R5.A.12 – réseaux de neurones

Partie 1 – construction du réseau de neurones

L'objectif de cette première partie est de construire un réseau de neurones en c++, sans bibliothèques tierces. Les livrables attendus sont : le code c++ (classes des différents objets) ainsi qu'une fonction `main` qui implémente les exemples demandés.

Un neurone est un objet qui accepte en entrée un vecteur de n valeurs réelles $(x_1, x_2, ..., x_n)$ et fournit en sortie une valeur réelle y définie par :

$$y = H(a_1x_1 + a_2x_2 + ... + a_nx_n + b),$$

où les coefficients a_i sont les n poids du neurone et b le biais. Ce sont des constantes réelles pour un neurone donné. La fonction H est appelée *fonction d'activation.*

(1) Créer une classe abstraite représentant une fonction d'activation ainsi qu'une implémentation de la fonction d'activation ReLU définie par :

$$ReLU(x) = \begin{cases} 0 & \text{si } x < 0 \\ x & \text{si } x \geq 0 \end{cases}$$

Dans la suite, il sera nécessaire de connaître la dérivée de la fonction d'activation : ajouter dans la classe abstraite et dans l'implémentation, l'expression de la dérivée de la fonction d'activation pour une valeur x donnée.

(2) Créer une classe représentant un neurone complet incluant les poids, le biais, la définition d'une fonction d'activation. Par défaut, le biais et les poids seront nuls.

(3) Ajouter une fonction d'évaluation qui calcule la sortie du neurone en fonction d'une entrée donnée. Tester l'implémentation avec le code suivant : neurone avec 2 entrées (valeurs : $(4, -1)$) dont les poids sont 2 et 3, le biais 0,5 et la fonction d'activation ReLU

(4) Créer une classe représentant une couche de neurones. Une couche de neurones est un vecteur contenant m neurones. Elle prend en entrée un vecteur de n valeurs réelles $(x_1, x_2, ..., x_n)$, appliqué à chacun des neurones de la couche (avec n poids). En sortie, elle produit un vecteur de m valeurs réelles $(y_1, y_2, ..., y_m)$.

Tester l'implémentation avec le code suivant : couche de 2 neurones – activation ReLU – avec 3 poids (1, 3, -1 et -1, 0, -2) et biais 0.1 et 0.3. Donnée en entrée : (4, -1, 0).

(5) La dernière étape consiste à créer une classe représentant un réseau de neurones : un réseau est composé de couches de neurones connectées entre elles. Dans ce TP, on ne considère que des couches *complètement connectées* : pour une couche i, chaque neurone est connecté à tous les neurones de la couche suivante $i + 1$ (le nombre de poids de la couche $i + 1$ étant égal au nombre de neurones de la couche i).

Le réseau accepte en entrée un vecteur de taille n correspondant à la taille en entrée de la couche 1 (le nombre de poids des neurones de la couche 1). La sortie de la couche 1, de taille p_1 doit correspondre à la taille en entrée de la couche 2, etc. La sortie du modèle est un vecteur de taille m égale à la taille de la sortie de la dernière couche (donc égale au nombre de neurones de la dernière couche).

Implémenter une fonction d'évaluation du réseau complet qui calcule la sortie (vecteur de taille m) en fonction de l'entrée (vecteur de taille n).

Tester l'implémentation à l'aide d'un réseau de 2 couches. Taille de l'entrée : 8, puis couche à 4 neurones, puis couche à 2 neurones. Activation ReLU et données en entrée : (4, -1, 0, 2, 5, 1, 0, 4).

