Nom : Mathis Pezet

Compte-Rendu TP Perceptron Multi-Couches

Introduction

Dans le dernier TP, on a vu que le perceptron simple n'arrivait pas à résoudre le problème du XOR car les données ne sont pas séparables avec une seule ligne droite.

Ce TP a donc pour but de construire un Perceptron Multi-Couches (MLP), pour voir s'il pouvait dépasser les limites imposées par le perceptron simple. Les objectifs étaient donc :

- Comprendre pourquoi on a besoin de plusieurs couches.
- Coder un réseau avec des couches cachées.
- Implémenter l'algorithme de rétropropagation pour l'entraîner.
- Vérifier s'il arrive enfin à résoudre le XOR.

Réponses aux Questions Théoriques

Exercice 1.1 - Questions d'analyse théorique :

- **Que signifie concrètement le théorème d'approximation universelle ?**
  Ça veut dire qu'un réseau avec une seule couche cachée, si elle est assez grande, peut en théorie dessiner n'importe quelle fonction continue. C'est comme essayer de dessiner un cercle avec des petits segments de droite : plus on a de segments (de neurones), plus on arrive à une bonne approximation du cercle.
- **Ce théorème garantit-il qu'on peut toujours trouver les bons poids ?**
  Non, le théorème dit juste que la solution parfaite (les bons poids) existe. Ça ne garantit pas que notre algorithme d'entraînement va la trouver. On peut se retrouver coincé dans une solution "pas trop mal" mais pas la meilleure (un minimum local).
- **Quelle est la différence entre "pouvoir approximer" et "pouvoir apprendre" ?**
  "Pouvoir approximer", c'est la capacité théorique du modèle (le fait que la solution existe). "Pouvoir apprendre", c'est la capacité de notre algorithme à trouver cette solution à partir des données qu'on lui donne.
- **Pourquoi utilise-t-on souvent beaucoup plus de couches cachées en pratique ?**
  Parce que c'est souvent plus efficace. Une seule grosse couche, c'est un peu "bourrin". Plusieurs couches permettent au réseau d'apprendre des choses de plus en plus complexes, étape par étape. La première couche apprend des formes simples, la

deuxième combine ces formes simples pour en faire des plus complexes, etc. C'est souvent plus performant.

- **En principe, vous avez déjà vu au lycée un autre type d'approximateur de fonctions, donner leurs noms ?**
  Oui, les polynômes, avec le développement de Taylor par exemple. On fait une approximation d'une fonction autour d'un point avec une somme de termes.

Exercice 1.2 - Expliquer la phrase suivante :
"Le théorème d'approximation universelle affirme qu'un réseau profond peut exactement retrouver les données d'entraînement."
Ça veut dire que le réseau est tellement flexible qu'il peut "tricher" en apprenant par cœur les données d'entraînement. Il peut créer une fonction qui passe exactement par tous les points qu'on lui a donnés. Le danger, c'est qu'il ne saura rien faire avec de nouvelles données. C'est le sur-apprentissage.

Méthodes

Pour faire ça, j'ai continué à utiliser Python et Numpy. L'idée n'était plus d'avoir un seul neurone, mais plusieurs, organisés en couches.

