Rapport : Projet Neural Network Digit Recognizer

- **Rapport : Projet Neural Network Digit Recognizer**
  - **I. Introduction**
    - Présentation du projet
      - *Objectifs du projet*
    - Présentation des données
  - **II. Réseaux de Neurones**
    - Qu'est-ce qu'un réseau de neurones ?
    - Entraînement d'un réseau
  - **III. Implémentation**
    - 3.1 Programmation du réseau
      - *Forward Pass*
      - *Backpropagation*
      - *Validation initiale*
    - 3.2 Entraînement et test
      - *Dataset de trois échantillons*
      - *Dataset plus grand*
  - **IV. Résultats**
    - Programmation du réseau de neurones
    - Evolution des pertes sur les données d'exemple
    - Entrainement du modèle
    - Différence de l'erreur entre le modèle entrainé et non entrainé
  - **Conclusion**
    - Synthèse des résultats

I. Introduction

Présentation du projet

Ce projet vise à programmer un réseaux de neurones artificiels en Python.
Il met l'accent sur les concepts fondamentaux des réseaux de neurones, notamment le **forward pass**, le **backpropagation**, l'entraînement, et les tests.

Ce projet se découpe en deux parties : la programmation du réseau de neurone, et l'entrainement d'un modèle avec une phase d'ajustement du modèle.

Objectifs du projet

L'objectif principal est de construire et entraîner un réseau de neurones ayant :

- **2 entrées**.
- **2 sorties**.
- **une couche cachée**.

Les tâches principales sont :

1. Programmer le **forward pass** et le **backpropagation** pour un réseau avec des poids initiaux donnés.
2. Tester le réseau sur un petit dataset de **deux échantillons étiquetés**.
3. Étendre le processus d'entraînement à plusieurs itérations.
4. Utiliser le réseau sur un dataset plus grand pour l'entraîner, puis tester sa capacité à généraliser sur de nouveaux échantillons.

Présentation des données

Le projet repose sur deux datasets :

- **NNTraining.data** : un ensemble de données d'entraînement, structuré pour entraîner le réseau.
- **NNTest.data** : un ensemble de données de test, utilisé pour évaluer les performances du réseau après entraînement.

Chaque ligne des fichiers représente un exemple avec deux caractéristiques en entrée ((x_1) et (x_2)) et deux cibles ((t_1) et (t_2)).

II. Réseaux de Neurones

Qu'est-ce qu'un réseau de neurones ?

Un **réseau de neurones artificiels** est un modèle d'apprentissage inspiré du fonctionnement du cerveau humain. Il est composé de neurones organisés en couches :

- Une **couche d'entrée** pour recevoir les données,
- Une ou plusieurs **couches cachées** pour extraire des caractéristiques,
- Une **couche de sortie** pour produire une prédiction.

Chaque connexion entre les neurones est associée à un **poids**, ajusté lors de l'entraînement pour minimiser l'erreur entre les prédictions et les vraies valeurs. Ce processus, appelé **backpropagation**, repose sur la propagation du gradient de l'erreur en utilisant l'algorithme de descente de gradient.

Entraînement d'un réseau

L'entraînement d'un réseau de neurones consiste à ajuster les poids des connexions entre les neurones pour minimiser une **fonction de perte** (comme l'erreur quadratique ou l'entropie croisée). Ce processus se déroule en plusieurs étapes :

1. **Propagation avant (forward pass)** : Les données traversent les couches du réseau, et les sorties sont calculées.
2. **Calcul de la perte** : La différence entre la sortie du modèle et la vérité terrain est mesurée.
3. **Rétropropagation (backpropagation)** : L'erreur est propagée en arrière à travers le réseau pour calculer les gradients des poids.
4. **Mise à jour des poids** : Les poids sont ajustés en utilisant une méthode comme la **descente de gradient**, en fonction des gradients calculés.

Ce processus est répété pour chaque itération, ou **epoch**, jusqu'à ce que le modèle atteigne un niveau de précision satisfaisant.

III. Implémentation

3.1 Programmation du réseau

Forward Pass

Le **forward pass** consiste à propager les entrées ((x_1, x_2)) à travers le réseau jusqu'à la sortie ((y_1, y_2)). Cela inclut le calcul des activations des neurones de la couche cachée et de sortie, en utilisant la **sigmoïde** comme fonction d'activation.

$$
\sigma(z) = \frac{1}{1 + e^{-z}}
$$

Backpropagation

La **backpropagation** calcule les gradients de la fonction de perte (ici, l'erreur quadratique moyenne) par rapport aux poids.
Ces gradients sont ensuite utilisés pour mettre à jour les poids du réseau via une méthode de **descente de gradient**.

