UBFC
UNIVERSITÉ
BOURGOGNE FRANCHE-COMTÉ

ESIREM
Informatique/Electronique — 4A

Systèmes Intelligents - Travaux IA

Python et perceptron

Auteur :
GONCALVES Hugo

ESIREM
ÉCOLE SUPÉRIEURE D'INGÉNIEURS
NUMÉRIQUE ET MATÉRIAUX
École associée Polytech

2023-2024

Sommaire

Table des Figures

Chapitre 1

Introduction

Lien Collab : //colab.research.google.com/drive/1P16I3Up_22Dly1xLXtxpnYdgOTKX1nzH?usp=sharing

L'intelligence artificielle est devenue une composante essentielle dans divers domaines d'application, tel que les diagnostics dans le secteur de la santé. Ce rapport se concentre sur l'utilisation de Python, un langage de programmation puissant et polyvalent, et très utilisé pour développer des modèles IA. Notre étude se divise en deux exercices principaux : la création d'un perceptron pour simuler une porte logique OR et le développement d'un réseau de neurones profonds en utilisant la bibliothèque PyTorch pour le diagnostic du cancer du sein.

Le premier exercice explore les fondements de l'apprentissage machine à travers la conception et l'implémentation d'un perceptron simple. Cette approche permet de comprendre les mécanismes de base des neurones artificiels et leur capacité à effectuer des tâches de classification simples. Nous démontrerons comment un perceptron peut être utilisé pour modéliser une porte logique OR, fournissant ainsi une introduction pratique aux concepts fondamentaux de l'apprentissage supervisé vu lors des cours magistraux.

Dans le deuxième exercice, nous abordons une application plus complexe de l'IA en santé. Utilisant PyTorch, nous construirons et entraînerons un DNN pour identifier la présence de cancer du sein à partir de données cliniques. Ce projet illustrera l'importance de l'IA dans l'amélioration du diagnostic et du traitement des maladies.

Chapitre 2

Introduction à Python

2.1 Python et son importance en IA

L'adoption de Python dans le domaine de l'intelligence artificielle est principalement due à sa simplicité et à sa large collection de bibliothèques. Ces outils permettent nous permettent de construire, expérimenter et déployer des modèles d'IA rapidement et efficacement. En outre, Python est largement utilisé dans l'enseignement et la recherche, ce qui en fait un choix privilégié pour les étudiants et les professionnels souhaitant se lancer dans le domaine de l'IA.

2.2 Concepts de base

Exercice 1 : Ajouter un point d'exclamation

Dans le cadre de cet exercice, nous avons développé une fonction nommée `add_exclamation(wor` Son but est d'ajouter un point d'exclamation à la fin d'une chaîne de caractères donnée. Pour réaliser cela, nous utilisons simplement la concaténation du mot initial et d'un point d'exclamation à la fin.

Exercice 2 : Somme cumulée

Dans le cadre de l'exercice 2, nous avons élaboré la fonction `cumulative_sum`, conçue pour calculer la somme cumulée progressive d'un ensemble de nombres. Cette fonction emploie une boucle "for" pour itérer sur chaque élément de la liste, en s'appuyant sur une variable "total" pour tenir le compte de la somme cumulée. À chaque passage dans la boucle, l'élément courant est additionné au total cumulé, et la valeur de cet élément dans la liste est mise à jour avec la somme actuelle.

Exercice 3 : Calcul du factorielle

La fonction `factoriel`, conçue dans notre travail, utilise la récursivité pour le calcul de la factorielle d'un entier. Cette méthode repose sur la propriété fondamentale de la factorielle. Pour un entier donné, si celui-ci est zéro, la fonction renvoie directement 1, car la factorielle de zéro est universellement acceptée comme étant 1. Pour tous les autres entiers positifs, la fonction s'engage dans une série d'opérations récursives, multipliant le nombre par la factorielle de son prédécesseur. Cette stratégie de récursion décompose le calcul en une suite d'opérations plus gérables, permettant ainsi de résoudre le calcul de la factorielle de manière efficace.

2.2.1 POO

Exercice 1 : Classe Personne

Dans cet exercice, nous avons mis en œuvre une classe nommée `Personne`, destinée à modéliser les informations essentielles d'un individu. Cette classe est caractérisée par deux attributs principaux : le nom et l'âge de la personne.

