy:
1.0000 - val_loss: 0.9551 - val_accuracy: 0.7600
Epoch 15/20
4/4 [=====] - 0s 50ms/step - loss: 0.0019 - accuracy:
1.0000 - val_loss: 0.9568 - val_accuracy: 0.7600
Epoch 16/20
4/4 [=====] - 0s 52ms/step - loss: 0.0017 - accuracy:
1.0000 - val_loss: 0.9562 - val_accuracy: 0.7600
Epoch 17/20
4/4 [=====] - 0s 56ms/step - loss: 0.0015 - accuracy:
1.0000 - val_loss: 0.9516 - val_accuracy: 0.7600
Epoch 18/20
4/4 [=====] - 0s 54ms/step - loss: 0.0014 - accuracy:
1.0000 - val_loss: 0.9486 - val_accuracy: 0.7600
```

```

Epoch 19/20
4/4 [=====] - 0s 54ms/step - loss: 0.0013 - accuracy:
1.0000 - val_loss: 0.9472 - val_accuracy: 0.7600
Epoch 20/20
4/4 [=====] - 0s 52ms/step - loss: 0.0012 - accuracy:
1.0000 - val_loss: 0.9498 - val_accuracy: 0.7600

Historique d'Apprentissage :
>> Nombre d'époques : 20
>> Perte moyenne finale : 0.0012
>> Précision moyenne finale : 1.0000
>> Validation Perte moyenne finale : 0.9498
>> Validation Précision moyenne finale : 0.7600

```

Visualisation de l'historique du modèle de réseau de neurones

```

[10]: # Détermination du nombre d'époques
nb_epochs = len(historique_apprentissage.history['loss'])

# Configuration du style Seaborn
sns.set(style="whitegrid")

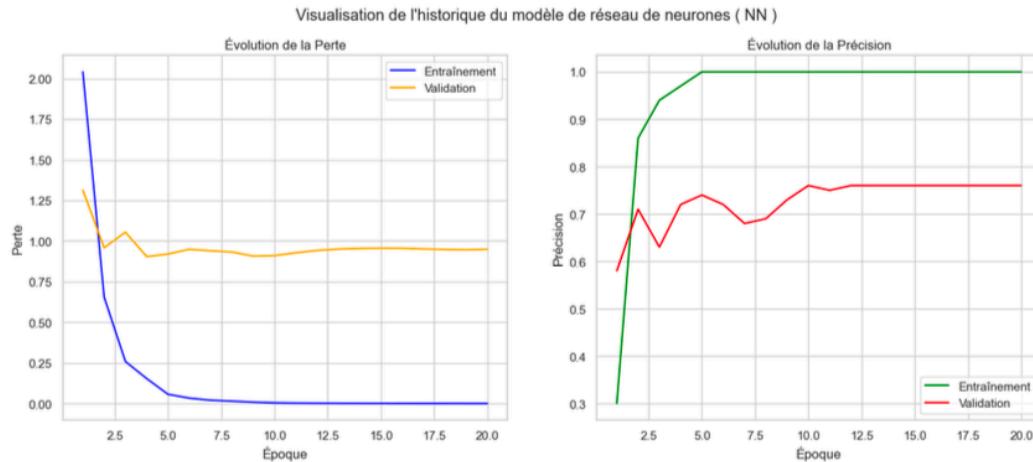
# Création de la figure avec une taille appropriée
plt.figure(figsize=(16, 6))

# Courbe de perte
plt.subplot(1, 2, 1)
sns.lineplot(x=range(1, nb_epochs + 1), y=historique_apprentissage.
             history['loss'], label='Entraînement', color='blue')
sns.lineplot(x=range(1, nb_epochs + 1), y=historique_apprentissage.
             history['val_loss'], label='Validation', color='orange')
plt.title('Évolution de la Perte')
plt.xlabel('Époque')
plt.ylabel('Perte')
plt.legend()
plt.grid(True)

# Courbe de précision
plt.subplot(1, 2, 2)
sns.lineplot(x=range(1, nb_epochs + 1), y=historique_apprentissage.
             history['accuracy'], label='Entraînement', color='green')
sns.lineplot(x=range(1, nb_epochs + 1), y=historique_apprentissage.
             history['val_accuracy'], label='Validation', color='red')
plt.title('Évolution de la Précision')
plt.xlabel('Époque')
plt.ylabel('Précision')
plt.legend()
plt.grid(True)

```

```
# Ajout du titre à la figure
plt.suptitle("Visualisation de l'historique du modèle de réseau de neurones ( NN )")
plt.show()
```



Évaluation et Visualisation des Métriques de Classification

```
[11]: # Prédiction sur l'ensemble de test
y_pred = np.argmax(modele_reseau_neurones.predict(x_test), axis=1)

# Calcul de la matrice de confusion
matrice_confusion = confusion_matrix(y_test, y_pred)

# Tracer la matrice de confusion avec un style amélioré
plt.figure(figsize=(5, 4))
sns.heatmap(matrice_confusion, annot=True, fmt="d", cmap="viridis", cbar=False,
            square=True,
            xticklabels=np.arange(10), yticklabels=np.arange(10), linewidths=.5, linecolor='gray')
plt.xlabel('Prédict', fontsize=12, fontweight='bold')
plt.ylabel('Réel', fontsize=12, fontweight='bold')
plt.title('Matrice de Confusion', fontsize=14, fontweight='bold')
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.show()