On prendra pour paramètres :

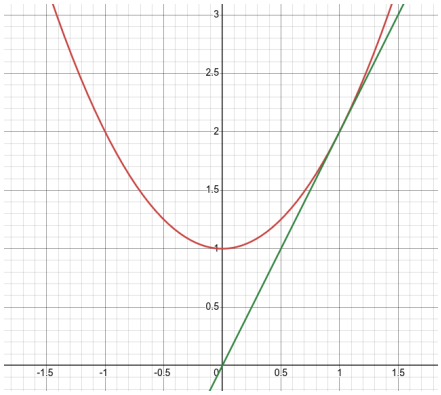
- Couche 1, neurone 1 : poids $(1\ ;\ 1\ ;\ -1\ ;\ 2\ ;\ -1\ ;\ 0\ ;\ 1\ ;\ 1)$, biais $-0,1$
- Couche 1, neurone 2 : poids $(0\ ;\ 0\ ;\ 1\ ;\ 1\ ;\ 1\ ;\ 3\ ;\ 2\ ;\ 0)$, biais $0,2$
- Couche 1, neurone 3 : poids $(1\ ;\ -1\ ;\ 1\ ;\ -1\ ;\ 0\ ;\ 0\ ;\ 0\ ;\ 1)$, biais $1,2$
- Couche 1, neurone 4 : poids $(2\ ;\ 2,5\ ;\ 3\ ;\ -2\ ;\ -2,5\ ;\ 1\ ;\ 0\ ;\ 0)$, biais $-2,5$
- Couche 2, neurone 1 : poids $(-1,3\ ;\ 1,7\ ;\ 0,7\ ;\ 0,4)$, biais 0

- Couche 2, neurone 2 : poids $(0,35\ ;\ -0,95\ ;\ 1,1\ ;\ 0,2)$, biais $1,2$

Partie 2 – descente de gradient

L'objectif de la descente de gradient est de déterminer un minimum (en général local) d'une fonction, en suivant la direction de plus la grande pente (négative), donnée par l'opposé du gradient. Les livrables attendus sont les codes c++ répondant aux différents exercices.

(1) Dans ce premier exemple, on cherche à déterminer le minimum x^* de la fonction $y = x^2 + 1$ (courbe rouge). La solution recherchée est $x^* = 0$, mais nous allons la déterminer à l'aide de la méthode de descente de gradient en c++.



La méthode est la suivante :

1. On choisit une valeur initiale aléatoire (ou estimée), par exemple $x_0 = 1$.
2. On calcule le gradient de la fonction au point x_0. Dans le cas d'une fonction d'une variable comme ici, le gradient se réduit au calcul de la dérivée. La dérivée de f s'écrit $f'(x) = 2x$. Au point x_0 elle vaut $f'(x_0) = 2 \times x_0 = 2 \times 1 = 2$. Localement, la fonction f est croissante et de pente $+2$ (comme sa tangente – droite verte).
3. Pour aller dans la direction d'un minimum local, il suffit de suivre la direction opposée au gradient (ici le gradient est le vecteur $(+2)$, la direction opposée est donnée par le vecteur (-2) – direction des x négatifs) : en choisissant $x_1 < x_0$ on obtiendra $f(x_1) < f(x_0)$ (car la fonction est ici croissante).

On applique la relation suivante :

$$x_{n+1} = x_n - \alpha f'(x_n).$$

Cette relation donne une nouvelle valeur x_{n+1} du minimum à partir de la valeur précédemment estimée x_n en se déplaçant dans la direction opposée du gradient (ici le vecteur $(+2)$). Le déplacement dépend :

- du gradient – ici $f'(x_n)$: plus la fonction varie vite, plus on peut se déplacer loin. Lorsque l'estimation est proche d'un minimum, le gradient s'approche de zéro, et le déplacement tend à devenir nul.
- d'un coefficient α, appelé *taux d'apprentissage* qui permet de contrôler la vitesse à laquelle les itérés successifs se mettent à jour. Un taux d'apprentissage élevé peut permettre de converger plus vite, mais conduire à un algorithme instable ; à l'inverse, un taux d'aprentissage faible, sera plus lent à converger, mais sera plus stable (mais pourra aussi rester dans un minimum local non optimal).

