le cnam
école d'ingénieur·e·s

Réseaux de neurones

 github/projet

Réalisation : Alexis GUILLTON & Clément MARIE--BRISSON

Superviseur : Philippe CARRRE

2023-03-17

Table des matières

1. Développement d'un perceptron

Un perceptron est un modèle de réseau neuronal artificiel simple inventé par Frank Rosenblatt en 1957. L'objectif de ce modèle est de générer une sortie binaire pour une entrée donnée en utilisant un ensemble de poids synaptiques réglés à l'aide d'un algorithme d'apprentissage. Nous allons implémenter un perceptron simple qui fournit une fonction de sortie qui est activée en fonction de la fonction d'activation choisie.

1.1 Mise en place d'un perceptron simple

Dans la première partie du code, nous avons créé une fonction nommée "`perceptron_simple`" qui prend en entrée un vecteur d'entrée x, un vecteur de poids synaptiques w et un indicateur de fonction d'activation active. Cette fonction évalue la sortie d'un perceptron simple pour une entrée élément de R2.

Nous avons commencé par insérer une valeur de biais (1) à la première position de x en utilisant la fonction d'insertion numpy. Cette valeur est utilisée pour calculer le seuil des neurones.

Ensuite, nous avons initialisé la variable y à 0. Il s'agit de la somme pondérée de toutes les entrées multipliées par les poids synaptiques correspondants.

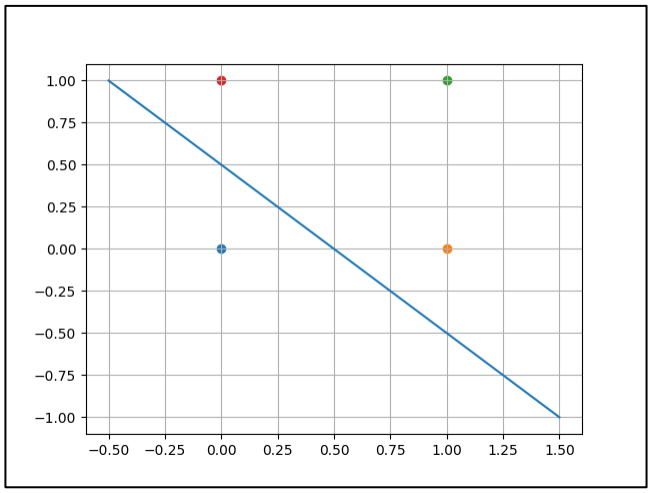
La boucle for itère sur chaque élément de w et multiplie chaque élément de x par le poids synaptique correspondant pour calculer une somme pondérée des entrées.

Si la fonction d'activation sélectionnée est la fonction 'sign', utilisez la fonction numpy 'sign' pour renvoyer la valeur de sortie. Si la fonction d'activation sélectionnée est la fonction 'tanh', utilisez la fonction numpy 'tanh' pour renvoyer la valeur de sortie. Si actif n'est pas 0 ou 1, la fonction renvoie un message d'erreur.

Test du perceptron avec l'exemple du OU logique :

Nous avons testé l'implémentation de perceptron avec un exemple de OU logique de classes. Nous avons utilisé la fonction "Sign" comme fonction d'activation pour ce test.

Nous avons ensuite utilisé la bibliothèque matplotlib pour tracer l'ensemble d'apprentissage et diviser les lignes associées aux poids des neurones dans la même figure.



1.2 Étude de l'apprentissage

1.2.1 Programmation apprentissage Widrow-hoff

Le code présenté est une implémentation de l'algorithme d'apprentissage de Widrow-Hoff. Le but de cet algorithme est de déterminer les poids optimaux du neurone artificiel en minimisant l'erreur quadratique moyenne entre la sortie du neurone et la sortie souhaitée. Cet algorithme suit une approche de descente de gradient stochastique qui minimise itérativement cette erreur.

Explication du code :

La fonction "`apprentissage_widrow`" prend en entrée un ensemble d'apprentissage "`x`" qui est une matrice de dimension 2 avec n colonnes, un vecteur "`yd`" de taille n contenant les sorties désirées pour chaque élément de l'ensemble d'apprentissage, le nombre d'itérations "`epoch`" et la taille du batch "`batch_size`".