La structure de la classe `Personne` comprend :

Attributs : `nom` : Une chaîne de caractères représentant le nom de l'individu. `age` : Un entier indiquant l'âge de la personne.

Le constructeur de la classe, `__init__`, est défini pour initialiser ces attributs. Il prend deux paramètres : `nom` et `age`, et les assigne aux attributs correspondants de l'instance de la classe. Cette approche fournit un moyen structuré et efficace de créer des objets représentant des individus.

Exercice 2 : Affichage des informations

Dans cette partie nous avons conçue une méthode pour retourner une chaîne de caractères contenant les informations de la personne, à savoir son nom et son âge.

Chapitre 3

Notre premier Perceptron

Pour cet exercice, notre tâche consistait à développer un perceptron en utilisant le langage Python. Notre but spécifique était de concevoir un perceptron avec trois entrées et une sortie, tout en intégrant un terme de biais dans le modèle.

3.0.1 Schéma du Perceptron

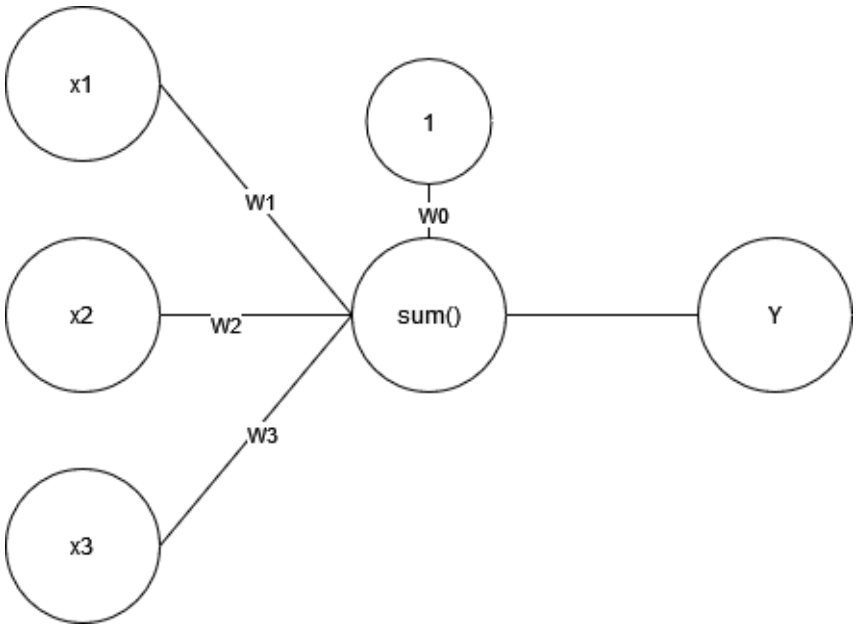

Figure 3.1 – Schéma Perceptron

Nous avons élaboré un schéma pour visualiser la structure de notre perceptron. Il se compose de trois entrées, notées x_1, x_2, et x_3, et d'un unique neurone qui calcule la somme pondérée $x_1w_1 + x_2w_2 + x_3w_3$. Le résultat de cette somme pondérée est ensuite transmis à la sortie du neurone, désignée par y.
De plus, nous avons intégré un biais avec une valeur fixée à -1, qui est ajouté à cette somme. L'inclusion de ce biais est cruciale car elle permet au réseau de neurones d'atteindre un apprentissage optimal, en lui donnant la capacité de minimiser l'erreur jusqu'à zéro.

3.0.2 Données X et Y

Nous avons ensuite créé un ensemble de données pour entraîner notre perceptron. Les données d'entrée, désignées par la variable `x`, consistent en une série de vecteurs

à trois composants. Chaque vecteur représente une combinaison unique des valeurs binaires 0 et 1, couvrant ainsi toutes les permutations possibles de trois variables binaires. La structure de `x` est définie comme :

```
x = [[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1],
     [1, 0, 0], [1, 0, 1], [1, 1, 0], [1, 1, 1]]
```

Parallèlement, la variable `y` représente les sorties correspondantes pour chaque vecteur d'entrée dans `x`. Ces sorties suivent la logique d'une porte OR, où la présence d'au moins un '1' dans un vecteur d'entrée entraîne une sortie de '1'. La seule exception est le vecteur [0, 0, 0], qui produit une sortie de '0'. Le vecteur `y` est donc défini comme :

```
y = [0, 1, 1, 1, 1, 1, 1, 1]
```