# Rapport de classification
rapport_classification = classification_report(y_test, y_pred,
                                                target_names=[str(i) for i in range(10)])
```

```

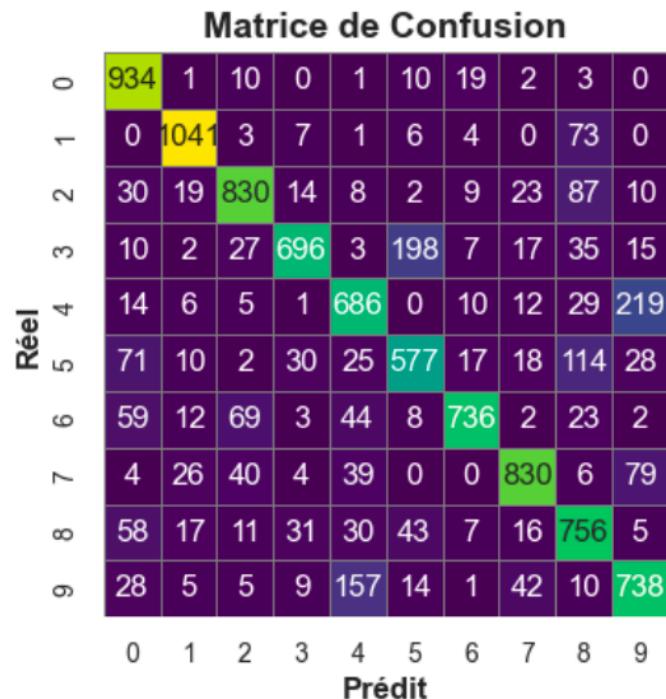
print("Rapport de Classification:\n", rapport_classification)

# Affichage des résultats d'évaluation
precision = accuracy_score(y_test, y_pred)

print("\nRésultats de l'Évaluation sur l'Ensemble de Test :")
print(f">> Précision : {precision}")

```

313/313 [=====] - 2s 6ms/step



Rapport de Classification:

	precision	recall	f1-score	support
0	0.77	0.95	0.85	980
1	0.91	0.92	0.92	1135
2	0.83	0.80	0.82	1032
3	0.88	0.69	0.77	1010
4	0.69	0.70	0.69	982
5	0.67	0.65	0.66	892
6	0.91	0.77	0.83	958
7	0.86	0.81	0.83	1028
8	0.67	0.78	0.72	974
9	0.67	0.73	0.70	1009

accuracy		0.78	10000
macro avg	0.79	0.78	10000
weighted avg	0.79	0.78	10000

Résultats de l'Évaluation sur l'Ensemble de Test :

>> Précision : 0.7824

##

Modèle 2 : Modèle de Réseau de Neurones avec drop Drop_NN sur 100 Échantillons

- Dans un deuxième temps, comme mentionné dans l'article, nous allons utiliser le modèle proposé, le modèle drop NN, qui est un modèle evalué de NN avec des couche de dropout. nous allons utilser ce modele qui a le drop out pour voir comment ces perfomances evoluent

Le concept de dropout dans le deep learning est une technique de régularisation utilisée pour prévenir le surajustement dans les réseaux neuronaux. Il consiste à aléatoirement “supprimer” (ignorer) un certain pourcentage de neurones pendant l'entraînement, ce qui permet d'améliorer la généralisation du modèle. Cette approche contribue souvent à obtenir des modèles plus robustes et performants.

[Plus d'informations..](#)

Création du modèle de Réseau de Neurones avec Couches Dropout

```
[12]: # Création du modèle DropNN : réseau de neurones avec un Dropout de 0.5 pour
      ↵ les couches cachées et 0.2 pour la couche d'entrée
modele_dropNN = Sequential()
# Ajout d'une couche Dropout pour les entrées avec un taux de 0.2
modele_dropNN.add(Dropout(0.2, input_shape=(784,)))
# Ajout d'une couche dense avec 5000 neurones et une fonction d'activation ReLU
modele_dropNN.add(Dense(5000, activation='relu'))
# Ajout d'une couche Dropout avec un taux de 0.5
modele_dropNN.add(Dropout(0.5))
# Ajout d'une couche dense avec 10 neurones et une fonction d'activation
      ↵ sigmoïde
modele_dropNN.add(Dense(10, activation='sigmoid'))

# Affichage du résumé du modèle
modele_dropNN.summary()
```

Model: "sequential_1"

Layer (type)	Output Shape	Param #
dropout (Dropout)	(None, 784)	0
dense_2 (Dense)	(None, 5000)	3925000

```

dropout_1 (Dropout)           (None, 5000)          0
dense_3 (Dense)              (None, 10)            50010
=====
Total params: 3975010 (15.16 MB)
Trainable params: 3975010 (15.16 MB)
Non-trainable params: 0 (0.00 Byte)
=====
```

Configuration de l'Optimiseur avec Taux d'Apprentissage Exponentiel