Validation initiale

Le réseau est testé sur un petit dataset de **deux échantillons** pour valider la mise en œuvre du forward pass et du backpropagation.

3.2 Entraînement et test

Dataset de trois échantillons

Un réseau est entraîné sur un petit dataset de **trois échantillons** pour observer la diminution de la fonction de perte au fil des itérations. Cette étape permet de visualiser la convergence et de valider la capacité d'apprentissage du réseau.

Dataset plus grand

Le réseau est ensuite appliqué à un dataset plus grand (**NNTraining.data**). L'ensemble est divisé en **mini-batches** pour accélérer l'apprentissage. Après chaque **epoch** (parcours complet du dataset), la perte moyenne est calculée et enregistrée pour traquer l'évolution de l'entraînement.

Le modèle final est évalué sur le dataset de test (**NNTest.data**) pour mesurer sa capacité à généraliser.

IV. Résultats

Programmation du réseau de neurones

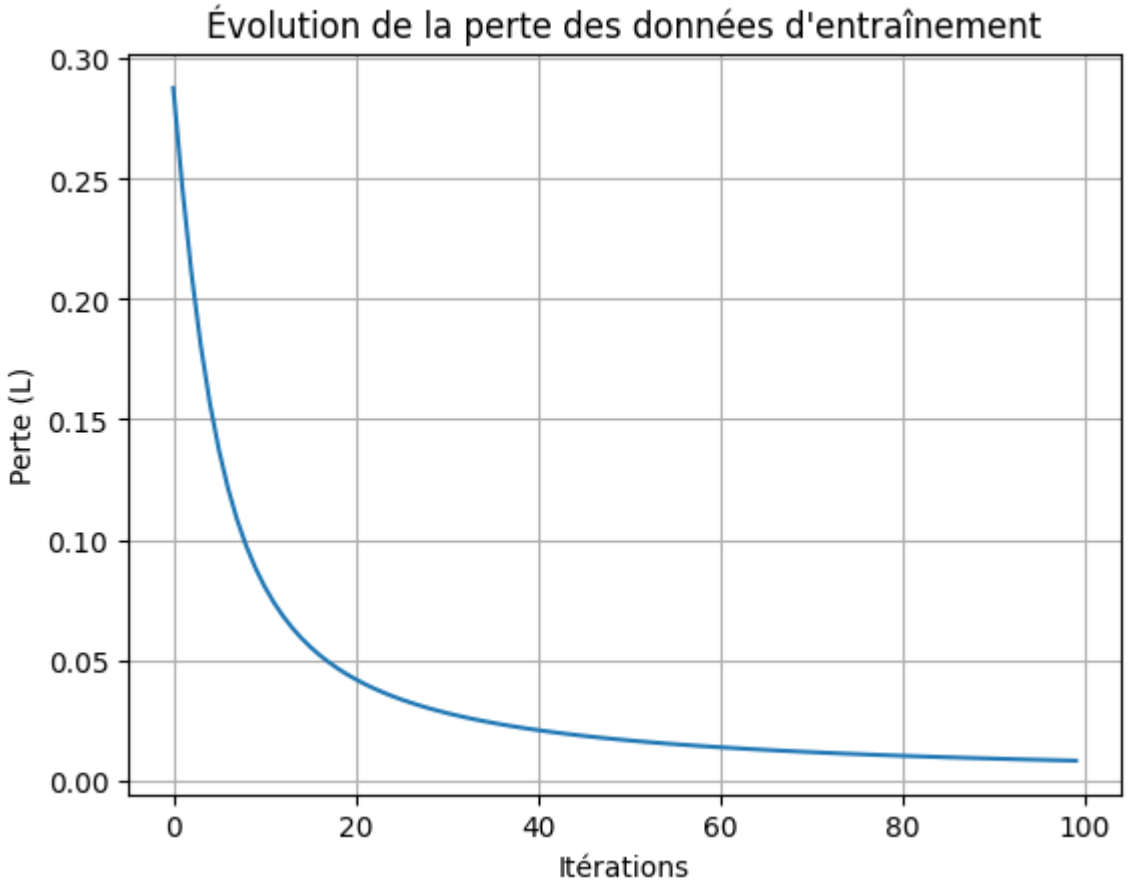
Le réseau de neurones que j'ai programmé me donne :