3.0.3 Création de notre Perceptron

Nous avons développé une classe nommée `Perceptron` en Python, qui sert de base pour un modèle de perceptron simple. Cette classe se distingue par ses composants clés, décrits ci-dessous :

— **Attributs** :
  — `w` : Il s'agit d'une liste qui contient les poids du perceptron, éléments essentiels pour le processus d'apprentissage.
— **Méthodes** :
  — `train(inputs, outputs, epochsNumber, lr)` : Cette méthode est dédiée à l'entraînement du perceptron, ajustant les poids en fonction des entrées et des sorties.
  — `predict(inputs)` : Méthode utilisée pour prédire la sortie du perceptron en fonction des entrées fournies.

Le constructeur `__init__` de la classe initialise les poids (`w`) en assignant des valeurs aléatoires. Le paramètre `input_size` indique le nombre d'entrées au perceptron, avec un poids supplémentaire ajouté pour le biais.

La méthode `predict` joue un rôle crucial en effectuant la somme pondérée des entrées et du biais. Le résultat de cette somme détermine la sortie du perceptron : si la somme est supérieure ou égale à 0.5, la sortie est 1, sinon elle est 0.

Ensemble, ces éléments donnent à notre classe `Perceptron` la capacité de simuler le fonctionnement d'un neurone artificiel.

3.0.4 Entraînement du Perceptron

Notre méthode `train` de notre classe `Perceptron` est cruciale pour l'apprentissage automatique de notre perceptron. Il est donc nécessaire de la définir le plus soigneusement possible :

- **Paramètres de la Méthode** :
— `inputs` : Les vecteurs d'entrée qui alimentent le perceptron.
— `outputs` : Les sorties attendues, servant de référence pour l'ajustement des poids.
— `epochsNumber` : Le nombre total d'itérations sur l'ensemble de données.
— `lr` (Learning Rate) : Le taux d'apprentissage. Il permet d'accélérer ou ralentir la convergence de notre apprentissage.
- **Algorithme d'Entraînement** :

1. *Initialisation* : Une variable d'erreur est réinitialisé à chaque epoch pour suivre le nombre d'erreur sur chaque itération.
2. *Itération sur les Données* : Pour chaque paire d'entrée et de sortie attendue, l'algorithme effectue les opérations suivantes :
   — *Prédiction* : Calcul de la sortie prévue en utilisant la méthode `predict`.
   — *Évaluation de l'Erreur* : Détermination de l'erreur comme la différence entre la sortie attendue et la sortie prédite.
   — *Mise à jour des Poids selon la Règle de Widrow-Hoff* : Ajustement des poids en suivant la règle de Widrow-Hoff. Chaque poids est ajusté en fonction de l'erreur, du taux d'apprentissage et de la valeur d'entrée correspondante.
3. *Vérification de la Convergence* : Si aucune erreur n'est détectée durant un epoch, le perceptron est considéré comme correctement entraîné et la méthode s'arrête.

Cette approche, permet une convergence efficace du perceptron vers une solution optimale, en minimisant l'erreur de prédiction à chaque itération.

3.0.5 Observations autour des hyperparamètres

1. Que se passe-t-il quand il y a peu d'epochs ?

Quand le nombre d'epochs est faible, le perceptron ne dispose pas de suffisamment d'itérations pour apprendre à partir des données d'entraînement. Cela conduit à un sous-entraînement, où le perceptron n'atteint pas sa capacité optimale de prédiction et montre ainsi une mauvaise généralisation sur des nouvelles données.

2. Que se passe-t-il quand le learning rate est trop grand ou trop petit ?

— *Trop Grand :* Un learning rate élevé entraîne une convergence rapide mais instable. Le perceptron peut osciller ou même dépasser le minimum optimal, ce qui mène à des résultats imprévisibles et souvent pas optimaux.
— *Trop Petit :* Inversement, un learning rate faible entraîne une convergence lente, augmentant le risque que le perceptron reste ”piégé” dans des minima locaux et nécessite un nombre d'epochs élevé pour une meilleure convergence.