```
[13]: # Définition d'une planification de taux d'apprentissage exponentielle
planification_lr_exponentielle = keras.optimizers.schedules.ExponentialDecay(
    initial_learning_rate=1.5,
    decay_steps=10,
    decay_rate=0.88
)

# Création d'un optimiseur SGD avec la planification de taux d'apprentissage
# exponentielle
optimiseur = keras.optimizers.SGD(learning_rate=planification_lr_exponentielle)
```

Configuration du Modèle DropNN

```
[14]: modele_dropNN.compile(optimizer=optimiseur,
                         loss='sparse_categorical_crossentropy',
                         metrics=['accuracy'])

# Affichage de la configuration du modèle
print("Configuration du Modèle :")
print(f"    Optimiseur : SGD Optimizer")
print(f"    Fonction de Perte : 'sparse_categorical_crossentropy'")
print(f"    Métriques : ['accuracy']")
```

Configuration du Modèle :
 Optimiseur : SGD Optimizer
 Fonction de Perte : 'sparse_categorical_crossentropy'
 Métriques : ['accuracy']

Apprentissage du Modèle DropNN avec les 100 Données Étiquetées

```
[15]: # Entraînement du modèle DropNN avec les données étiquetées (100 époques, batch
      # size de 16) et validation
historique_dropNN = modele_dropNN.fit(
    x_etiquetees,
    y_etiquetees,
    epochs=100,
```

```

    batch_size=16,
    validation_data=(x_validation, y_validation)
)

# Affichage de l'historique d'apprentissage
print("\nHistorique d'Apprentissage :")
print(f" >> Nombre d'époques : {len(historique_dropNN.history['loss'])}")
print(f" >> Perte moyenne finale : {historique_dropNN.history['loss'][-1]:.4f}")
print(f" >> Précision moyenne finale : {historique_dropNN.
    history['accuracy'][-1]:.4f}")
print(f" >> Validation Perte moyenne finale : {historique_dropNN.
    history['val_loss'][-1]:.4f}")
print(f" >> Validation Précision moyenne finale : {historique_dropNN.
    history['val_accuracy'][-1]:.4f}")

```

```

Epoch 1/100
7/7 [=====] - 1s 52ms/step - loss: 81.4814 - accuracy:
0.1000 - val_loss: 64.3078 - val_accuracy: 0.1100
Epoch 2/100
7/7 [=====] - 0s 27ms/step - loss: 4132.9717 -
accuracy: 0.1300 - val_loss: 635.1395 - val_accuracy: 0.1100
Epoch 3/100
7/7 [=====] - 0s 28ms/step - loss: 363.3368 - accuracy:
0.1400 - val_loss: 99.5256 - val_accuracy: 0.1100
Epoch 4/100
7/7 [=====] - 0s 28ms/step - loss: 98.6816 - accuracy:
0.1900 - val_loss: 22.2817 - val_accuracy: 0.1800
Epoch 5/100
7/7 [=====] - 0s 31ms/step - loss: 11.4246 - accuracy:
0.3300 - val_loss: 9.9729 - val_accuracy: 0.2100
Epoch 6/100
7/7 [=====] - 0s 29ms/step - loss: 9.5525 - accuracy:
0.2600 - val_loss: 6.6100 - val_accuracy: 0.2100
Epoch 7/100
7/7 [=====] - 0s 31ms/step - loss: 6.4163 - accuracy:
0.2800 - val_loss: 7.6978 - val_accuracy: 0.2400
Epoch 8/100
7/7 [=====] - 0s 31ms/step - loss: 6.0588 - accuracy:
0.3700 - val_loss: 2.7495 - val_accuracy: 0.2600
Epoch 9/100
7/7 [=====] - 0s 30ms/step - loss: 2.4499 - accuracy:
0.4300 - val_loss: 1.9078 - val_accuracy: 0.5000
Epoch 10/100
7/7 [=====] - 0s 30ms/step - loss: 2.3366 - accuracy:
0.4600 - val_loss: 10.0556 - val_accuracy: 0.1600
Epoch 11/100
7/7 [=====] - 0s 29ms/step - loss: 3.2949 - accuracy:

```

