Fabriquer son propre réseau de neurones

Téo Chauvin (Z-118)
teo.chauvin@enac.fr

October 27, 2025

1 Présentation

Les réseaux de neurones sont devenus un pilier de l'intelligence artificielle, en particulier du *Deep Learning*. Ils sont utilisés dans de nombreuses applications : prédiction, reconnaissance, ou encore génération de nouvelles données. Il existe aujourd'hui une grande variété d'architectures, chacune adaptée à un objectif et à un type de données spécifiques.

Dans ce projet, nous nous intéresserons aux réseaux *feed-forward*, où la donnée d'entrée traverse successivement plusieurs couches (*layers*) avant de produire une sortie. Ce type de réseau est également appelé *MLP* (*Multi-Layer Perceptron*).

L'architecture du MLP est à la fois simple et fondamentale : elle constitue la base de nombreuses variantes plus complexes (CNNs, GANs, autoencodeurs, etc.).

Objectif du projet : construire et entraîner notre propre MLP en Python, afin de comprendre les principes fondamentaux du fonctionnement d'un réseau de neurones artificiel.

À l'issue de ce travail, vous aurez appris à :

- Comprendre ce qu'est un MLP sur le plan théorique.
- Expliquer comment il apprend à partir de données d'entrée.
- Distinguer les problèmes de régression et de classification.
- Mettre en œuvre une descente de gradient.
- Exploiter et diviser des jeux de données (train/validation/test sets).
- Analyser l'influence des fonctions de perte et d'activation.
- Reconnaître le sur-apprentissage (*overfitting*).
- Utiliser le minibatching/full batching.

2 Quelques explications

Quelques explications et ressources nécessaires à la réalisation du projet. Tous ces points seront réexpliqués durant le projet. Si tout n'est pas encore clair, c'est normal.

2.1 Fonctionnement d'un neurone

Le fonctionnement d'un neurone artificiel est inspiré des neurones biologiques. Un neurone possède N entrées et une seule sortie. Les N entrées correspondent, par exemple, aux sorties d'autres neurones. Chaque entrée x_i est pondérée par un poids $w_i \in R$. Le neurone possède également un biais $b \in R$.

Le neurone calcule d'abord une combinaison linéaire des entrées, donnée par la somme pondérée des entrées et des poids :

$$z = w^\top x + b.$$

Ensuite, il applique à ce résultat une fonction non linéaire appelée *fonction d'activation* σ. La sortie du neurone, appelée *activation* a, est donc :

$$a = \sigma(w^\top x + b).$$

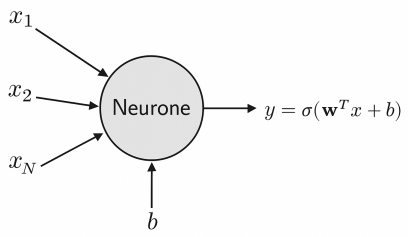

Figure 1: Le perceptron

2.2 Une couche dense du MLP

Une couche du MLP est constituée d'un ensemble de neurones. Elle est dite *dense* lorsque toutes les entrées des neurones de la couche l sont connectées aux sorties des neurones de la couche $l-1$. On note $h^{(l)}$ la hauteur de la couche l, c'est-à-dire le nombre de neurones sur cette couche. On en déduit qu'il y a $h^{(l)} \times h^{(l-1)}$ connexions entre les couches l et $l-1$.

Les poids associés aux entrées de la couche l sont regroupés dans une matrice $W^{(l)}$ de taille $h^{(l)} \times h^{(l-1)}$. Par exemple, le poids $w_{ij}^{(l)}$ correspond au poids que le neurone i de la couche l associe au neurone j de la couche $l-1$. De même, les

biais de la couche l sont regroupés dans le vecteur $b^{(l)}$ de taille $h^{(l)}$. Les entrées de la couche l sont représentées par le vecteur $x^{(l)}$, avec $x^{(l)} = a^{(l-1)}$, c'est-à-dire que l'entrée d'une couche correspond à l'activation de la couche précédente.