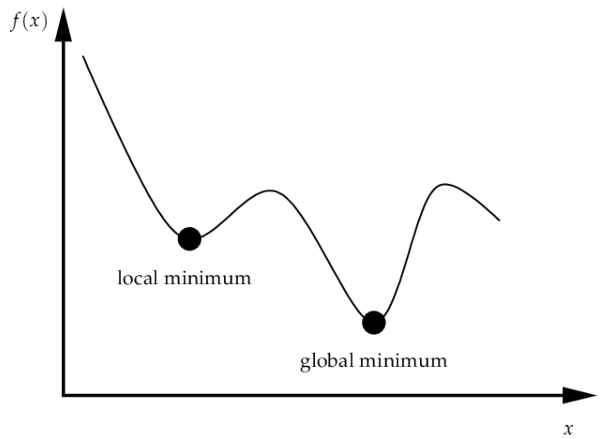

Figure 3.2 – Minimum local

3. Quel est le rôle des hyperparamètres dans un apprentissage ?

Les hyperparamètres jouent un rôle crucial dans la performance de notre perceptron. Ils influencent la vitesse d'apprentissage, la capacité de notre perceptron à converger vers une solution optimale, et sa performance générale à prédire. Un réglage appropriés de ces hyperparamètres sont très importants pour obtenir des résultats optimums.

3.0.6 Test de notre perceptron

Nous avons ensuite créé des données de test, pour évaluer la performance de notre perceptron. Elles permettent de vérifier l'efficacité de notre perceptron dans des conditions "réelles", c'est-à-dire sur des données qu'il n'a jamais vues auparavant.

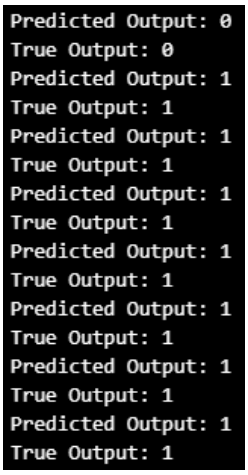

Figure 3.3 – Résultat de notre perceptron

Nous l'avons donc testé sur ce dataset de test. Les résultats ont été bons puisque notre perceptron a atteint une accuracy de 100%. Notre perceptron a donc acquis la capacité de généraliser ses connaissances sur de nouvelles données.

Chapitre 4

Introduction à Pandas

Nous allons maintenant aborder Pandas, une bibliothèque Python conçue pour la manipulation de données, adaptée à une vaste gamme d'applications. Sa popularité dans les domaines du Data Management et de l'Intelligence Artificielle s'explique par sa capacité à faciliter et accélérer le traitement des données. Jusqu'à présent, notre perceptron était conçu pour fonctionner avec une couche d'entrée de trois unités seulement. Notre objectif est désormais de le rendre plus polyvalent et adaptable à différentes tailles d'entrées, exploitant ainsi la flexibilité offerte par Pandas.

4.0.1 Principes de bases

Au début de notre exploration de Pandas, nous avons acquis une compréhension essentielle des outils et méthodes clés de cette bibliothèque. Notre apprentissage s'est concentré sur des aspects fondamentaux tels que la manipulation efficace des DataFrames, l'utilisation stratégique de la fonction `loc` pour accéder aux données, la compréhension des structures de données via `shape`, l'emploi des `iterrows` pour itérer sur les lignes des DataFrames, ainsi que `df.values` qui nous retourne une représentation NumPy de notre DataFrame. Ces compétences constituent la base de la gestion avancée des données.

Avec ces connaissances nouvellement acquises, nous sommes désormais en mesure d'intégrer ces méthodes de manipulation de données dans notre modèle de perceptron précédemment développé. Cette intégration vise à améliorer la capacité du perceptron à gérer des ensembles de données de différentes tailles et formats.

4.0.2 Implémentation de Pandas dans notre Perceptron

Nous avons donc actualisé notre classe `Perceptron` afin d'intégrer la bibliothèque Pandas. Nous avons donc apporter les modifications suivantes :

Initialisation des Poids

Dans le constructeur (`__init__`), les poids (`self.w`) sont initialisés pour correspondre à la taille des entrées. La taille des entrées est basée sur le nombre de colonnes dans le DataFrame Pandas, déterminée par `len(x.columns)`.

Méthode d'Entraînement

La méthode `train` a été adaptée pour travailler avec un DataFrame Pandas en tant qu'entrée :

— Elle utilise `iterrows()` pour parcourir chaque ligne du DataFrame, traitant les données d'entrée ligne par ligne.
— À chaque itération, la sortie prédite est comparée à la sortie attendue, accessible via `outputs.iloc[i]`.
— Les poids sont ajustés en fonction de l'erreur entre la prédiction et la sortie attendue, avec le taux d'apprentissage (`lr`).