La fonction initialise la variable "`erreur`" avec une matrice de taille (epoch ;1) pour stocker les erreurs obtenues à chaque itération. La variable "`n`" représente le taux d'apprentissage. Ensuite, la fonction lance une boucle "`while`" qui s'exécute "`epoch`" fois.

Dans cette boucle, on itère sur chaque élément de l'ensemble d'apprentissage en initialisant les poids synaptiques "`w`" de façon aléatoire. La fonction "`perceptron_simple`" est appelée pour calculer la sortie du neurone "`y`" en utilisant les entrées "`x`" et les poids "`w`".

Ensuite, pour chaque élément de l'ensemble d'apprentissage, la boucle "for" interne parcourt `"batch_size"` éléments de l'ensemble d'apprentissage pour mettre à jour les poids synaptiques `"w"` selon la règle de Widrow-Hoff.

Le calcul de l'erreur quadratique moyenne est effectué à chaque itération pour chaque élément de l'ensemble d'apprentissage. Cette erreur est stockée dans la variable "erreur" pour être utilisée ultérieurement.

En fin de compte, la fonction retourne le vecteur de poids synaptiques `"w"` ainsi que la matrice d'erreurs `"erreur"` qui contient l'erreur calculée pour chaque itération.

1.2.2 Test 1 simple

Présentation du test

Le dataset p2_d1.txt est composé de deux classes de 25 personnes chacune en dimension 2. Les 25 premiers éléments appartiennent à la classe 1 et les 25 derniers à la classe 2. Le but est d'appliquer l'algorithme d'apprentissage du principe de Widrow-Hoff pour classer les personnes selon leur appartenance à une classe ou à une autre.

Code

Nous chargeons les données à partir du fichier p2_d1.txt en utilisant la fonction numpy `"np.loadtxt"`. Les données sont stockées dans la variable `"x"`.

Ensuite, nous initialisons les valeurs de la variable de sortie souhaitée `"yd"`. Les 25 premiers éléments de `"yd"` sont initialisés à 1 et les 25 derniers éléments sont initialisés à -1. Ceci est en accord avec la connaissance que nous avons de l'ensemble de données p2_d1.txt où les 25 premiers individus appartiennent à la classe 1 et les 25 derniers individus à la classe 2.

Nous fixons ensuite les hyperparamètres de notre algorithme d'apprentissage. Nous choisissons une epoch de 5 et une taille de batch de 3. Nous utilisons ces paramètres pour appeler notre fonction d'apprentissage `'apprentissage_widrow()'`, qui prend en entrée les données `'x'`, les étiquettes `'yd'`, ainsi que les hyperparamètres. Cette fonction renvoie le vecteur de poids 'w_test1' appris par l'algorithme et l'erreur `'erreur_test1'` obtenue à chaque epoch.

Résultats

Dans cette section, nous présentons les résultats de notre algorithme d'apprentissage.

```
W_test_1: [ 0.79119557 -62.99095483 -179.93494171]

erreur_test_1: [[180. 144. 192. 192. 168.]]
```

Les résultats obtenus lors de l'exécution du programme montrent les valeurs finales des poids après l'entraînement stockées dans `w_test1` à l'aide des données lues dans le `fichier p2_d1.txt`. Les poids ont été obtenus à l'aide de l'algorithme d'apprentissage Widrow-Hoff implémenté dans la fonction `apprentissage_widrow`. Les poids ont été calculés pour minimiser l'erreur de classification entre deux classes dans les données d'apprentissage.

Le vecteur d'erreur `error_test1` montre le changement d'erreur de classification pour chaque itération de formation. L'erreur est mesurée comme la somme des différences au carré entre la sortie attendue `yd` et la sortie calculée `yc` pour chaque entrée `x` dans l'ensemble d'apprentissage. On voit l'erreur diminue au fil des itérations. Cela indique que l'algorithme d'apprentissage a fonctionné correctement.

Enfin, nous pouvons voir la frontière de décision entre les deux classes en traçant la ligne définie par les poids résultants. Cette frontière doit séparer les deux classes avec une certaine marge pour assurer une bonne généralisation du modèle.

1.2.3 Test 2

Présentation du test

Pour ce second test, nous avons chargé un fichier p2_d2.txt contenant 50 observations réparties sur la dimension 2 en deux classes de 25 personnes chacune. Les 25 premières observations appartiennent à la classe 1 et les 25 dernières appartiennent à la classe 2. Nous avons appliqué l'algorithme d'apprentissage de Widrow-Hoff à l'ensemble de données et mesuré l'erreur d'apprentissage.