Le minimum (local) est considéré comme atteint lorsque les itérés successifs ne varient quasiment plus, c'est-à-dire :

$$|x_{n+1} - x_n| < \varepsilon,$$

où ε est un réel (petit).

Écrire un algorithme en c++ permettant de calculer un minimum local pour une fonction d'une variable à partir d'une valeur initiale estimée. Tester l'application sur la fonction f ci-dessus, avec un taux d'apprentissage de $\alpha = 0,1$ et un critère d'arrêt $\varepsilon = 10^{-8}$.

(2) Dans le deuxième exemple, on se donne un ensemble de valeurs réelles $x_1, x_2, ..., x_n$ et on cherche la valeur x^* réelle *la plus proche* de toutes ces valeurs. Par *plus proche* de x_i, on entend la valeur de x^* qui minimise la distance – ou pénalité – suivante :

$$(x_i - x^*)^2$$

Cette pénalité est d'autant plus grande que x^* est éloigné de x_i. En faisant la moyenne des pénalités pour toutes les valeurs x_i, on peut définir une fonction d'erreur globale :

$$E(x^*) = \frac{1}{n}\sum_{i=1}^{n}(x_i - x^*)^2.$$

L'objectif est ici de trouver x^* qui minimise cette fonction d'erreur, à l'aide de la descente de gradient. Calculer la dérivée $E'(x^*)$ et implémenter un algorithme en c++ de descente de gradient avec un taux d'apprentissage de $\alpha = 0,1$ et un critère d'arrêt $\varepsilon = 10^{-8}$, pour les données $(1, 3, 0, 5, 7, 2, 9)$.

(3) Dans ce troisième exemple, on cherche l'équation de la droite de régression linéaire d'équation $y = ax + b$ qui approche le mieux un ensemble de n points $(x_i; y_i)$.

Les inconnues sont les réels a et b. Pour une entrée x_i, la droite approxime la sortie par :

$$y_i^* = ax_i + b.$$

L'erreur commise pour le point i s'écrit de manière similaire à l'exemple précédent :

$$(y_i - y_i^*)^2 = (y_i - ax_i - b)^2.$$

La fonction d'erreur globale peut s'écrire (elle dépend des 2 inconnues a et b) :

$$E(a, b) = \frac{1}{n}\sum_{i=1}^{n}(y_i - ax_i - b)^2.$$

En calculant les dérivées partielles de E par rapport aux variables a et b, on peut former le gradient de la fonction E (qui est un vecteur à 2 composantes) :

$$\nabla E = \begin{pmatrix} \frac{\partial E}{\partial a} \\ \frac{\partial E}{\partial b} \end{pmatrix}$$

De manière identique à l'exemple précédent, si l'on se donne un estimé initial (a_0, b_0), alors on peut améliorer l'estimation d'un minimum local par :

$$(a_1, b_1) = (a_0, b_0) - \alpha\nabla E.$$

Proposer une implémentation c++ permettant de calculer a et b par descente de gradient pour les données : $(0; 1)$, $(1; 3)$, $(2; 2)$, $(4; 3)$, $(6; 5)$ et un taux d'apprentissage de $\alpha = 0,001$.

(4) Comme dernier exemple, considérons un neurone à 2 entrées (et donc 2 poids a et b) et 1 biais c. Les réels a, b et c ne sont pas connus et doivent être déterminés.

Le neurone admet en entrée un ensemble de données sous la forme de vecteurs (x_i, y_i) et donne en sortie le réel z_i respectivement. La fonction d'activation est ReLU.

Écrire la fonction d'erreur globale. Déterminer les dérivées partielles de cette fonction par rapport aux inconnues a, b et c. Proposer un algorithme en c++ pour déterminer des valeurs optimales pour a, b et c par descente de gradient pour les données $((0; 0); 0)$, $((0; 1); 0, 5)$, $((1; 0); 0, 6)$, $((1; 1); 1)$.