Méthode de Prédiction

La méthode `predict` reste similaire à la version originale.

Test sur différents DataFrames

Nous avons évalué notre perceptron sur une porte logique ET, et les résultats obtenus sont comparables à ceux de la porte OU. De plus, nous avons étendu notre test à un scénario avec quatre entrées, et grâce à la flexibilité de notre perceptron, cette configuration s'est exécutée avec succès avec une accruacy de 100%.

Bonus : Afficheur 7 segments

Nous allons maintenant développer un afficheur à 7 segments en utilisant notre modèle de perceptron. Il est cependant crucial de souligner que les perceptrons simples, avec leur nature linéaire, sont souvent limités dans leur capacité à résoudre des problèmes complexes, notamment ceux qui ne sont pas linéairement séparables. Un afficheur à 7 segments, conçu pour représenter les chiffres de 0 à 9, s'inscrit dans cette catégorie en raison de sa complexité intrinsèque.

Un perceptron simple ne dispose que de deux valeurs de sorties possibles, ce qui s'avère insuffisant pour représenter directement les états binaires des 7 segments individuels d'un affichage numérique. Pour surmonter cette limitation, notre approche consiste à entraîner sept perceptrons distincts, chacun se concentrant sur la prédiction de l'état (allumé ou éteint) d'un segment spécifique de l'afficheur. En d'autres termes, chaque perceptron apprendra à identifier l'état d'un segment en particulier, basé sur une entrée binaire donnée.

Cette méthode nous permet de contourner la contrainte de linéarité du perceptron simple. En entraînant chaque perceptron séparément pour un segment spécifique, nous pourrons assembler leurs prédictions pour obtenir l'affichage complet à 7 segments.

Grâce à une aide extérieur et de la bibliothèque Matplotlib, nous avons la capacité de concevoir et de dessiner un afficheur à 7 segments. Cette étape visuelle est cruciale pour démontrer de manière concrète les capacités de notre modèle de perceptron. Notre prochaine tâche consiste à intégrer cette représentation graphique avec la logique de prédiction de nos perceptrons.

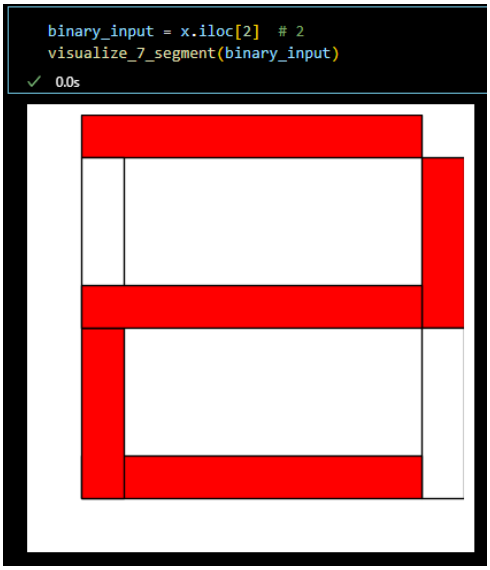

Figure 4.1 – Résultat 7 segments

Une fois cette intégration achevée, les résultats obtenus sont satisfaisants. La figure précédente illustre ces résultats.

Conclusion sur le perceptron

Pour conclure, notre étude a mis en évidence les limitations intrinsèques d'un perceptron à neurone unique lorsqu'il s'agit de résoudre des problèmes non linéairement séparables. Bien que le perceptron soit efficace et suffisant pour traiter des tâches simples, sa structure linéaire limite sa capacité à aborder des problématiques plus complexes. Cette limitation devient particulièrement évidente dans des scénarios où les données ne peuvent pas être séparées par une seule frontière de décision linéaire.

Il devient nécessaire de se tourner vers des modèles plus sophistiqués, tels que les réseaux de neurones profonds (DNN) lorsque le problème s'avère plus complexe.

Chapitre 5

Diagnostic du Cancer du Sein avec un DNN en PyTorch

Dans cette nouvelle section de notre étude, nous allons développer une application capable de distinguer entre un cancer bénin et malin. Cette tâche implique l'analyse de multiples variables pour effectuer un diagnostic. Pour cela, nous ferons appel à la librairie Pytorch, un outil couramment utilisé dans le domaine de l'apprentissage.