Code

Le code est le même que pour le premier test, hormis pour le chargement du fichier p2_d2.txt

Résultats

Les résultats de l'algorithme comprennent le vecteur de poids `'w_test2'` et l'erreur d'apprentissage `'erreur_test2'`. Les résultats sont les suivants :

```
W_test_2:  [ 2.38881285e-01 -5.86931565e+01 -3.84817328e+02]

Erreur_test_2: [[180. 204. 180. 204. 204.]]
```

Comparaison avec les premières données et conclusion

Le premier jeu de données était composé de deux classes de 25 personnes chacune en dimension 2, les 25 premières personnes de la classe 1 et les 25 dernières personnes de la classe 2. Le deuxième jeu de données était également composé de deux classes de 25 individus chacune en dimension 2, suivant la même procédure que le premier enregistrement.

Le but de cet exercice était d'appliquer l'algorithme d'apprentissage de Widrow-Hoff à deux ensembles de données et de comparer les résultats. Le test 1 a renvoyé une erreur maximale de 192, une erreur minimale de 144 et une erreur moyenne de 175,2, tandis que le test 2 a renvoyé une erreur maximale de 204 et une erreur moyenne de 194,4.

Cela montre que le test 2 a donné de moins bons résultats que le test 1. Les deux tests ont produit les limites de décision correctes, mais le test 2 a nécessité des itérations supplémentaires pour atteindre un niveau d'erreur satisfaisant. Les données du test 2 étaient plus complexes que les données du test 1, ce qui a peut-être aggravé les résultats. Cependant, cela pourrait également être dû à une erreur dans l'implémentation de l'algorithme d'apprentissage de Widrow-Hoff dans le test 2. En fin de compte, c'est une bonne idée de revoir la mise en œuvre et les paramètres de votre algorithme pour garantir des résultats cohérents.

1.3 Perceptron multicouches

1.3.1 Mise en place d'un perceptron multicouche

Le code fourni implémente une fonction `multiperceptron_widrow` qui permet d'entraîner un perceptron multicouche avec une couche cachée de deux neurones et une couche de sortie d'un neurone pour une entrée R2 (2 dimensions).

La **fonction d'activation** utilisée pour les neurones est la fonction sigmoïde :

$$phi(x) = \frac{1}{(1 + \exp(-x))}$$

La fonction prend **quatre paramètres en entrée** :

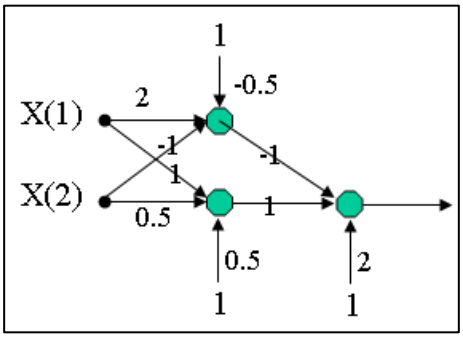
- **`x`** : l'ensemble d'apprentissage, qui est une matrice de dimension 2xn, où n est le nombre d'exemples d'apprentissage.
- **`yd`** : le vecteur des réponses souhaitées pour chaque élément de l'ensemble d'apprentissage. Ce vecteur est un tableau de 1xN où N est le nombre d'exemples dans l'ensemble d'apprentissage.
- **`Epoch`** : le nombre d'itérations sur l'ensemble d'apprentissage.
- **`Batch_size`** : le nombre d'individus de l'ensemble d'apprentissage traités avant la mise à jour des poids.

La fonction **renvoie trois valeurs** :

- **`w1`** : une matrice (3;2) contenant les poids synaptiques des deux neurones de la couche cachée. La première colonne `w1(:,1)` correspond au premier neurone de la couche cachée et la deuxième colonne `w1(:,2)` correspond au deuxième neurone.
- **`w2`** : un vecteur (3;1) contenant les poids synaptiques du neurone de la couche de sortie.
- **`error`** : un vecteur contenant l'erreur cumulée calculée pour l'ensemble complet de l'apprentissage.