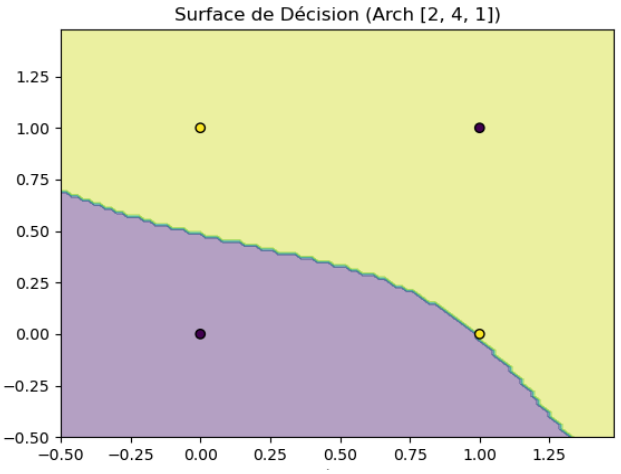
- **Architecture des réseaux multicouches :** L'idée, c'est d'empiler des couches de neurones. Il y a une couche d'entrée, une ou plusieurs couches "cachées", et une couche de sortie. Les couches cachées sont la clé : elles transforment les données pour qu'à la fin, la couche de sortie puisse faire une séparation simple.
- **Algorithme de rétropropagation :** Pour que le réseau apprenne, on fait deux choses :
  - **Une passe "avant" (forward)** : On donne les données en entrée, et elles traversent chaque couche jusqu'à la sortie pour donner une prédiction. On sauvegarde les calculs intermédiaires.
  - **Une passe "arrière" (backward)** : On calcule l'erreur entre la prédiction et la vraie réponse. Puis, on remonte le réseau à l'envers, et à chaque couche, on calcule à quel point ses poids sont responsables de l'erreur. On ajuste alors un peu les poids pour réduire cette erreur la prochaine fois.

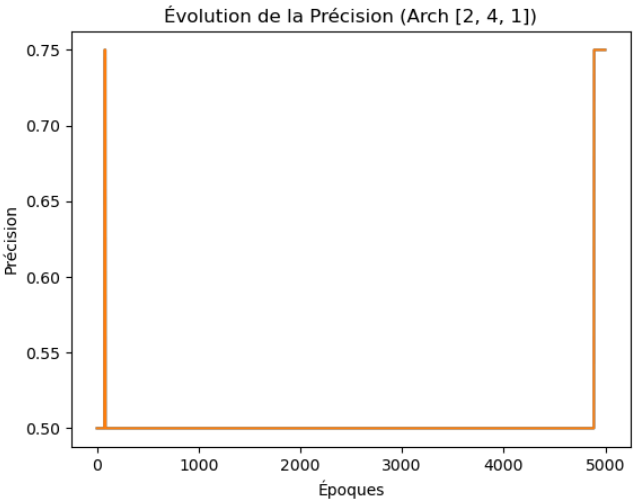
- **Fonctions de coût et d'optimisation :** Pour savoir si le réseau se trompe, on a utilisé l'erreur quadratique moyenne (MSE). Pour réduire cette erreur, on a utilisé la descente de gradient, comme dans le TP précédent, en ajustant les poids grâce au taux d'apprentissage.

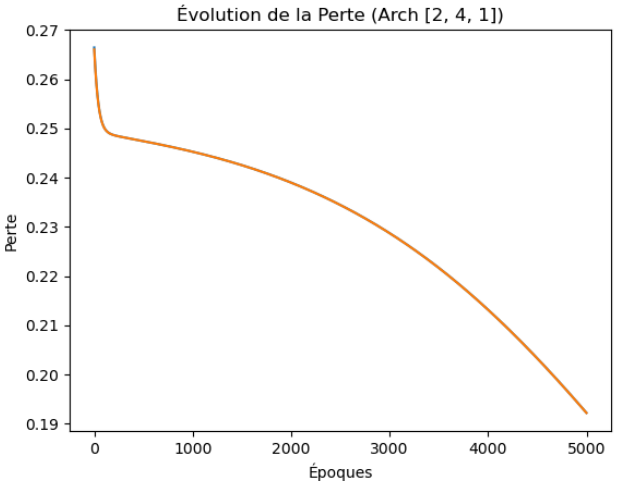
Résultats

Tentative N°1

Ma première approche pour résoudre le problème XOR a été de tester une architecture `[2, 4, 1]` (2 neurones en entrée, 4 dans la couche cachée, 1 en sortie) avec un taux d'apprentissage de `0.1`, entraînée sur 5000 époques.Voici ce que ça a donné:



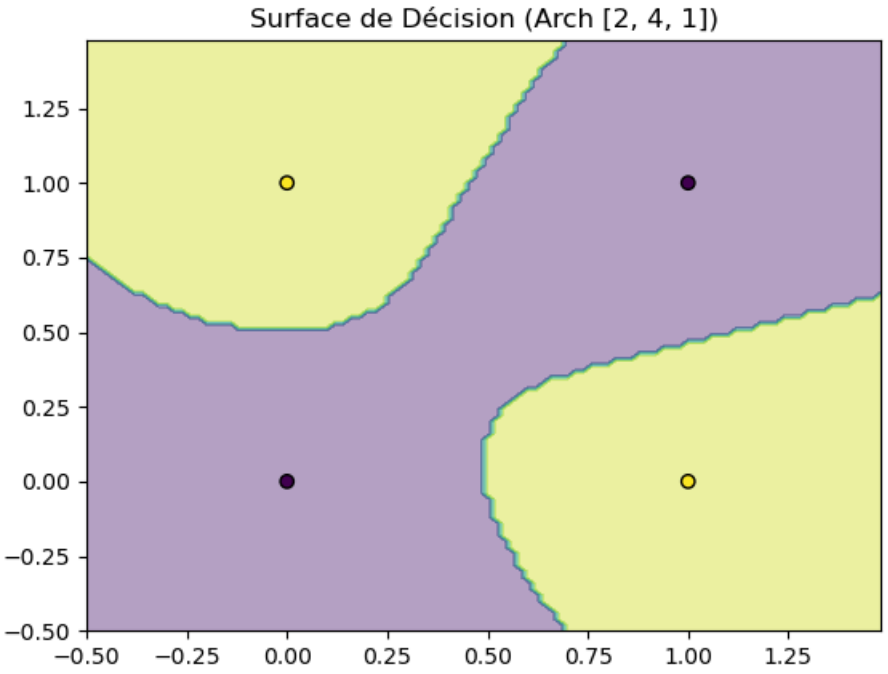


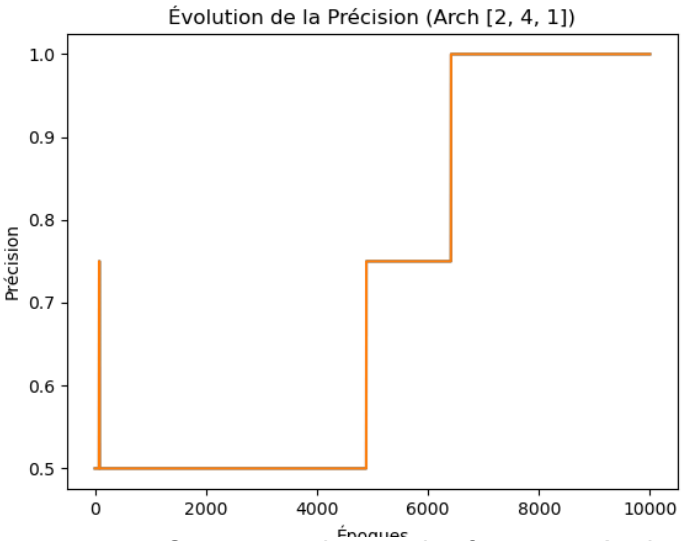


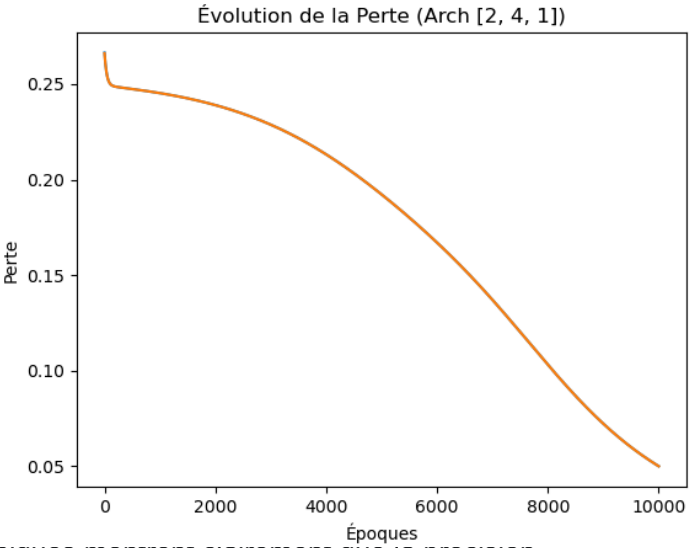
L'analyse de ces graphiques révèle un échec partiel. La précision se bloque immédiatement à 0.75, signifiant que le modèle ne classe correctement que 3 des 4 points. La surface de décision confirme ce diagnostic : elle classe incorrectement le point `(1, 1)`. Bien que la perte continue de diminuer, cela indique que le modèle devient plus confiant dans ses prédictions (bonnes ou mauvaises) mais n'arrive pas à corriger son erreur fondamentale. Le réseau est coincé dans un minimum local.

Tentative N°2

Face à ce blocage, j'ai émis l'hypothèse que le modèle n'avait simplement pas eu assez de temps pour converger car en supposant le minimum local, j'ai pensé que le modèle allait s'en sortir si on lui laissait simplement plus de temps. J'ai donc décidé de doubler le nombre d'époques, passant de 5000 à 10000, tout en conservant les autres paramètres identiques.







Cette seconde tentative fut un succès. Les graphiques montrent clairement que la précision atteint finalement 1(100%), prouvant que le modèle a réussi à s'échapper du minimum local en

augmentant simplement le nombre d’époques. La surface de décision finale sépare désormais parfaitement les quatre points du problème XOR. La descente continue de la perte vers une valeur proche de zéro confirme l’atteinte d’une solution optimale

Discussion

- **Avantages et inconvénients des réseaux multicouches :** Le gros point fort, c'est qu'un MLP peut apprendre des relations très complexes et non-linéaires. Il est beaucoup plus puissant que le perceptron simple. Le point faible, c'est que c'est plus compliqué car il y a beaucoup plus de choses à règler : le nombre de couches, de neurones, le taux d'apprentissage... Comme l'a montré mon expérience, un mauvais choix de ces paramètres peut empêcher le réseau d'apprendre.