On obtient ainsi une expression compacte pour toute la couche de neurones :

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}, \qquad a^{(l)} = \sigma^{(l)}(z^{(l)}).$$

En supposant que le réseau comporte L couches successives, on obtient en sortie :

$$\hat{y} = \sigma^{(L)}\big(W^{(L)}\sigma^{(L-1)}(W^{(L-1)} \dots \sigma^{(1)}(W^{(1)}x + b^{(1)}) \cdots + b^{(L-1)}) + b^{(L)}\big).$$

On appelle **poids** ou **paramètres** du réseau de neurones les couples $(W^{(l)}, b^{(l)})_{l=1}^{L}$. On les regroupe souvent sous la forme d'un unique vecteur de paramètres θ, de sorte que la sortie du réseau puisse s'écrire :

$$\hat{y} = F(x; \theta),$$

où F est une fonction non linéaire dépendant de θ dont on ne connaît pas l'expression analytique.

2.3 Apprentissage

Considérons un ensemble de données d'entraînement (x, y). L'objectif de l'apprentissage est d'estimer la relation entre x et y, de manière à ce qu'une fois le réseau entraîné, celui-ci soit capable de prédire des valeurs y cohérentes pour de nouvelles entrées x non vues durant l'entraînement. Mathématiquement, on cherche à faire converger F vers une fonction cible inconnue f^* en ajustant les paramètres θ pour minimiser une fonction de coût J mesurant l'erreur de prédiction :

$$\theta^* = \operatorname*{argmin}_{\theta} J(F(x; \theta), y).$$

Par exemple, si l'on choisit la *Mean Square Error* (MSE) :

$$J(\hat{y}, y) = \frac{1}{2}\|\hat{y} - y\|^2, \qquad \theta^* = \operatorname*{argmin}_{\theta} J(F(x; \theta), y).$$

L'apprentissage correspond donc à la résolution d'un problème d'optimisation non linéaire, que l'on ne peut pas résoudre analytiquement. On utilise alors des méthodes itératives approchées, appelées *descente de gradient*.

L'idée est la suivante :

1. Calculer la prédiction $F(x; \theta)$: c'est l'étape **forward**, où la donnée traverse le réseau jusqu'à la sortie.
2. Calculer l'erreur de prédiction J.
3. Ajuster les paramètres dans la direction opposée au gradient pour réduire l'erreur de prédiction J.

La mise à jour des paramètres s'écrit :

$$W^{(l)} \leftarrow W^{(l)} - \eta \frac{\partial J}{\partial W^{(l)}},$$
$$b^{(l)} \leftarrow b^{(l)} - \eta \frac{\partial J}{\partial b^{(l)}},$$

où η est le *learning rate* (taux d'apprentissage).

2.4 Calcul des gradients : la *backpropagation*

La fonction F étant une combinaison de fonctions non linéaires, il est difficile d'en calculer directement les gradients. On utilise donc l'algorithme de *rétropropagation du gradient*, qui applique la règle de dérivation en chaîne pour propager les gradients de la sortie vers l'entrée du réseau.

La dernière couche L est la plus simple à traiter, car on connaît :

$$\frac{\partial J}{\partial a^{(L)}} = \frac{\partial}{\partial a^{(L)}} \frac{1}{2} \|a^{(L)} - y\|^2 = a^{(L)} - y.$$

Comme $a^{(L)} = \sigma^{(L)}(z^{(L)})$, on a :

$$\frac{\partial J}{\partial z^{(L)}} = \frac{\partial J}{\partial a^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} = \frac{\partial J}{\partial a^{(L)}} \odot \sigma'^{(L)}(z^{(L)}).$$

On note ce vecteur $\delta^{(L)}$ le niveau d'erreur de la couche L :

$$\delta^{(L)} = (a^{(L)} - y) \odot \sigma'^{(L)}(z^{(L)}).$$

Propagation de l'erreur dans une couche cachée. Pour une couche cachée $l < L$, on cherche à calculer $\delta^{(l)} = \frac{\partial J}{\partial z^{(l)}}$. Tel que,

$$\delta^{(l)} = \frac{\partial J}{\partial a^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}}.$$

Or, on ne connaît pas $\frac{\partial J}{\partial a^{(l)}}$ pour la couche l. Pour résoudre ce problème, on repart de la relation entre deux couches consécutives :

$$z^{(l+1)} = W^{(l+1)} a^{(l)} + b^{(l+1)}, \quad \text{et} \quad a^{(l)} = \sigma^{(l)}(z^{(l)}).$$