Contrairement à nos travaux précédents où nous utilisions un perceptron simple, ici, nous allons mettre en œuvre un réseau de neurones profond. Cette évolution vers un modèle plus complexe est nécessaire pour la complexité des données liées au diagnostic du cancer.

5.1 Présentation du dataset Breast Cancer Wisconsin

Le dataset est constitué de caractéristiques calculées à partir d'images numérisées de biopsies par aiguille fine de masses mammaires. Ces caractéristiques décrivent les propriétés des noyaux cellulaires présents dans les images. Les données proviennent de l'étude de Bennett et Mangasarian en 1992 et sont également disponibles dans le dépôt UCI Machine Learning Repository.

Informations sur les Attributs :

Le dataset contient les attributs suivants :

1. Numéro d'identification
2. Diagnostic (M = malin, B = bénin)
3. à 32) Dix caractéristiques réelles pour chaque noyau cellulaire :
   — Rayon (moyenne des distances du centre aux points du périmètre)
   — Texture (écart-type des valeurs de gris)
   — Périmètre
   — Aire
   — Lisséité
   — Compacité
   — Concavité
   — Points concaves
   — Symétrie
   — Dimension fractale

Détails Supplémentaires :

— Distribution des classes : 357 cas bénins, 212 cas malins.

Sources : Kaggle

5.2 Traitement du Dataset

5.2.1 Importation et Exploration du Dataset

La première étape de notre projet implique l'importation du dataset de cancer du sein, qui est stocké sous forme de fichier .csv. Pour cela, nous utilisons la fonction `read_csv` de la bibliothèque Pandas. Cette fonction nous permet de charger facilement les données dans un format exploitable.

Une fois les données importées, la méthode `head` de Pandas peut être utilisée pour visualiser les premières lignes du dataset.

Cette visualisation nous offre un aperçu des caractéristiques disponibles dans le dataset, nous aidant ainsi à identifier les informations les plus pertinentes pour l'entraînement de notre modèle. Cette étape d'exploration des données est importante pour déterminer quelles variables seront utilisées dans notre modèle.

5.2.2 Pré-traitement des Données

Suppression et Conversion

Avant de procéder à l'entraînement, une préparation des données est nécessaire. Dans cette étape, nous filtrons et excluons les informations non pertinentes qui ne contribuent pas à la prédiction du diagnostic. Par exemple, nous avons identifié que les colonnes *"id"* et *"Unnamed : 32"* dans notre dataset ne sont pas utiles pour l'apprentissage. Leur suppression permet de réduire la complexité du modèle et d'éviter le bruit inutile dans les données.

```
df.drop(['id', 'Unnamed: 32'], axis=1, inplace=True)
```

De plus, il est important de noter que la colonne de diagnostic dans le dataset est exprimée sous forme de chaînes de caractères (*string*), indiquant *"maligne"* ou *"bénigne"*. Or, notre modèle nécessite une sortie binaire pour effectuer des prédictions précises. Par conséquent, ces valeurs doivent être converties en un format binaire : 0 pour *"bénigne"* et 1 pour *"maligne"*.

```
df['diagnosis'] = df['diagnosis'].map({'M':1, 'B':0})
```

Ce processus de filtrage et de transformation des données est une étape importante pour préparer le dataset à un apprentissage efficace.

Impact des échelles sur l'apprentissage

Lors de l'entraînement d'un modèle, des différences significatives dans les échelles de valeurs entre les différentes caractéristiques peuvent entraîner des problèmes comme :

1. **Convergence plus lente.**
2. **Sensibilité aux initialisations de poids.**
3. **Déséquilibre dans l'influence des caractéristiques.**

Pour palier à ces problèmes, nous allons devoir standardiser les données, en alignant toutes les caractéristiques sur une même échelle.

Standardisation

Pour réaliser cette standardisation, nous utilisons les fonctionnalités de `sklearn.preprocessing`, une composante de la bibliothèque sklearn.

```
from sklearn import preprocessing
X_standardized = preprocessing.scale(X)
```

Cette approche de standardisation garantit que notre modèle reçoit des données homogènes.

Ensembles d'entraînement et de test

La phase suivante consiste à diviser notre dataset en deux parties distinctes : un ensemble destiné à l'entraînement de notre modèle et un autre réservé aux tests. Nous allons allouer 80% des données pour l'entraînement et 20% pour les tests.

