Réalisation d'un Perceptron Multi-Couches

L2 CUPGE option Informatique

année 2019–2020

1 Présentation

Le but de ce projet est de réaliser un réseau de neurones de type *Perceptron Multi-Couche* (PMC). C'est un des types de réseaux utilisés en intelligence artificielle, et le *deep learning* concerne des réseaux structurés (sans tous les liens possibles entre les neurones d'une couche et ceux de la suivante) de plusieurs dizaines de couches. Pour avoir des temps de calcul raisonnables, on se limitera dans l'application à des réseaux à deux *couches cachées* (les couches entre l'entrée et la sortie), comme dans la Fig. 3.

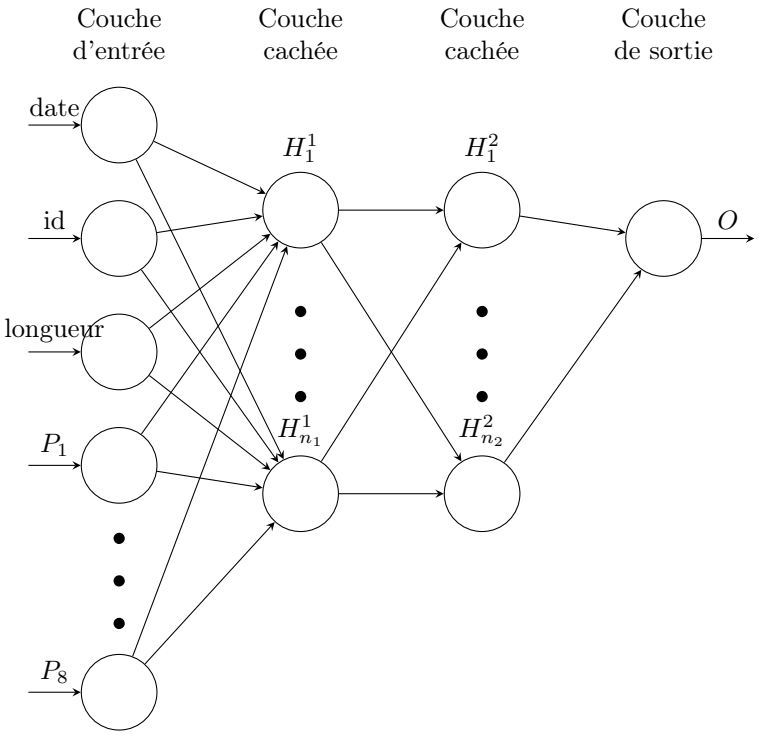

FIGURE 1 – Architecture proposée pour l'application

1.1 Réseau de neurones

En théorie, un neurone est défini par :
— des *synapses* qui le connectent à d'autres neurones ;
— une *activation* qui mesure à quel point un neurone est « excité ».
Dans les PMC, l'activation est mesurée par un `float`, positif ou négatif. Elle est une fonction f d'une somme pondérée de l'activation des neurones de la couche précédente qui sont connectés à ce neurone pour les neurones qui ne sont pas sur la couche d'entrée. On considère pour ce projet des *PMC uniformes* dans lesquels la fonction f est la même pour tous les neurones.

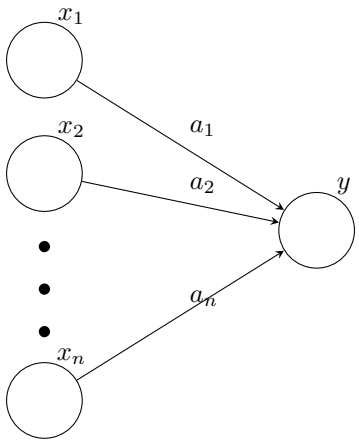

FIGURE 2 – Représentation d'un neurone connecté avec n neurones de la couche précédente. Pour un PMC uniforme avec une fonction d'activation f, on a $y = f(\Sigma_{i=1}^{n} a_i \cdot x_i)$.

1.2 Calculer avec un PMC

Un PMC effectue des calculs de la manière suivante :
— on donne une activation aux neurones de la couche d'entrée ;
— on calcule ensuite itérativement l'activation des neurones de la couche suivante (dans l'application, d'abord de la couché caché 1, puis de la couche cachée 2, puis de la sortie) jusqu'à arriver à la couche de sortie.
— le résultat du calcul du PMC est le vecteur (dans l'application, ce vecteur est de dimension 1) des activations des neurones de la couche de sortie.