- Pour x1 et t1:
    - Sorties du réseau : `y1 = 0.682`, `y2 = 0.379`
    - **Perte quadratique (Loss) : `L = 0.245`**
    - Nouveau gradient de `L = [-0.1245799920310608, 0.06774106237304037, -0.0622899960155304, 0.033870531186520184, -0.10106080200900529, 0.13055095827032773, -0.016478491947864934, 0.021287016052494324]`

- Pour x2 et t2:
    - Sorties du réseau : `y1 = 0.562`, `y2 = 0.881`
    - **Perte quadratique (Loss) : `L = 0.330`**
    - Nouveau gradient de `L = [-5.039984614038973e-06, -5.84951168236954e-09, 1.511995384211692e-05, 1.7548535047108622e-08, 4.622002203780851e-06, -4.180665865731328e-07, 0.2767429523030238, -0.02503178846017317]`

Nous obtenons les erreurs et les gradients prévus.

Evolution des pertes sur les données d'exemple



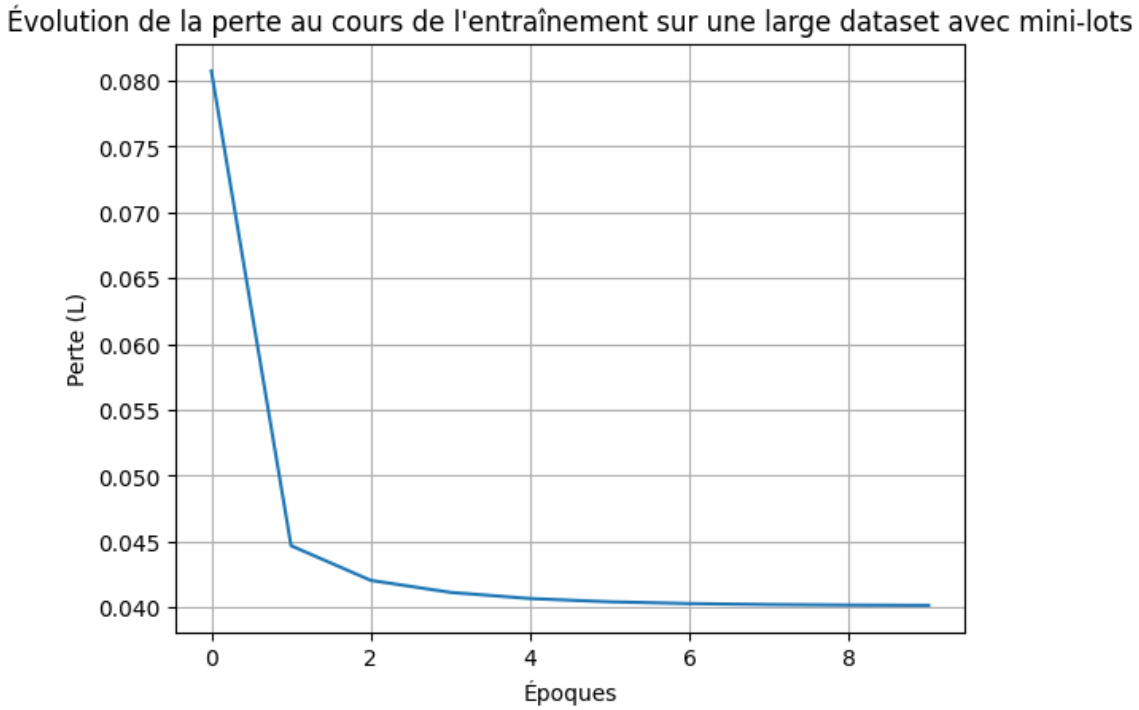
Sur ce graphique, nous pouvons observer l'évolution de la perte (ou erreur) au fil des itérations d'entraînement du modèle.
La courbe suit une tendance classique, similaire à celle présentée dans le support de TP : l'erreur diminue progressivement pour tendre vers zéro.

Ces résultats ont été obtenu avec ces hyperparameters :

- **Taux d'apprentissage** : (\rho = 1.2)
- **Nombre d'itérations** : (N = 100)

La perte finale atteinte après l'entraînement est de **0.8%**.
Lorsqu'on utilise ce modèle pour faire des prédictions, la perte calculée pour (x3) et (t3) est de **1.754797**.