Pour cela, nous utilisons la fonction `train_test_split()` de la bibliothèque sklearn :

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test =
train_test_split(X[:,1:], y[:,0], test_size=0.20, , random_state=42)
```

Cette méthode garantit que les données sont réparties de manière aléatoire et représentative entre les deux ensembles, permettant ainsi à notre modèle d'apprendre et de se généraliser efficacement.

5.3 Création de notre Neural Network

Avant d'entamer la création de notre réseau de neurones (NN), il est essentiel de définir et de comprendre plusieurs paramètres liés à sa conception.

Notre NN sera conçu pour traiter 30 valeurs d'entrée. Chacune de ces entrées correspond à une caractéristique différente du dataset de cancer du sein.

Un aspect crucial de notre NN est l'utilisation d'une fonction d'activation Ssigmoid. Cette fonction est choisie pour sa capacité à transformer efficacement l'entrée du neurone en un résultat compris entre 0 et 1. Cette caractéristique est adaptée à notre cas :

- **Sortie binaire :** Comme notre objectif est de classifier les tumeurs en bénignes (0) ou malignes (1), la fonction sigmoid nous permet de modéliser cette sortie binaire.
- **Probabilités :** Les valeurs de sortie du sigmoid peuvent être interprétées comme des probabilités, offrant une mesure de la probabilité que la tumeur soit maligne.

Nous présentons ici la définition d'un modèle de réseau de neurones profond (DNN) utilisé pour la classification binaire. Ce modèle est destiné à diagnostiquer le cancer du sein à partir d'un ensemble de caractéristiques cliniques.

5.3.1 Architecture du Modèle

La classe `DNN` est une sous-classe de `nn.Module`, qui est la classe de base pour tous les modules de réseau neuronal dans PyTorch. L'initialisation du modèle se fait comme suit :