```

0.9800 - val_loss: 0.9530 - val_accuracy: 0.7500
Epoch 92/100
7/7 [=====] - 0s 28ms/step - loss: 0.0544 - accuracy:
0.9900 - val_loss: 0.9530 - val_accuracy: 0.7500
Epoch 93/100
7/7 [=====] - 0s 27ms/step - loss: 0.0697 - accuracy:
0.9900 - val_loss: 0.9530 - val_accuracy: 0.7500
Epoch 94/100
7/7 [=====] - 0s 28ms/step - loss: 0.0569 - accuracy:
1.0000 - val_loss: 0.9529 - val_accuracy: 0.7500
Epoch 95/100
7/7 [=====] - 0s 26ms/step - loss: 0.0792 - accuracy:
0.9900 - val_loss: 0.9530 - val_accuracy: 0.7500
Epoch 96/100
7/7 [=====] - 0s 28ms/step - loss: 0.1090 - accuracy:
0.9800 - val_loss: 0.9529 - val_accuracy: 0.7500
Epoch 97/100
7/7 [=====] - 0s 27ms/step - loss: 0.0884 - accuracy:
0.9900 - val_loss: 0.9529 - val_accuracy: 0.7500
Epoch 98/100
7/7 [=====] - 0s 26ms/step - loss: 0.0578 - accuracy:
1.0000 - val_loss: 0.9529 - val_accuracy: 0.7500
Epoch 99/100
7/7 [=====] - 0s 29ms/step - loss: 0.0554 - accuracy:
0.9900 - val_loss: 0.9529 - val_accuracy: 0.7500
Epoch 100/100
7/7 [=====] - 0s 27ms/step - loss: 0.0879 - accuracy:
0.9900 - val_loss: 0.9529 - val_accuracy: 0.7500

```

Historique d'Apprentissage :

- >> Nombre d'époques : 100
- >> Perte moyenne finale : 0.0879
- >> Précision moyenne finale : 0.9900
- >> Validation Perte moyenne finale : 0.9529
- >> Validation Précision moyenne finale : 0.7500

Visualisation de l'historique du modèle de réseau de neurones avec Couches Dropout

```
[16]: # Détermination du nombre d'époques
nb_epochs = len(historique_dropNN.history['loss'])

# Configuration du style Seaborn
sns.set(style="whitegrid")

# Création de la figure avec une taille appropriée
plt.figure(figsize=(16, 6))

# Courbe de perte
```

```

plt.subplot(1, 2, 1)
sns.lineplot(x=range(1, nb_epochs + 1), y=historique_dropNN.history['loss'],  

             label='Entraînement', color='blue')
sns.lineplot(x=range(1, nb_epochs + 1), y=historique_dropNN.  

             history['val_loss'], label='Validation', color='orange')
plt.title('Évolution de la Perte')
plt.xlabel('Époque')
plt.ylabel('Perte')
plt.legend()
plt.grid(True)

# Courbe de précision
plt.subplot(1, 2, 2)
sns.lineplot(x=range(1, nb_epochs + 1), y=historique_dropNN.  

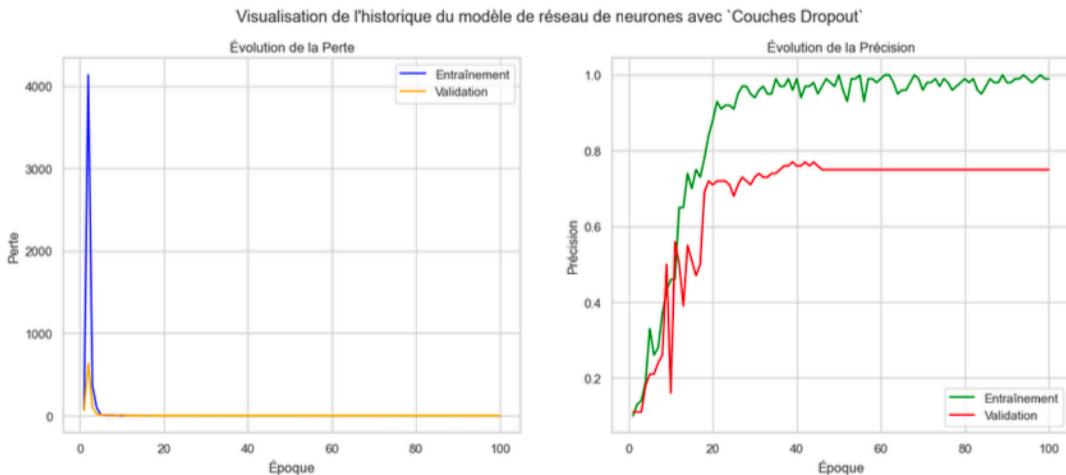
             history['accuracy'], label='Entraînement', color='green')
sns.lineplot(x=range(1, nb_epochs + 1), y=historique_dropNN.  

             history['val_accuracy'], label='Validation', color='red')
plt.title('Évolution de la Précision')
plt.xlabel('Époque')
plt.ylabel('Précision')
plt.legend()
plt.grid(True)

# Ajout du titre à la figure
plt.suptitle("Visualisation de l'historique du modèle de réseau de neurones  

              avec 'Couches Dropout'")
plt.show()

```



Évaluation et Visualisation des Métriques de Classification

```
[17]: # Prédiction sur l'ensemble de test
y_pred = np.argmax(modele_dropNN.predict(x_test), axis=1)

# Calcul de la matrice de confusion
matrice_confusion = confusion_matrix(y_test, y_pred)

# Tracer la matrice de confusion avec un style amélioré
plt.figure(figsize=(5, 4))
sns.heatmap(matrice_confusion, annot=True, fmt="d", cmap="viridis", cbar=False, square=True,
            xticklabels=np.arange(10), yticklabels=np.arange(10), linewidths=.5, linecolor='gray')
plt.xlabel('Prédit', fontsize=12, fontweight='bold')
plt.ylabel('Réel', fontsize=12, fontweight='bold')
plt.title('Matrice de Confusion', fontsize=14, fontweight='bold')
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.show()