- **Problèmes de sur-apprentissage :** Un nouveau risque apparaît : le sur-apprentissage (overfitting). C'est quand le modèle est tellement puissant qu'il apprend par cœur les données d'entraînement, y compris le bruit. Il devient super bon sur les données qu'il a déjà vues, mais nul sur de nouvelles données.
- **Stratégies de régularisation :** Pour éviter le sur-apprentissage, il y a des techniques. La régularisation L2, par exemple, pénalise le modèle s'il utilise des poids trop grands, ça le force à trouver des solutions plus simples. Le dropout est une autre technique : pendant l'entraînement, on "éteint" au hasard certains neurones, ce qui oblige les autres à mieux travailler et rend le réseau plus robuste.

Conclusion : Bilan et Perspectives

En conclusion, ce TP m'a permis de construire un réseau de neurones qui résout le problème du XOR, là où le perceptron simple échouait. L'analyse des résultats a été particulièrement instructive : elle a démontré qu'avoir une architecture capable de résoudre un problème ne garantit pas le succès de l'apprentissage. Il faut aussi choisir les bons paramètres (comme le nombre d'époques ou le taux d'apprentissage) pour éviter les pièges comme les minima locaux.

Coder la rétropropagation soi-même, même si c'était un peu difficile au début, m'a vraiment aidé à voir comment le réseau "pense" et corrige ses erreurs. C'est beaucoup plus clair que de juste lire la théorie. Pour la suite, on pourrait utiliser ce code pour attaquer des problèmes plus intéressants, comme la classification d'images simples (MNIST) ou d'autres datasets non-linéaires.