Entrainement du modèle

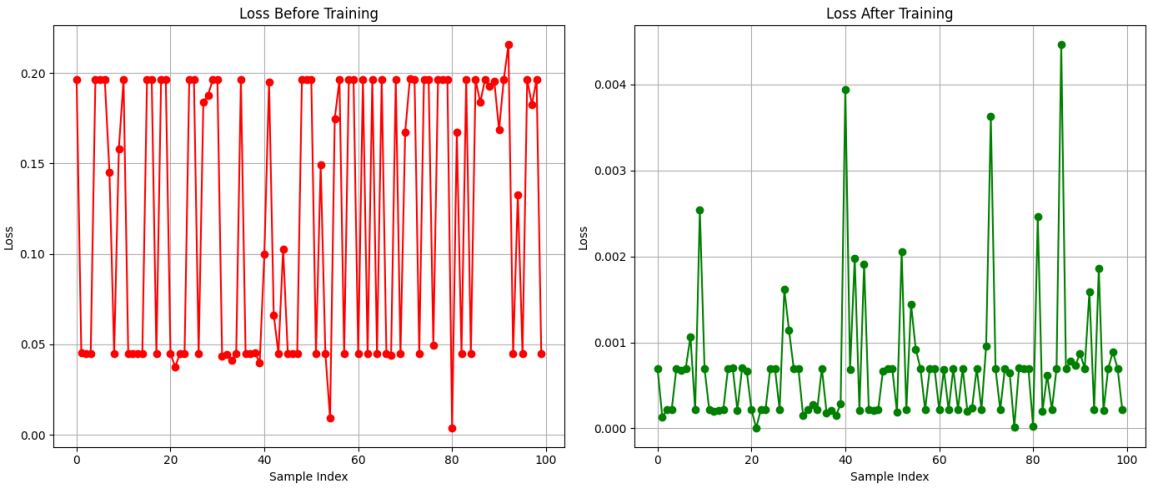


Ces résultats ont été obtenu avec ces hyperparameters :

- Taux d'apprentissage = 10
- Taille des mini-lots = 50
- Nombre d'époques = 10

Avec Nous obtenons une perte moyenne de 0.040155.

Différence de l'erreur entre le modèle entrainé et non entrainé



Sur les deux graphiques, nous pouvons observer la distribution des erreurs commises par le modèle avant et après l'entraînement.

- **Avant l'entraînement** :
  L'erreur du modèle est relativement élevée, se situant principalement dans une plage comprise entre **0.2 et 0.05**. Il semblerait que le réseau effectue des prédictions presque aléatoires, ce qui explique la large dispersion des erreurs.

- **Après l'entraînement** :
  Après avoir ajusté les poids via le processus d'entraînement, l'erreur est significativement réduite. La majorité des erreurs se situent désormais dans une plage comprise entre **0 et 0.001**, avec quelques rares pics atteignant **0.004**.

En comparant les deux distributions, il est clair que l'entraînement permet d'améliorer drastiquement les performances du modèle. Un modèle non entraîné produit des erreurs importantes, tandis qu'un modèle correctement entraîné réduit ces erreurs à des niveaux négligeables pour la plupart des échantillons.

Conclusion

Synthèse des résultats

Ce projet a permis d'explorer de manière pratique la mise en œuvre d'un réseau de neurones simple en Python, en suivant les étapes essentielles pour comprendre son fonctionnement et son apprentissage. À travers ce rapport, nous avons abordé les différentes phases de conception, d'entraînement et de test d'un réseau de neurones caractérisé par deux entrées, une couche cachée et deux sorties.

Nous avons commencé par implémenter une fonction pour effectuer un passage avant, où les données sont propagées à travers le réseau pour obtenir une prédiction basée sur les poids initiaux.
Nous avons ensuite intégré la rétropropagation dans cette fonction, une étape essentielle permettant de calculer les gradients des poids en fonction des erreurs observées. Cela a permis d'ajuster les poids du réseau afin de réduire progressivement l'erreur.

En utilisant un jeu de données réduit contenant deux échantillons étiquetés, nous avons entraîné le réseau sur plusieurs itérations. Cela a permis de visualiser comment la perte diminue progressivement grâce à l'optimisation des poids.

Une fois le fonctionnement validé sur un petit jeu de données, nous avons appliqué les mêmes concepts à un jeu de données plus conséquent (`NNTraining.data`) pour entraîner le réseau.
Le modèle a ensuite été testé sur un jeu de données séparé (`NNTest.data`), permettant d'évaluer sa capacité à généraliser sur des données inconnues.