Partie 3 – rétropropagation du gradient

Dans cette partie, les livrables attendus sont : démonstrations demandées (sur feuille ou document informatique) et codes c++.

Si l'on considère un réseau de neurones formé par les couches successives C_1, C_2, ..., C_n transformant un vecteur en entrée X_k (de longueur L_1) en un vecteur de sortie Y_k (de longueur L_s), alors on peut écrire que le réseau est équivalent à une fonction f telle que :

$$Y_k = f(X_k).$$

Pour cette observation k, la valeur attendue en sortie (qui est une donnée d'apprentissage) est notée Y_k^*. L'erreur commise par le réseau pour cette donnée k peut s'écrire (en faisant le choix d'une erreur de type norme 2) :

$$E_k(\{p\}) = \|Y_k - Y_k^*\|^2.$$

L'erreur dépend de la donnée d'apprentissage k, (X_k, Y_k^*), mais aussi des paramètres du réseau (poids et biais), ici notés $\{p\}$.

Pour l'ensemble des N données d'apprentissage, l'erreur moyenne est donnée par la moyenne des erreurs de chaque donnée :

$$E(\{p\}) = \frac{1}{N}\sum_{k=1}^{N} E_k(\{p\}) = \frac{1}{N}\sum_{k=1}^{N}\|Y_k - Y_k^*\|^2.$$

L'erreur globale est une fonction de $\mathbb{R}^M \to \mathbb{R}$, où M est le nombre de paramètres du réseau, potentiellement très grand. La phase d'apprentissage d'un réseau de neurones consiste à trouver les paramètres qui vont minimiser au mieux cette fonction d'erreur.

Si l'on se donne des paramètres du réseau $\{p_0\}$, on peut déterminer des paramètres plus optimaux en appliquant la descente du gradient. Pour chaque paramètre p, on peut donc écrire la relation :

$$p_1 = p_0 - \alpha\frac{\partial E}{\partial p}.$$

L'application de la descente du gradient pour un réseau de neurones nécessite donc le calcul de M dérivées partielles.

(1) Montrer que pour tout paramètre p du réseau et la fonction d'erreur choisie ci-dessus :

$$\frac{\partial E}{\partial p} = \frac{2}{N}\sum_{k=1}^{N}\sum_{i=1}^{L_s}\left(Y_{k,i} - Y_{k,i}^*\right)\frac{\partial Y_{k,i}}{\partial p},$$

où $Y_{k,i}$ est la composante i du vecteur de sortie obtenu pour la donnée k.

(2) $Y_{k,i}$ correspond au vecteur de sortie du réseau, soit en sortie de la couche n. Dans la suite, on renomme cette valeur en $Y_{n,k,i}$. On peut ainsi exprimer la sortie de n'importe quelle couche du réseau, par exemple la couche j : $Y_{j,k,i}$. L'objectif ici est de chercher la relation de récurrence qui existe entre 2 couches successives $j-1$ et j :

$$\frac{\partial Y_{j-1,k,i}}{\partial p} \text{ et } \frac{\partial Y_{j,k,i}}{\partial p}.$$

En exprimant $Y_{j,k,i}$ en fonction des $Y_{j-1,k,i}$ (et donc de la fonction d'activation H, du biais $b_{j,i}$ et des poids $a_{j,i,r}$ du neurone i de la couche j), montrer que :

$$\frac{\partial Y_{j,k,i}}{\partial p} = H'\left(b_{j,i} + \sum_{r=1}^{L_j} a_{j,i,r}Y_{j-1,k,i}\right) \times \frac{\partial}{\partial p}\left(b_{j,i} + \sum_{r=1}^{L_j} a_{j,i,r}Y_{j-1,k,i}\right).$$

(3) Un premier cas peut apparaître : le paramètre p avec lequel on calcule la dérivée partielle peut être un des paramètres du neurone (soit le biais $b_{j,i}$, soit l'un des poids $a_{j,i,r}$). Dans ce cas, les $Y_{j-1,k,i}$ ne dépendent pas de p. Simplifier l'expression précédente pour chacun de ces cas.