En pratique, on représente l'activation d'une couche de n neurones par un tableau de n `float`. Les coefficients des synapses utilisés pour pondérer la somme (les a_i de la Fig 2) sont stockés dans une matrice (voir Fig. 4 pour un exemple simple). Dans l'application, on utilise 3 matrices :
— une entre la couche d'entrée et la première couche cachée ;
— une entre les deux couches cachées ;
— une de la seconde couche cachée vers la couche de sortie.

1.3 Calcul du résultat

En théorie, un neurone est défini par :

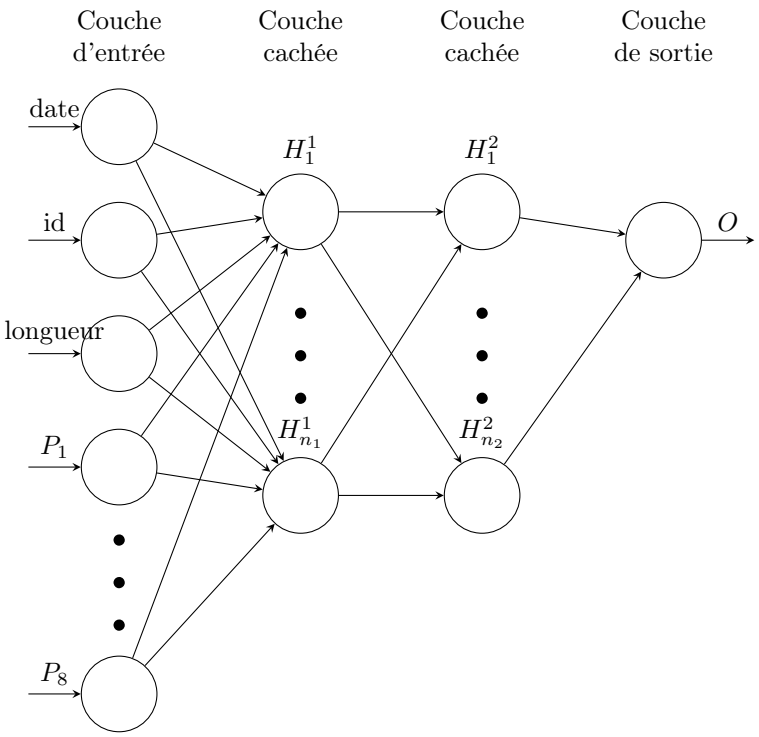

FIGURE 3 – Architecture proposée pour l'application

— des *synapses* qui le connectent à d'autres neurones ;
— une *activation* qui mesure à quel point un neurone est « excité ».

Dans les PMC, l'activation est mesurée par un `float`, positif ou négatif. Elle est une fonction f d'une somme pondérée de l'activation des neurones de la couche précédente qui sont connectés à ce neurone pour les neurones qui ne sont pas sur la couche d'entrée. On considère pour ce projet des *PMC uniformes* dans lesquels la fonction f est la même pour tous les neurones.

La matrice entre une couche de n neurones et une couche de m neurones est de taille $m \times n$. Si l'activation des neurones d'une couche est représentée par le vecteur ligne$\vec{u}$ et que la matrice entre cette couche et la suivante est M, on calcule l'activation des neurones de la couche suivante :

— en calculant $\vec{v} = \vec{u} \cdot M$;
— en posant $\vec{v} = (v_1, \ldots, v_m)$, le vecteur ligne d'activation des neurones de la couche suivante est $(f(v_1), \ldots, f(v_m))$.

On dit que le calcul *propage* les valeurs d'entrée à travers le réseau par des multiplications successives de matrices (calcul par **propagation**).

1.4 Descente de gradient

Le résultat dépend entièrement des coefficients qui sont stockés dans la matrice. L'utilité des réseaux de neurones est qu'on dispose d'une méthode qui permet de rapprocher le résultat du calcul. Pour cela, on a besoin de connaître, étant donnée une entrée e, le résultat attendu r_{att} (en général, c'est un vecteur,

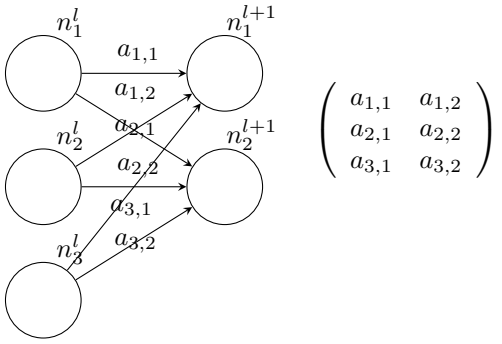

FIGURE 4 – Traduction en matrice des coefficients entre la couche l et la couche $l+1$.

dans l'application, il s'agit juste d'une valeur 1 ou -1.) Le but de l'algorithme d'apprentissage par rétro-propagation est de minimiser la distance entre ce résultat attendu et le résultat produit par le PMC avec les cœfficients actuels, r.