# Rapport de classification
rapport_classification = classification_report(y_test, y_pred, target_names=[str(i) for i in range(10)])
print("Rapport de Classification:\n", rapport_classification)

# Affichage des résultats d'évaluation
precision = accuracy_score(y_test, y_pred)

print("\nRésultats de l'Évaluation sur l'Ensemble de Test :")
print(f">> Précision : {precision}")
```

313/313 [=====] - 2s 6ms/step

		Matrice de Confusion									
		0	1	2	3	4	5	6	7	8	9
Réel	0	943	0	10	4	2	7	10	0	3	1
	1	0	1038	22	15	1	11	6	0	42	0
	2	16	18	822	31	5	11	16	15	86	12
	3	9	0	19	806	0	116	7	13	33	7
	4	6	6	9	2	723	3	25	6	23	179
	5	86	7	7	73	7	531	21	6	123	31
	6	34	6	70	3	50	12	770	0	10	3
	7	5	19	59	8	29	2	1	825	16	64
	8	26	8	16	53	13	33	18	7	795	5
	9	13	5	18	14	161	11	1	26	28	732

Rapport de Classification:

	precision	recall	f1-score	support
0	0.83	0.96	0.89	980
1	0.94	0.91	0.93	1135
2	0.78	0.80	0.79	1032
3	0.80	0.80	0.80	1010
4	0.73	0.74	0.73	982
5	0.72	0.60	0.65	892
6	0.88	0.80	0.84	958
7	0.92	0.80	0.86	1028
8	0.69	0.82	0.75	974
9	0.71	0.73	0.72	1009
accuracy			0.80	10000
macro avg	0.80	0.80	0.79	10000
weighted avg	0.80	0.80	0.80	10000

Résultats de l'Évaluation sur l'Ensemble de Test :

>> Précision : 0.7985

##

Modèle 3 : DropNN avec PseudoLabeling (Semi-Supervisé)

- Comme discuté dans l'article, nous allons maintenant définir et implémenter le pseudo labeling dans le modèle DropNN.

Le Pseudo Labling dans l'apprentissage semi-supervisé est une technique où, en plus des exemples d'entraînement étiquetés, des exemples non étiquetés sont également utilisés pour l'entraînement. Les prédictions du modèle sur les données non étiquetées sont ensuite traitées comme des "pseudos-étiquettes" et sont intégrées dans le processus d'entraînement. Cette approche vise à améliorer les performances du modèle en utilisant un ensemble de données plus large et diversifié.

Plus d'informations sur le Pseudo dans le cadre de l'apprentissage semi-supervisé

Prédiction avec Pseudo-Étiquetage

```
[18]: # Prédiction des étiquettes pseudo avec le modèle DropNN sur les données non_étiquetées
predictions_pseudo_etaquettes = modele_dropNN.predict(x_non_etaqueteres)

# Initialisation de la liste des étiquettes pseudo
y_pseudo_etaquettes = []

# Extraction des étiquettes pseudo en prenant l'indice du maximum de chaque_prediction
for i in range(len(predictions_pseudo_etaquettes)):
    y_pseudo_etaquettes.append(np.argmax(predictions_pseudo_etaquettes[i]))
```

1869/1869 [=====] - 12s 6ms/step

Configuration de l'Optimiseur et de la Fonction de Perte pour le Pseudo-Étiquetage

```
[19]: # Configuration de l'optimiseur Adam pour le pseudo-étiquetage
optimiseur_pseudo_etaquetage = keras.optimizers.Adam()

# Configuration de la fonction de perte catégorielle sparse pour le_pseudo-étiquetage
fonction_perte_pseudo_etaquetage = keras.losses.SparseCategoricalCrossentropy(from_logits=True)
```

Préparation des Datasets d'Apprentissage : Images Labelisées et Pseudo-Étiquetées

```
[20]: # Tailles de lot (batch size) pour les données étiquetées et pseudo-étiquetées
taille_lot_etaquette = 32
taille_lot_PseudoEtaquette = 256

# Création des datasets d'apprentissage avec les images étiquetées et_pseudo-étiquetées
ensemble_apprentissage_etaquette = tf.data.Dataset.from_tensor_slices((x_etaqueteres, y_etaqueteres))
```

```

ensemble_apprentissage_etaquete = ensemble_apprentissage_etaquete.
    ↪shuffle(buffer_size=1024).batch(taille_lot_etaquete)

ensemble_apprentissage_PseudoEtiquette = tf.data.Dataset.
    ↪from_tensor_slices((x_non_etaquetees, y_pseudo_etaquettes))
ensemble_apprentissage_PseudoEtiquette = ensemble_apprentissage_PseudoEtiquette.
    ↪shuffle(buffer_size=1024).batch(taille_lot_PseudoEtiquette)