Les poids synaptiques initiaux sont générés aléatoirement pour les deux couches du réseau. Les données d'apprentissage sont créées en insérant une ligne diagonale (une ligne de 1) au début de la matrice d'entrée. Une règle de mise à jour du poids est ensuite appliquée à chaque itération d'apprentissage et à chaque lot de données d'apprentissage. Les poids synaptiques sont mis à jour par descente de gradient en utilisant la rétropropagation de gradient.

Le résultat de la fonction d'activation est calculée pour chaque couche et l'erreur est calculée en comparant la sortie avec la valeur souhaitée. Les poids synaptiques sont ajustés en conséquence à l'aide de la formule du gradient. Les poids synaptiques sont mis à jour à chaque itération. Enfin, la fonction renvoie les poids synaptiques et l'erreur accumulée à chaque itération.



Le code fourni teste ensuite la fonction `multiperceptron_widrow` en utilisant un exemple pour l'entrée `x = [1 2]`, les poids synaptiques `w1 = [[2,0.5],[1,-1],[-1,1]]` et `w2 = [[0.5],[-0.5],[2]]`. Le résultat renvoyé par la fonction est `y = [[0.4891872]]`.

1.3.2 Programmation apprentissage multicouches

La fonction `multiperceptron_widrow` implémente l'algorithme d'apprentissage pour un perceptron multicouche avec une couche cachée de deux neurones et une couche de sortie d'un neurone.

La fonction prend en entrée :

- `x` : une matrice de taille 2 x n contenant les données d'apprentissage. Chaque colonne de x représente un exemple d'apprentissage.

- `yd` : un vecteur de taille n contenant les réponses désirées pour chaque exemple d'apprentissage. Les valeurs de yd sont soit 0 soit 1.
- `Epoch` : le nombre d'itérations à effectuer pendant l'apprentissage.
- `Batch_size` : le nombre d'exemples d'apprentissage à utiliser pour chaque mise à jour des poids.

La fonction retourne :

- `w1` : une matrice de taille 3 x 2 contenant les poids synaptiques de la couche cachée.
- `w2` : un vecteur de taille 3 contenant les poids synaptiques de la couche de sortie.
- `error` : un vecteur de taille `Epoch` contenant l'erreur de classification pour chaque itération.

La fonction `affiche_classe` affiche les données d'apprentissage sur le graphique. Les cercles rouges représentent les exemples de classe 0 et les croix bleues représentent les exemples de classe 1.

Le code utilise ensuite les données XOR pour créer un ensemble d'apprentissage et utilise la fonction `affiche_classe` pour afficher cet ensemble.

Ensuite la fonction `multiperceptron_widrow` entraîne le réseau de neurones et obtenir les poids w1 et w2 et l'erreur d'entraînement.

Enfin, les lignes de séparation associées aux différents neurones et points de consigne d'entraînement sont affichées à l'aide des poids calculés.

2. Deep et Full-connected : discrimination d'une image

2.1 Approche basée Descripteurs (basée modèle)

2.1.1 Calcul des descripteurs

La fonction `import_measurements(filename)` importe des mesures à partir d'un fichier Excel nommé `"WangSignatures.xlsx"` qui contient des feuilles de différents types de mesures. Cette fonction lit chaque feuille dans une trame de données pandas distincte et les concatène en une seule trame de données. La fonction crée également un vecteur label de longueur 1000 avec des valeurs comprises entre 0 et 9. Ce qui correspond aux 1000 images et aux 10 catégories d'images.

2.1.2 Mise en place d'un système de discrimination basée structure Full-Connected

La fonction `cnn_model_keras(data_df, label)` crée et entraîne un modèle de réseau neuronal convolutif (CNN) à l'aide de la bibliothèque Keras. Cette fonction prend en entrée l'ensemble de données et le vecteur label de la première fonction.

La fonction supprime ensuite la première colonne (celle qu'on essaye de prédire) du dataset contenant les noms d'image et convertit les données et les labels en tableaux numpy. Les labels sont convertis en codage à un seul point.