Pour simplifier les calculs, on considère le carré de cette distance divisé par 2. Cette fonction est monotone par rapport à la distance, et a l'avantage de simplifier les calculs. On veut changer les poids pour minimiser cette distance. Avec un neurone sur la couche de sortie, cette distance peut être exprimée en fonction de les activations (x_i pour le $i$ème neurone) des neurones sur la couche précédente et du coefficient a_i entre le $i$ème neurone et le neurone de sortie :

$$\begin{aligned} E &= \tfrac{1}{2} \cdot (r_{\text{att}} - r)^2 \\ &= \tfrac{1}{2} \cdot (r_{\text{att}} - f(\Sigma_{i=1}^n a_i \cdot x_i))^2 \end{aligned}$$

Pour essayer de rendre cette erreur plus petite, pour chaque neurone i de la couche précédente on la dérive par rapport à a_i. Cette dérivée nous indiquera comme modifier a_i pour minimiser la distance. De manière générale, on calcule une dérivée de l'erreur dans chaque neurone pour chaque cœfficient qui contribue à l'activation de ce neurone. Par la suite, on utilisera des indices l pour numéroter les couches ($l = 0$ pour la couche d'entrée), et des indices i et j pour désigner des neurones au sein d'une couche. Pour simplifier les expressions, comme exemple, on va commencer par dériver l'erreur finale (la dernière couche ayant un seul neurone, on peut se passer d'indice) par rapport au cœfficient a_{17} de la contribution du 17ème neurone de la seconde couche cachée (un indice en moins, il n'en reste plus que dans les sommes) :

$$\begin{aligned} \frac{\partial E}{\partial a_{17}} &= \frac{\partial(\frac{1}{2} \cdot (r_{\text{att}} - f(\Sigma_{i=1}^n a_i \cdot x_i))^2)}{\partial a_{17}} \\ &= \tfrac{1}{2} \cdot 2 \cdot (r_{\text{att}} - f(\Sigma_{i=1}^n a_i \cdot x_i)) \cdot \frac{\partial(r_{\text{att}} - f(\Sigma_{i=1}^n a_i \cdot x_i))}{\partial a_{17}} \\ &= (r_{\text{att}} - r) \cdot \frac{\partial(r_{\text{att}} - f(\Sigma_{i=1}^n a_i \cdot x_i))}{\partial a_{17}} \\ &= -(r_{\text{att}} - r) \cdot f'(\Sigma_{i=1}^n a_i \cdot x_i) \cdot \frac{\partial(\Sigma_{i=1}^n a_i \cdot x_i)}{\partial a_{17}} \\ &= -(r_{\text{att}} - r) \cdot f'(\Sigma_{i=1}^n a_i \cdot x_i) \cdot x_{17} \end{aligned}$$

La dernière ligne est parce qu'on dérive par rapport à a_{17}. Il faudrait faire ce

calcul pour tous les neurones de la couche cachée précédant la sortie, mais on voit que hormis le coefficient x_{17} à la fin de l'expression, les autres parties ne dépendent pas du neurone choisi dans la couche précédente. Pour le neurone j de la couche l, si l'*écart* par rapport au résultat attendu est e_j^l, on pose :

$$z_j^l = e_j^l \cdot f'(\underbrace{\Sigma_{i=1}^n a_i \cdot x_i}_{\substack{\text{somme des} \\ \text{entrées du} \\ \text{neurone}}})$$

En prenant, sur le neurone de sortie, $e = r_{\text{att}} - r$ comme déviation par rapport au résultat attendu, la dérivée est :

$$\frac{\partial E}{\partial a_{17}} = -z_{\text{sortie}} \cdot x_{17}$$

Localement, quand on fait varier le poids a_{17}, la distance au résultat attendu comme une fonction linéaire en la variable x_{17}, l'activation du neurone 17 de la dernière couche cachée. Plus généralement, si le neurone i de la couche $l+1$ a une erreur e_i^{l+1}, et que le neurone j de la couche l est connecté avec le poids $a_{j,i}$, alors :

$$\frac{\partial e_i^{l+1}}{\partial a_{j,i}} = -z_i^{l+1} \cdot x_j^l$$

Calculer toutes les dérivées partielles revient à calculer (dans $\mathbb{R}^{\#\text{neurones}}$) le gradient de la courbe qui décrit le carrée de la distance à la valeur voulue, et à suivre ce vecteur dans le sens opposé sur une petite distance, et on espère que cela diminuera la distance. D'où le nom de **descente de gradient**.