# Dataset de validation
ensemble_validation = tf.data.Dataset.from_tensor_slices((x_validation, y_validation))
ensemble_validation = ensemble_validation.batch(taille_lot_etaquete) # Utilisation de la même taille de lot que pour les données étiquetées

```

Entraînement avec Pseudo-Étiquetage et Validation

Fonction d'entraînement avec pseudo-étiquetage

La fonction `etape_apprentissage` est conçue pour l'entraînement d'un modèle de réseau de neurones dans un scénario semi-supervisé avec pseudo-étiquetage. Voici un résumé de son fonctionnement :

Stratégie de Pseudo-Étiquetage - L'hyperparamètre `alpha` est ajusté linéairement entre les époques T1 et T2. - La fonction de perte combine la perte sur les données étiquetées et la perte pondérée sur les données pseudo-étiquetées.

Entraînement 1. Calcul des logits pour les données étiquetées et pseudo-étiquetées. 2. Ajustement de l'hyperparamètre `alpha` en fonction de l'époque. 3. Calcul de la perte totale en combinant les deux types de données. 4. Calcul des gradients de la perte par rapport aux poids du modèle. 5. Application des gradients à l'optimiseur pour mettre à jour les poids du modèle. Les limites sont limitées.

```
[21]: # Fonction d'entraînement avec pseudo-étiquetage
def etape_apprentissage(x_etaquete, y_etaquete, x_PseudoEtiquette, y_PseudoEtiquette, epoque):
    """
    Fonction d'entraînement avec pseudo-étiquetage.

    Parameters:
        x_etaquete (tf.Tensor): Données d'entraînement étiquetées.
        y_etaquete (tf.Tensor): Étiquettes correspondant aux données étiquetées.
        x_PseudoEtiquette (tf.Tensor): Données d'entraînement pseudo-étiquetées.
        y_PseudoEtiquette (tf.Tensor): Pseudo-étiquettes correspondant aux données pseudo-étiquetées.
        epoque (int): Numéro de l'époque actuelle.

    Returns:
        tf.Tensor: Valeur de perte pour l'époque actuelle.
    """

```

```

"""
alpha = 0.
T1 = 100
T2 = 600
af = 3.

with tf.GradientTape() as tape:
    logits_etiquete = modele_dropNN(x_etiquete, training=True)
    logits_PseudoEtiquette = modele_dropNN(x_PseudoEtiquette, training=True)

    if epoque > T1:
        alpha = ((epoque - T1) / (T2 - T1)) * af
        if epoque > T2:
            alpha = af

    # Calcul de la valeur de perte
    valeur_perte = fonction_perte_pseudo_etiquetage(y_etiquete,
    ↳logits_etiquete) + alpha * ↳
    fonction_perte_pseudo_etiquetage(y_PseudoEtiquette, logits_PseudoEtiquette)
    # Application des gradients à l'optimiseur
    gradients = tape.gradient(valeur_perte, modele_dropNN.trainable_weights)

    optimiseur_pseudo_etiquetage.apply_gradients(zip(gradients, modele_dropNN.
    ↳trainable_weights))
    return valeur_perte

```

Boucle d'Apprentissage

- **Époques** : Nous itérons sur un nombre prédéfini d'époques.
- **Métriques d'Entraînement** : Nous stockons la perte moyenne et la précision pour chaque époque pendant l'entraînement.
- **Métriques de Validation** : Nous calculons également ces métriques sur un ensemble de validation distinct.
- **Affichage des Résultats** : Les résultats, y compris la perte, la précision, la perte de validation et la précision de validation, sont affichés à chaque époque.

```
[22]: # Boucle d'apprentissage avec validation
epochs = 100

# Listes pour stocker la perte et la précision d'entraînement pour chaque époque
pertes_entrainement = []
precisions_entrainement = []

# Listes pour la validation
pertes_validation = []
precisions_validation = []
```

```

for epoque in range(epochs):
    print("\nDébut de l'époque %d" % (epoque,))
    temps_depart = time.time()
    etape = 1

    perte_epoque = 0.0
    predictions_correctes = 0
    total_echantillons = 0

        # Boucle d'entraînement
        for (lot_x_etiquete, lot_y_etiquete), (lot_x_pseudo_etiquete, u
        ↵lot_y_pseudo_etiquete) in zip(ensemble_apprentissage_etiquete, u
        ↵ensemble_apprentissage_PseudoEtiquette):
            # Appel de la fonction d'entraînement
            valeur_perte = etape_apprentissage(lot_x_etiquete, lot_y_etiquete, u
            ↵lot_x_pseudo_etiquete, lot_y_pseudo_etiquete, epoque)

            # Mise à jour des métriques
            perte_epoque += valeur_perte.numpy()

            logits_etiquete = modele_dropNN(lot_x_etiquete, training=False)
            predictions_etiquete = np.argmax(logits_etiquete, axis=1)
            predictions_correctes += np.sum(predictions_etiquete == lot_y_etiquete.
            ↵numpy())
            total_echantillons += len(lot_y_etiquete)

        # Calcul des métriques d'entraînement
        perte_moyenne = perte_epoque / etape
        precision = predictions_correctes / total_echantillons

        # Stockage des métriques d'entraînement
        pertes_entraînement.append(perte_moyenne)
        precisions_entraînement.append(precision)