(4) Un deuxième cas peut apparaître : le paramètre p avec lequel on calcule la dérivée partielle n'est pas un des paramètres du neurone, mais d'un autre neurone de la même couche. Simplifier l'expression précédente pour de cas.

(5) Un troisième cas peut apparaître : le paramètre p avec lequel on calcule la dérivée partielle n'est pas un des paramètres du neurone, mais d'un autre neurone d'une autre couche (précédente). Dans ce cas, les $Y_{j-1,k,i}$ dépendent de p. Simplifier l'expression pour ce cas et établir la relation de récurrence recherchée :

$$\frac{\partial Y_{j,k,i}}{\partial p} = H' \left(b_{j,i} + \sum_{r=1}^{L_j} a_{j,i,r} Y_{j-1,k,i} \right) \times \sum_{r=1}^{L_j} a_{j,i,r} \frac{\partial Y_{j-1,k,i}}{\partial p}.$$

(6) Implémenter en c++ une fonction qui calcule la dérivée partielle de l'erreur en fonction d'un paramètre du réseau. Un paramètre pourra être défini par (1) la couche j concernée, (2) le neurone i concerné et (3) un identifiant (par exemple 0 pour le biais, puis de 1 à N pour les N poids du neurone).

La fonction à implémenter doit l'être au niveau de chaque couche. La valeur retournée sera soit la valeur de la dérivée pour le neurone concerné par le paramètre, soit un appel récursif remontant le réseau sinon.

L'appel final lors de la phase d'apprentissage sera réalisé au niveau de la dernière couche, pour chaque neurone de sortie.

Partie 4 – applications

(1) On considère un réseau de neurones composé d'une seule couche et d'un seul neurone. Ce neurone ne possède d'un seul poids. La fonction d'activation est ReLU. Le réseau possède 2 paramètres : le poids a et le biais b du neurone.

Implémenter la phase d'apprentissage du réseau en appliquant la descente de gradient sur les données suivantes :

$$(1; 2), (2; 2, 5), (3; 3), (4; 5, 5), (5; 5, 5)$$

Montrer que le neurone seul se comporte comme une régression linéaire pour ces données (et expliquer pourquoi).

(2) On propose le même exercice mais avec 2 couches d'un neurone (avec donc 4 paramètres : 2 poids et 2 biais). Montrer que l'erreur ne descend pas vers des valeurs faibles avec l'implémentation proposée.

Calculer à la main les expressions des gradients (pour la première itération) pour les 4 paramètres du réseau et montrer que tous sauf 1 resteront toujours nuls, avec des poids initiaux nuls.

Proposer une modification de l'implémentation afin de corriger le problème des gradients nuls.

(3) Importer le jeu de données `dataset1.csv` : ce fichier CSV contient en première colonne l'année (entre 1880 et 2024) et en seconde colonne l'anomalie de température de la Terre (écart entre la moyenne mondiale de l'année et la moyenne mondiale sur la période de référence – de 1951 à 1980, en degrés Celsius).

Utiliser un réseau avec une couche et un seul neurone (sans fonction d'activation – cela nécessite d'ajouter une implémentation de fonction d'activation de la forme $H(x) = x$) et lancer un apprentissage afin d'obtenir une régression linéaire sur les données. Afficher les données et la sortie du modèle dans un graphique et montrer que la régression linéaire est insuffisante pour représenter l'anomalie de température.

(4) Reprendre l'exercice précédent avec un réseau de neurones modifié : utiliser 3 couches de N neurones avec activation ReLU, puis une dernière couche à 1 neurone sans activation. On prendra pour valeur de N : 10 neurones.

Lancer l'apprentissage et vérifier sur un graphique que le modèle prédit plus fidèlement les variations climatiques relevées. On pourra utiliser une erreur maximale de $0,01$ et un taux d'apprentissage de $0,01$