1.5 Rétro-propagation

Pour itérer, il faut calculer l'erreur sur le $j$ème neurone de la couche précédente. On pourra ainsi remonter les couches jusqu'à la couche d'entrée en associant à chaque neurone une erreur e_i^l à corriger. Pour faire ce calcul, on commence par estimer la différence entre la valeur de x_j^l lors du calcul et une valeur idéale qui minimiserait l'erreur dans la couche $l+1$. Pour cela, on dérive la somme des carrés des écarts dans la couche $l+1$ par rapport à x_j^l. Avec un calcul très proche du calcul précédent, on trouve :

$$\begin{aligned} e_j^l &= \Sigma_{k=1}^{n_{l+1}} \frac{\partial e_i^{l+1}}{\partial x_j^l} \\ &= \Sigma_{k=1}^{n_{l+1}} z_k^{l+1} a_{k,j} \end{aligned}$$

L'avantage d'avoir défini les variables z_k^l est que, si on a bien les yeux en face des trous, on remarque que l'écart à la couche l se calcule en multipliant le vecteur z^{l+1} par la transposée de la matrice des coefficients utilisée pour le calcul du réseau de neurone. Et on aura toujours $z_j^l = e_j^l \cdot f'(\text{somme des entrées})$.

En résumé, on propage l'écart sur la sortie vers l'entrée du réseau de neurones en faisant des multiplications par les transposées des matrices utilisées pour le calcul (on obtient alors les e_i^l), et en multipliant chaque valeur par la dérivée de la fonction d'activation (pour obtenir les z_i^l). C'est surtout à cause de la propagation de l'écart qu'on parle d'**apprentissage par rétro-propagation**.

1.6 Résumé

L'apprentissage se fait en 2 temps :

— on commence par corriger le coefficient $a_{j,i}$: graphiquement, on regarde la pente de E_i, et on tente d'aller (à une certaine vitesse) vers le bas. On considère le cas où la vitesse de progression est fixée à $\lambda = 0,001$. On corrige a_j en faisant :

$$a_{j,i} \quad \rightarrow \quad a_{j,i} + \lambda \cdot z_i^{l+1} \cdot x_j^l$$

— On propage les écarts en direction de la couche d'entrée par rétro-propagations :

$$\begin{array}{rcl} e_j^l & = & \Sigma_{i=1}^{n_{l+1}} a_{i,j} \cdot e_i^{l+1} \\ z_j^l & = & f'(x_j^l) \cdot e_j^l \end{array}$$

— On itère le calcul en remontant sur la couche d'entrée pour corriger les coefficients de toutes les matrices.

1.7 Les fonctions d'activation usuelles

On liste dans cette section les fonctions d'activation qui sont souvent utilisées, avec leur dérivée. Pour calculer ces fonctions en C, on peut utiliser la librairie `math.h` qui contient toutes les fonctions usuelles. Un des critère pour le choix de ces fonctions est qu'une fois le résultat de la fonction connu (c'est l'activation du neurone calculée pendant la phase de propagation), il est relativement simple de calculer la dérivée.

fonction	dérivée
$f(x) = \frac{1}{1+e^{-x}}$	$\frac{e^{-x}}{(1+e^{-x})^2} = e^{-x} \cdot (f(x))^2$
$\tanh(x) = \frac{1+e^{-2x}}{1-e^{-2x}}$	$1 - (\tanh(x))^2$
$\mathrm{ReLU}(x) = \max(0, x)$	1 si $x \geq 0$, 0 sinon

1.8 Autres ressources pour la description

Les pages Wikipédia en français sont assez pauvres sur le sujet, et n'entrent pas dans les détails. Essayez la version anglaise pour plus d'information ! Sinon, la section 1 de la description des PMC sur la page :

`http://penseeartificielle.fr/focus-reseau-neurones-artificiels-perceptron-multicouche/`

est assez bien faite. De même, la page Wikipédia :

`https://fr.wikipedia.org/wiki/Rétropropagation_du_gradient`

explique bien la technique que nous utilisons pour corriger les erreurs, mais utilise .

2 Travail à réaliser

2.1 Opérations matricielles

Un premier travail consiste à implémenter une structure matrice contenant :
- le nombre de lignes et de colonnes de la matrice ;
- le tableau des coefficients de la matrice.