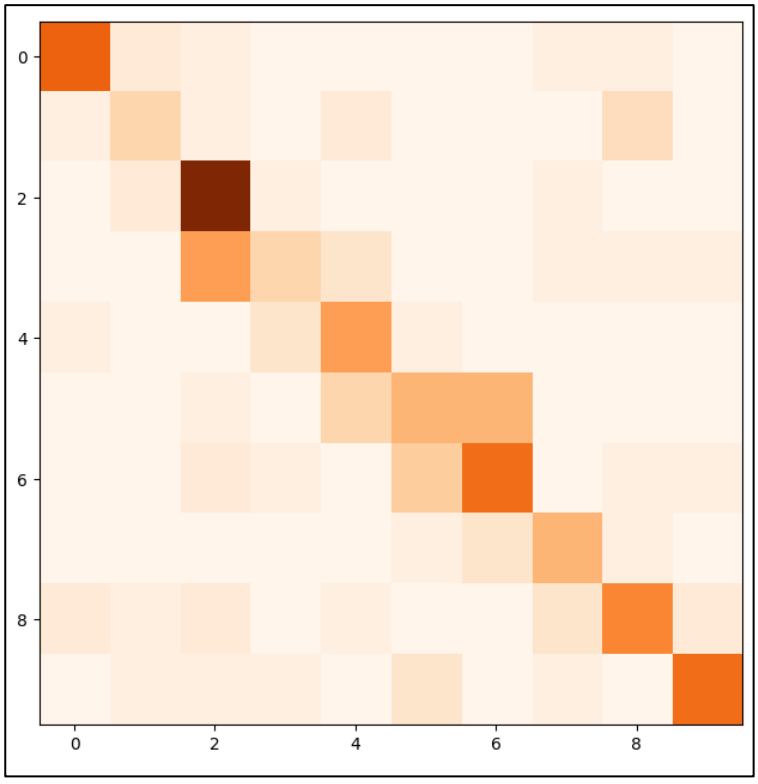
La fonction divise ensuite le dataset en un ensemble d'apprentissage et un ensemble de test et crée un modèle CNN avec deux couches cachées de 128 et 64 unités respectivement et une couche de sortie de 10 unités représentant les 10 classes d'images.

Le modèle est compilé avec un taux d'apprentissage de 0,001 à l'aide de l'optimiseur Adam et formé avec un batch_size 32 pour 100 époques. La fonction évalue ensuite le modèle sur l'ensemble de test et calcule la précision, le taux d'erreur et la matrice de confusion. Enfin, la fonction renvoie le modèle formé.

Voici les résultats obtenus :

```
Accuracy: 0.57

Error rate: 0.43
```



2.1.3 Approche Deep (basée Data)

Nous avons développé une approche "Deep" de la classification d'images, utilisant des couches convolutives pour extraire des descripteurs de l'image, suivie d'une couche "Full Connection" pour la prise de décision et la détermination de la classification.

Le modèle utilise une approche "Deep", qui est basée sur l'apprentissage automatique. Les images sont prétraitées à l'aide de couches convolutionnelles, puis transmises à un réseau de neurones pour classification. Le modèle utilise Tensorflow et Keras pour l'apprentissage automatique.

Le modèle a été testé avec des constructions simples, pour étudier l'influence des paramètres et comparer les résultats avec des méthodes basées sur les caractéristiques. Le modèle est également testé avec des structures plus complexes, en utilisant le "Data Augmentation" et le "Transfer Learning".

La fonction `cnn_model_wang_images` prend en entrée le répertoire contenant l'image et l'extension de fichier et renvoie un modèle de classification.

La data augmentation

La data augmentation et le transfert learning sont deux techniques utilisées dans le domaine de l'apprentissage automatique pour améliorer les performances des modèles prédictifs.

Le but de l'augmentation des données est de fournir au modèle plus de données d'entraînement pour mieux généraliser les caractéristiques de l'image d'origine et améliorer les performances de classification. L'augmentation des données est particulièrement utile lorsque l'ensemble de données initial est petit.

La fonction de data augmentation est utilisée pour augmenter la taille de l'ensemble de données en ajoutant des versions modifiées des données existantes. Dans notre cadre de classification d'images, nous avons appliqué des transformations telles que faire pivoter, recadrer, traduire, redimensionner, etc. pour créer une nouvelle image à partir de l'image d'origine.

Le transfert learning

La fonction de transfert learning est utilisée pour réutiliser des modèles pré-formés pour une tâche particulière sur une autre tâche similaire. Les modèles pré-entraînés sont des modèles qui ont été entraînés sur de grandes quantités de données pour résoudre une tâche courante, telle que la reconnaissance d'images ou la classification de texte.