En appliquant la règle de dérivation en chaîne, on obtient :

$$\frac{\partial J}{\partial z^{(l)}} = \frac{\partial J}{\partial a^{(l)}} \odot \sigma'^{(l)}(z^{(l)}).$$

Or, $\frac{\partial J}{\partial a^{(l)}}$ dépend de la couche suivante :

$$\frac{\partial J}{\partial a^{(l)}} = \left(\frac{\partial z^{(l+1)}}{\partial a^{(l)}} \right)^\top \frac{\partial J}{\partial z^{(l+1)}} = (W^{(l+1)})^\top \delta^{(l+1)}.$$

En remplaçant dans l'équation précédente :

$$\delta^{(l)} = \left((W^{(l+1)})^\top \delta^{(l+1)} \right) \odot \sigma'^{(l)}(z^{(l)}).$$

Résultat final. On obtient donc, pour toutes les couches du réseau :

$$\boxed{\begin{aligned}\delta^{(L)} &= (a^{(L)} - y) \odot \sigma'^{(L)}(z^{(L)}),\\ \delta^{(l)} &= (W^{(l+1)})^\top \delta^{(l+1)} \odot \sigma'^{(l)}(z^{(l)}).\end{aligned}}$$

Les gradients des poids et biais s'en déduisent naturellement :

$$\boxed{\begin{aligned}\frac{\partial J}{\partial W^{(l)}} &= \frac{\partial J}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial W^{(l)}} = \delta^{(l)} (a^{(l-1)})^\top\\ \frac{\partial J}{\partial b^{(l)}} &= \frac{\partial J}{\partial z^{(l)}} \frac{\partial z^{(l)}}{\partial b^{(l)}} = (\delta^{(l)})^\top 1.\end{aligned}}$$

Après l'étape **forward**, on connaît tous les $z^{(l)}$ et $a^{(l)}$ du réseau. On évalue ensuite la fonction de perte J, puis on calcule les gradients de la dernière couche jusqu'à la première : c'est l'étape **backward**. Une fois revenu à la première couche, on met à jour tous les paramètres du réseau. En répétant ces étapes pendant plusieurs *epochs*, le réseau converge (ou pas) vers une solution qui tend à minimiser l'erreur de prédiction.

2.5 Un petit exemple

On considère x un vecteur de 500 echantillons tirés uniformément de l'intervalle $[0, 2\pi[$. On a comme données d'entraînement $(x_i, \sin(x_i))_{i\in[1,500]}$. L'objectif est d'entraîner un réseau de neurones à approximer la fonction sinus sur cet intervalle. J'ai utilisé un réseau $(1, 16) \times (16, 16) \times (16, 1)$ avec tanh comme fonction d'activation. J'ai utilisé $\eta = 5.10^{-2}$ sur 2000 epochs, des batchs de taille 16 et la MSE comme fonction d'erreur.

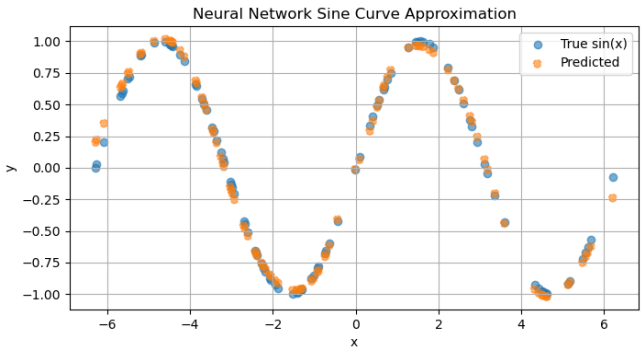

Figure 2: Approximation du sinus

3 Étapes du projet

Vous n'aurez pas le temps de tout faire, mais voici un exemple de ce qui pourrait être attendu :

1. Se familiariser avec la théorie.
2. Réfléchir à une structure modulaire du code (approche orientée objet).
3. Implémenter votre classe *Neural Network*.
4. Améliorer (minibatching, optimizers, initilisation, regularisation, dropouts, serialization, autres fonction d'activation, etc.).
5. Essayer votre réseau sur différents problèmes de régressions (sine, Insurance predictions).
6. Essayer des problèmes de classification (XOR, IRIS, MNIST)