Il est demandé d'écrire les fonctions permettant d'utiliser cette structure :
- création d'une matrice à partir du nombre de lignes et de colonnes. Plusieurs fonctions peuvent être écrites pour soit initialiser les coefficients de la matrice à 0, soit les initialiser par des nombres aléatoires ;
- demander une matrice à un utilisateur et afficher une matrice ;
- lire et écrire des matrices dans des fichiers ;
- additionner et multiplier des matrices, et multiplier un vecteur par une matrice.

Vous pouvez écrire des fonctions supplémentaires, et vous inspirer du TP !

2.2 Réseau de neurones

Un réseau de neurones est un tableau de n matrices. Pour les calculs (voir plus haut) il est pratique de stocker pour chaque couche l de n_l neurones :
- le tableau des sommes $\Sigma_{k=1}^{n_{l-1}} a_{k,i} \cdot x_k^{l-1}$
- le tableau des activations $f(\Sigma_{k=1}^{n_{l-1}} a_{k,i} \cdot x_k^{l-1})$

Il faut créer une structure ayant toutes ces informations, et écrire des fonctions :
- permettant de calculer la sortie d'un réseau de neurones en fonction de son entrée ;
- permettant d'apprendre les poids corrects pour minimiser l'erreur.

Fonction d'activation. Il est conseillé de commencer par choisir une fonction d'activation (et donc sa dérivée) fixée par le programme. Si vous avez le temps, il est aussi possible de modifier la structure « réseau de neurones » pour inclure ces deux fonctions, voire d'avoir des réseaux de neurones non-uniforme avec des fonctions différentes suivant les couches.

2.3 Application

L'apprentissage avec un réseau avec 2 couches cachées a déjà été fait par l'Institut Fraunhöfer de Darmstadt sur les messages du bus CAN (le réseau interne permettant la communication entre les différents composants des automobiles). Le fichier (choisir ensuite `CAN-intrusion`) est disponible [1] à partir de la page suivante :

`http://ocslab.hksecurity.net/Dataset/CAN-intrusion-dataset`

1. pas au moment de l'écriture du sujet, mais en théorie

Chaque ligne contient un message et un indicateur (le dernier nombre sur la ligne, qui vaut 1 ou -1). Le but de l'apprentissage est d'apprendre ce dernier nombre (qui est donc la sortie attendue) en fonction des 11 nombres précédents (qui sont donc les entrées).

Pour cela, il faut itérer (une centaine de fois) l'algorithme d'apprentissage, avec à chaque fois un apprentissage pour chaque ligne du fichier Vous devez créer un programme spécifique qui va faire cet apprentissage.

3 Considérations administratives

D'ici la fin du mois de novembre, nous verrons :
- la définition de nouveaux types, incluant les structures ;
- la séparation d'un programme en différents *modules*, qui sont des fichiers contenant les fonctions relatives à un type ou à un ensemble de fonctionnalités ;
- l'utilisation de `Makefile` pour automatiser la compilation de programmes ;
- en TP, l'utilisation de tests pour vérifier les fonctions d'un module.

Le but de ce projet n'est pas de résoudre un nouveau problème, mais que vous appreniez en pratique à utiliser ces informations. Vous serez d'abord noté sur l'utilisation de ce qu'on aura vu.

Groupes. Les projets sont écrits en binômes. Les trinômes et les monômes doivent être évités, sauf accord de votre enseignant de TP (avec une préférence pour les trinômes).

Écriture du projet. Vous devez écrire votre projet sur un répertoire git privé et hébergé sur github. Vous devrez donner l'accès aux membres du groupe et à votre enseignant de TP. **L'utilisation de `git` n'est pas optionnelle !** En particulier, en cas de gros déséquilibre dans les contributions, les notes des participants pourront être modulées (par défaut, la note est pour le projet).

Documentation. Tout programme doit être documenté ! Vous devrez donc rendre à la fin du projet deux documents :
- un document **pdf** (ni word, ni txt, ni autre) décrivant les différents modules, et les fonctions dans ces modules ;
- un document (du texte simple écrit dans un bloc note ou un fichier markdown) appelé `README` (.txt ou .md) dans lequel vous décrirez comment compiler les fichiers sources, comment tester les modules, et comment utiliser le programme.

Pièges. Nous avons sciemment piégé cet énoncé en introduisant ce qui pourrait ressembler à des erreurs. Pour vous tromper encore plus, il se peut même qu'il n'y ait pas d'erreurs ! Par pure bienveillance, nous corrigerons sur l'énoncé toute erreur qui nous signalée par une personne.

Date de fin de projet. Le projet doit être rendu le **6 décembre 2019**. Pour le rendre, vous n'avez rien à faire, c'est l'enseignant de TP qui clonera votre dépôt.