    # Boucle de validation
    perte_validation = 0.0
    predictions_correctes_validation = 0
    total_echantillons_validation = 0

    for lot_x_validation, lot_y_validation in ensemble_validation:
        logits_validation = modele_dropNN(lot_x_validation, training=False)
        perte_validation += fonction_perte_pseudo_etiquetage(lot_y_validation, u
        ↵logits_validation).numpy()

        predictions_validation = np.argmax(logits_validation, axis=1)
        predictions_correctes_validation += np.sum(predictions_validation == u
        ↵lot_y_validation.numpy())

```

```

    total_echantillons_validation += len(lot_y_validation)

    # Calcul des métriques de validation
    perte_moyenne_validation = perte_validation / total_echantillons_validation
    precision_validation = predictions_correctes_validation / ↴
    ↵total_echantillons_validation

    # Stockage des métriques de validation
    pertes_validation.append(perte_moyenne_validation)
    precisions_validation.append(precision_validation)

    # Affichage des résultats de l'époque
    print("Époque %d : Temps écoulé : %.2fs, Perte : %.4f, Précision : %.4f, ↴
    ↵Perte Validation : %.4f, Précision Validation : %.4f" % (
        époque, time.time() - temps_depart, perte_moyenne, precision, ↴
        ↵perte_moyenne_validation, precision_validation))

```

Début de l'époque 0

Époque 0 : Temps écoulé : 1.06s, Perte : 0.4457, Précision : 1.0000, Perte Validation : 0.0338, Précision Validation : 0.7500

Début de l'époque 1

Époque 1 : Temps écoulé : 0.84s, Perte : 0.3123, Précision : 1.0000, Perte Validation : 0.0344, Précision Validation : 0.7400

Début de l'époque 2

Époque 2 : Temps écoulé : 0.91s, Perte : 0.4017, Précision : 1.0000, Perte Validation : 0.0345, Précision Validation : 0.7400

Début de l'époque 3

Époque 3 : Temps écoulé : 0.92s, Perte : 0.2715, Précision : 1.0000, Perte Validation : 0.0350, Précision Validation : 0.7500

Début de l'époque 4

Époque 4 : Temps écoulé : 0.88s, Perte : 0.1383, Précision : 1.0000, Perte Validation : 0.0361, Précision Validation : 0.7500

Début de l'époque 5

Époque 5 : Temps écoulé : 0.89s, Perte : 0.1392, Précision : 1.0000, Perte Validation : 0.0371, Précision Validation : 0.7600

Début de l'époque 6

Époque 6 : Temps écoulé : 0.90s, Perte : 0.1753, Précision : 1.0000, Perte Validation : 0.0370, Précision Validation : 0.7300

Début de l'époque 7

```
Époque 91 : Temps écoulé : 0.80s, Perte : 0.0042, Précision : 1.0000, Perte  
Validation : 0.0435, Précision Validation : 0.7600
```

Début de l'époque 92

```
Époque 92 : Temps écoulé : 0.85s, Perte : 0.0037, Précision : 1.0000, Perte  
Validation : 0.0443, Précision Validation : 0.7600
```

Début de l'époque 93

```
Époque 93 : Temps écoulé : 0.94s, Perte : 0.0050, Précision : 1.0000, Perte  
Validation : 0.0445, Précision Validation : 0.7600
```

Début de l'époque 94

```
Époque 94 : Temps écoulé : 0.92s, Perte : 0.0016, Précision : 1.0000, Perte  
Validation : 0.0446, Précision Validation : 0.7600
```

Début de l'époque 95

```
Époque 95 : Temps écoulé : 0.90s, Perte : 0.0030, Précision : 1.0000, Perte  
Validation : 0.0442, Précision Validation : 0.7600
```

Début de l'époque 96

```
Époque 96 : Temps écoulé : 0.90s, Perte : 0.0052, Précision : 1.0000, Perte  
Validation : 0.0440, Précision Validation : 0.7600
```

Début de l'époque 97

```
Époque 97 : Temps écoulé : 0.88s, Perte : 0.0534, Précision : 1.0000, Perte  
Validation : 0.0443, Précision Validation : 0.7600
```

Début de l'époque 98

```
Époque 98 : Temps écoulé : 0.86s, Perte : 0.0038, Précision : 1.0000, Perte  
Validation : 0.0471, Précision Validation : 0.7700
```

Début de l'époque 99

```
Époque 99 : Temps écoulé : 0.85s, Perte : 0.0041, Précision : 1.0000, Perte  
Validation : 0.0496, Précision Validation : 0.7700
```

Visualisation de l'historique du modèle DropNN avec PseudoLabeling

```
[23]: # Création de la figure  
plt.figure(figsize=(16, 6))  
  
# Courbe de perte  
plt.subplot(1, 2, 1)  
sns.lineplot(x=range(1, epochs + 1), y=pertes_entrainement, label='Entrainement', color='blue')  
sns.lineplot(x=range(1, epochs + 1), y=pertes_validation, label='Validation', color='orange')  
plt.title('Évolution de la Perte')  
plt.xlabel('Époque')
```