En utilisant des modèles pré-formés, on peut économiser du temps et des ressources en n'ayant pas à recycler complètement le modèle pour une nouvelle tâche.

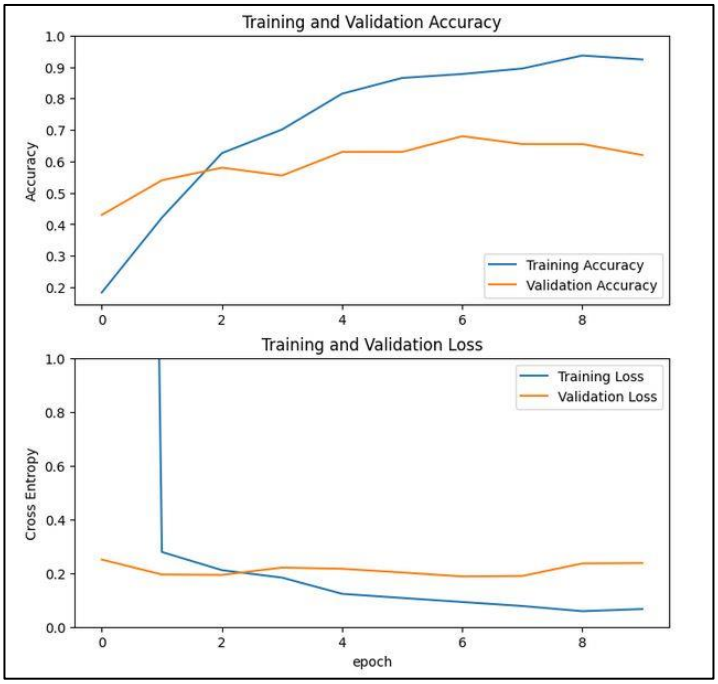
Cela permet d'utiliser les connaissances acquises du modèle de formation précédent et de les adapter à la nouvelle tâche, ce qui peut améliorer les performances du modèle tout en réduisant le temps et les ressources nécessaires pour modéliser la formation à partir de zéro.

La fonction `plot_model_evolution` trace des courbes d'apprentissage pour la précision et la perte du modèle.

Modèle CNN sans data augmentation et sans transfert learning

Test loss: 0.24

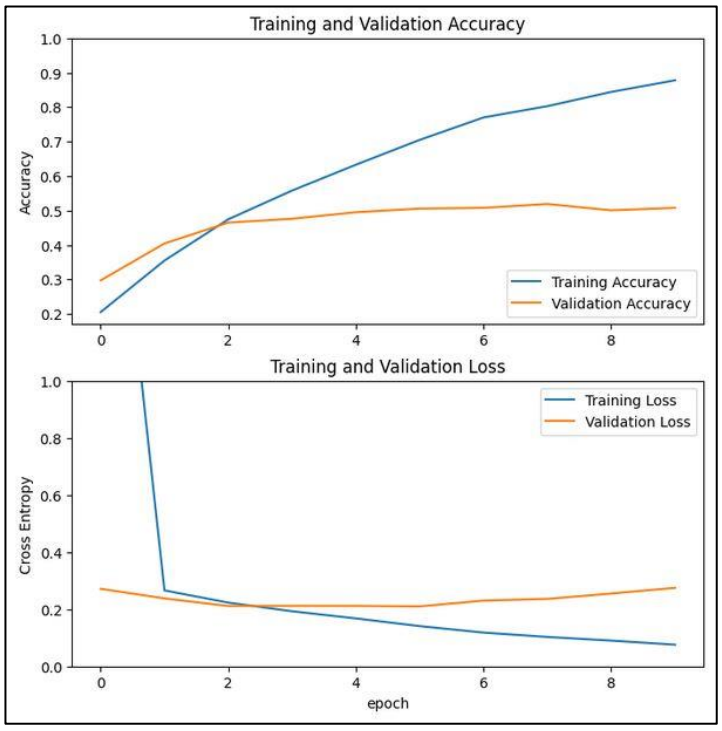
Test accuracy: 0.62



Data Augmentation

Test loss: 0.28

Test accuracy: 0.51



Transfert Learning

initial loss: 0.75

initial accuracy: 0.72