```
class DNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.archiNN = nn.Sequential(
            nn.Linear(input_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, output_size),
            nn.Sigmoid()
        )
    def forward(self, x):
        return self.archiNN(x)
```

La première couche linéaire, `nn.Linear(input_size, hidden_size)`, transforme l'entrée de dimension `input_size` en une représentation cachée de dimension `hidden_size`. Après la couche linéaire, nous appliquons une fonction d'activation ReLU pour introduire de la non-linéarité dans le modèle. La dernière couche linéaire, `nn.Linear(hidden_size, output_size)`, mène à la couche de sortie qui a une seule unité (puisque `output_size` est 1). La fonction d'activation Sigmoid a été vu précédemment.

5.3.2 Tensor et DataLoader

Les ensembles de données d'entraînement et de test sont convertis en tenseurs PyTorch, ce qui est nécessaire pour les traiter avec le framework PyTorch.

Les tenseurs sont ensuite ”emballés” dans des objets `TensorDataset`, qui sont ensuite passés à des `DataLoaders`. Ces DataLoaders faciliteront l'itération sur les ensembles de données pendant la phase de train et de tests.

La fonction de perte `BCELoss` est appropriée pour notre problème de classification binaire. Elle mesure la performance du modèle dont la sortie est une probabilité entre 0 et 1. L'optimiseur SGD (Descente de Gradient Stochastique) avec un taux d'apprentissage de `1e-1` est choisi pour l'ajustement des paramètres du modèle.

Nous avons défini la taille de batch à 16, ce qui signifie que notre modèle traitera 16 échantillons à la fois lors de l'entraînement. Cette taille de batch est un compromis entre la précision de l'estimation du gradient et la vitesse de calcul.

5.3.3 Fonction d'entraînement

La fonction `train` est conçue pour l'entraînement de notre modèle.

Architecture

— **Epochs :** Chaque *epoch* représente une itération sur l'ensemble du dataset.
— **Optimizer :** À chaque batch, l'optimiseur ajuste les paramètres du modèle en fonction du gradient de la fonction loss. L'initialisation de l'optimiseur à zéro (`optimizer.zero_grad()`) est là pour éviter l'accumulation de gradients.

— **Loss :** La fonction loss mesure l'erreur entre les prédictions du modèle et les valeurs réelles. Son calcul et sa "backpropagation" (`loss.backward()`) sont très importants pour l'ajustement des paramètres de notre modèle.

Metrics

— **Accuracy :** L'accuracy se calcule avec le rapport entre le nombre de prédictions correctes et le nombre total de prédictions. Elle fournit une mesure globale de la performance de notre modèle.
— **Precision :** Le calcul de la précision, en considérant les vrais positifs (VP) et les faux positifs (FP), est important pour évaluer la capacité de notre modèle à ne pas classer un patient négatif comme positif.
— **Losses :** L'enregistrement des pertes à chaque epoch aide à visualiser l'évolution de l'apprentissage du modèle, permettant d'identifier les potentiels problèmes de sur-apprentissage ou de sous-apprentissage.

5.3.4 Fonction de test

Il est maintenant pertinent d'examiner la fonction de test. La fonction `test` est conçue pour évaluer la performance du modèle sur l'ensemble de test, sans effectuer d'ajustement des paramètres du modèle :

— **Absence de Gradient :** `torch.no_grad()` est utilisé pour désactiver le calcul des gradients. Contrairement à l'entraînement, nous n'avons pas besoin d'optimiser les paramètres du modèle lors des tests.
— **Métriques de Performance :** Comme dans la fonction d'entraînement, les mesures de précision et de perte sont calculées. Cependant, ces mesures sont utilisées uniquement pour évaluer et non pour ajuster le modèle.

5.4 Résultats obtenus

Après avoir conduit la phase de train et de tests de notre modèle , nous avons observé les résultats suivants :

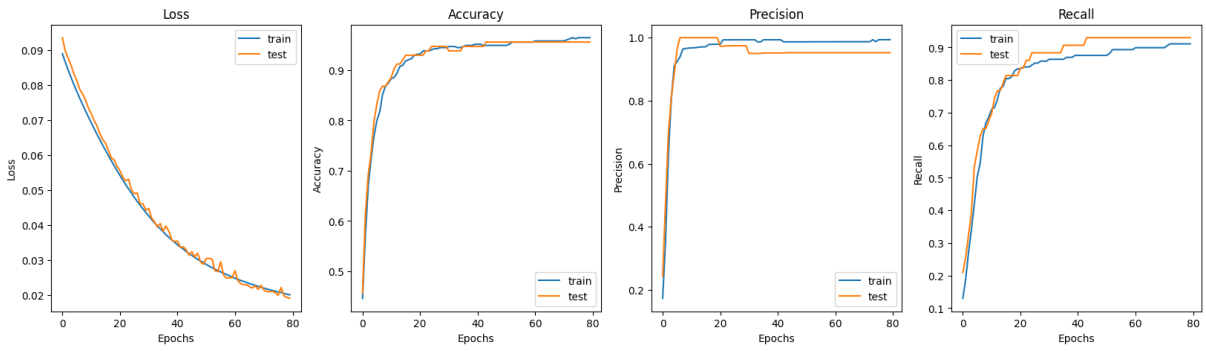

Figure 5.1 – Metrics du DNN

Loss

— Durant les phases d'entraînement et de test, nous avons observé une diminution similaire de la perte, stabilisée autour de 0.02. Cette baisse cohérente montre que le modèle s'adapte bien tant aux données d'entraînement qu'aux données de test, indiquant une bonne généralisation.

Accuracy

— L'accuracy, atteignant environ 95%, a montré des valeurs comparables dans les deux phases. Cette performance uniforme suggère que le modèle ne souffre pas d'un surapprentissage significatif et maintient sa précision sur les données nouvelles et inconnues.

Precision

— La précision, avoisinant les 98% pendant l'entraînement, a également été observée dans la phase de test. Cela démontre que le modèle est efficace pour identifier les cas positifs, tant dans un environnement contrôlé que dans des conditions réelles.

Recall

— Le recall, a atteint environ 90% dans les deux phases, indiquant que le modèle est capable de détecter la majorité des cas positifs réels. Un recall élevé est particulièrement important dans des contextes cliniques où manquer un diagnostic positif peut avoir des conséquences graves.

Chapitre 6

Conclusion Générale

Ce travail pratique a couvert un large éventail de concepts et d'applications en intelligence artificielle, depuis les bases de Python et la programmation orientée objet, jusqu'à la création et l'évaluation de perceptrons et de réseaux de neurones profonds (DNN) en utilisant PyTorch.

Ce travail pratique a non seulement renforcé notre compréhension théorique de l'intelligence artificielle mais a également fourni une expérience pratique significative dans la mise en œuvre de ses concepts. La capacité à naviguer entre les aspects théoriques et pratiques de l'IA est essentielle pour toute future application dans ce domaine. Les résultats encourageants obtenus dans ce projet ouvrent la voie à des développements plus approfondis.