```

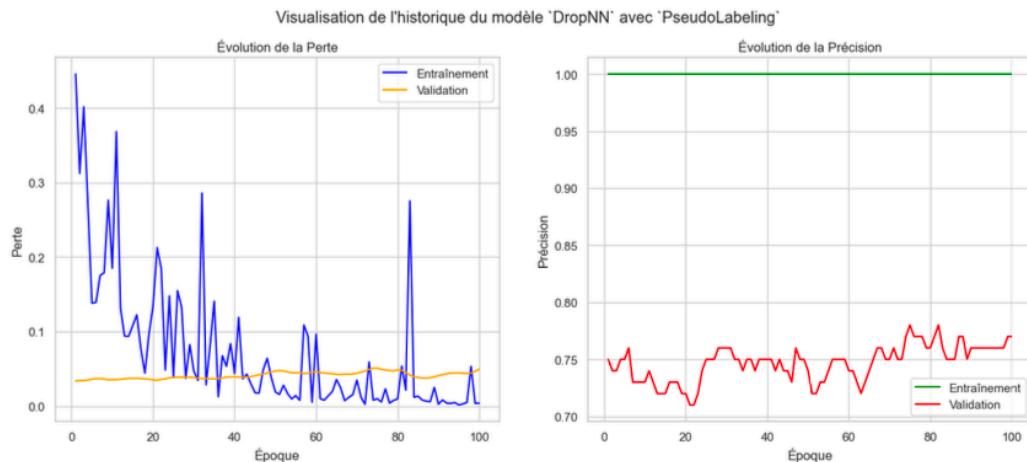
plt.ylabel('Perte')
plt.legend()
plt.grid(True)

# Courbe de précision
plt.subplot(1, 2, 2)
sns.lineplot(x=range(1, epochs + 1), y=precisions_entrainement, u
             ↪label='Entrainement', color='green')
sns.lineplot(x=range(1, epochs + 1), y=precisions_validation, u
             ↪label='Validation', color='red')
plt.title('Évolution de la Précision')
plt.xlabel('Époque')
plt.ylabel('Précision')
plt.legend()
plt.grid(True)

# Ajout du titre à la figure
plt.suptitle("Visualisation de l'historique du modèle `DropNN` avec"
             ↪`PseudoLabeling`")

# Affichage de la figure
plt.show()

```



Évaluation et Visualisation des Métriques de Classification

```

[24]: # Prédiction sur l'ensemble de test
y_pred = np.argmax(modele_dropNN.predict(x_test), axis=1)

# Calcul de la matrice de confusion
matrice_confusion = confusion_matrix(y_test, y_pred)

```

```

# Tracer la matrice de confusion avec un style amélioré
plt.figure(figsize=(5, 4))
sns.heatmap(matrice_confusion, annot=True, fmt="d", cmap="viridis", cbar=False, u
↳square=True,
            xticklabels=np.arange(10), yticklabels=np.arange(10), linewidths=.
↳5, linecolor='gray')
plt.xlabel('Prédit', fontsize=12, fontweight='bold')
plt.ylabel('Réel', fontsize=12, fontweight='bold')
plt.title('Matrice de Confusion', fontsize=14, fontweight='bold')
plt.xticks(fontsize=10)
plt.yticks(fontsize=10)
plt.show()

# Rapport de classification
rapport_classification = classification_report(y_test, y_pred, u
↳target_names=[str(i) for i in range(10)])
print("Rapport de Classification:\n", rapport_classification)

# Affichage des résultats d'évaluation
precision = accuracy_score(y_test, y_pred)

print("\nRésultats de l'Évaluation sur l'Ensemble de Test :")
print(f">> Précision : {precision}")

```

313/313 [=====] - 2s 6ms/step

Matrice de Confusion											
Réel	0	1	2	3	4	5	6	7	8	9	
Prédit	0	947	1	7	0	2	3	13	4	3	0
0	0	1050	13	5	2	1	4	0	60	0	
1	18	13	834	28	11	1	11	14	89	13	
2	10	1	17	818	1	91	9	13	29	21	
3	3	4	4	1	709	0	11	5	26	219	
4	34	11	5	70	18	494	18	7	162	73	
5	29	7	59	6	72	5	755	1	16	8	
6	2	24	63	10	21	2	1	807	10	88	
7	26	15	10	40	22	22	9	8	803	19	
8	10	6	6	13	93	7	1	21	19	833	
9											

Rapport de Classification:

	precision	recall	f1-score	support
0	0.88	0.97	0.92	980
1	0.93	0.93	0.93	1135
2	0.82	0.81	0.81	1032
3	0.83	0.81	0.82	1010
4	0.75	0.72	0.73	982
5	0.79	0.55	0.65	892
6	0.91	0.79	0.84	958
7	0.92	0.79	0.85	1028
8	0.66	0.82	0.73	974
9	0.65	0.83	0.73	1009
accuracy			0.81	10000
macro avg	0.81	0.80	0.80	10000
weighted avg	0.81	0.81	0.80	10000

Résultats de l'Évaluation sur l'Ensemble de Test :
 >> Précision : 0.805