

Informatique

Intelligence artificielle

Réseau de neurones

Inspiré du sujet CCINP PSI Modélisation 2024

Température dans une turbine

Contexte de l'étude

Ce sujet s'appuie sur le sujet du concours CCINP PSI Modélisation 2024, dont le sujet complet est fourni en annexe. Il porte sur l'optimisation d'un correcteur de facteur de puissance (CFP), un convertisseur statique intercalé entre le réseau électrique et une installation domestique. Son rôle est de forcer le courant absorbé à être sinusoïdal et en phase avec la tension du réseau, réduisant ainsi les pertes en ligne. C'est un enjeu classique en électrotechnique et en gestion de l'énergie.

Le CFP est piloté par un paramètre Δ . Plus Δ est petit, plus les pertes en ligne diminuent, mais plus le transistor de commutation chauffe. Il existe donc une valeur optimale Δ_{opt} qui minimise les pertes totales (ligne + transistor), et celle-ci dépend de la configuration de l'installation (appareils allumés ou éteints). Déterminer Δ_{opt} analytiquement pour toutes les combinaisons possibles est difficile, c'est pourquoi on fait appel à l'intelligence artificielle.

Ce sujet est structuré en trois grandes phases, typiques d'une démarche d'ingénierie numérique :

- Modélisation du réseau de neurones : on construit et on comprend l'architecture d'un perceptron multicouche (structure, fonction d'activation sigmoïde, représentation matricielle).
- Implémentation Python : on code soi-même les phases d'inférence et de rétropropagation du gradient, ce qui permet de comprendre concrètement l'algorithme d'apprentissage supervisé.
- Validation et analyse : on entraîne le réseau sur des données réelles, on évalue ses performances et on discute de ses limites, notamment la question du surapprentissage et du manque de données.

En fin de sujet, le même réseau de neurones est appliqué à un nouveau contexte industriel : la redondance de capteurs de température dans une turbine à gaz, ce qui illustre la généralisation de l'approche et son transfert à d'autres problèmes de l'ingénieur.

Contexte du sujet de concours

Pour toute la première partie physique du sujet de concours CCINP PSI MODELISATION 2024, je tente ici d'en résumer les points clés.

Un correcteur de facteur de puissance est introduit dans le sujet par l'association d'un pont de Graetz :

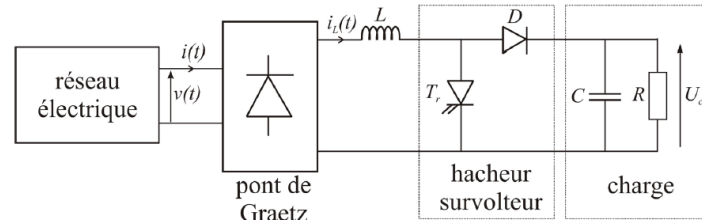


Figure 7 - Circuit de puissance du correcteur de facteur de puissance

Avec $i_{L,cons}(t) = |\hat{I} \cos(2\pi ft)|$, le courant dans la bobine qui est associé à un courant de ligne sinusoïdal de forme $i(t) = \hat{I} \cos(2\pi ft)$ avec $\hat{I} > 0$ et en phase avec la tension du réseau $v(t) = \hat{V} \cos(2\pi ft)$. Le principe de ce correcteur de facteur de puissance est de commander le transistor T_r pour que le courant d'intensité $i_L(t)$ soit le plus proche possible de $i_{L,cons}(t)$. Pour cela, on dispose :

- D'une tension de consigne $v_{i_{L,cons}}(t)$ telle que $v_{i_{L,cons}} = k i_{L,cons}$ avec $k = 1,0 \text{ V} \cdot \text{A}^{-1}$
- D'une tension $v_{i_{L,mes}}$ image du courant d'intensité i_L telle que $v_{i_{L,mes}} = k i_L$

La loi de commande adoptée est, à tout instant, construite à partir d'un paramètre $\Delta > 0$ comme suit :

- Si $|v_{i_{L,cons}} - v_{i_{L,mes}}| < \Delta$, le transistor est inchangé
- Si $(v_{i_{L,mes}} - v_{i_{L,cons}}) > \Delta$, le transistor passe à l'état bloqué
- Si $(v_{i_{L,mes}} - v_{i_{L,cons}}) < -\Delta$, le transistor passe à l'état passant

Ce correcteur de facteur de puissance est utilisé pour minimiser les pertes en lignes engendrées par l'installation électrique d'un petit studio composé d'un ensemble (box Internet/télévision, système d'éclairage, radiateur électrique, chauffe-eau) :

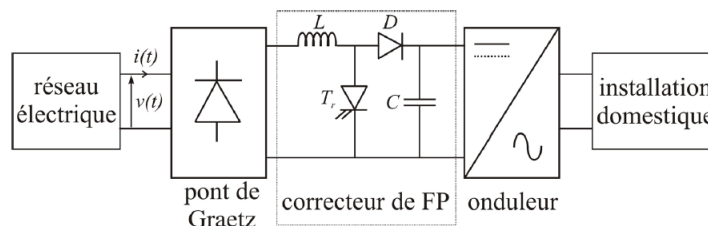


Figure 14 - Correcteur de facteur de puissance utilisé pour une installation domestique

La tension aux bornes du condensateur C est placée en entrée de l'association d'un onduleur de tension et de l'installation électrique du studio. Une boucle de régulation, non étudiée ici, permet de réguler cette tension à une valeur constante, de sorte que le fonctionnement du correcteur de facteur de puissance est identique au précédent.

Les données

Le dispositif étudié permet de réduire les pertes en ligne engendrées par l'absorption d'un courant impulsionnel. Celles-ci tendent vers leur valeur minimale lorsque le paramètre Δ tend vers 0. Cependant, cette diminution se fait au prix d'une augmentation de la fréquence de commutation moyenne f_{cm} et donc des pertes au sein du transistor T_r . Si la puissance dissipée dans T_r ne peut pas être exploitée, il est alors nécessaire de rechercher une valeur du paramètre Δ correspondant à une minimisation des pertes totales : pertes en ligne + pertes dans T_r .

Chacun des quatre éléments de l'installation présente 3 modes de fonctionnement. Un niveau x_i valant 0, 1 ou 2 est associé à chacun des modes. Le tableau 2 donne la correspondance entre le niveau x_i et le mode de fonctionnement. Certaines puissances consommées sont également données.

	box-TV x_1	éclairage x_2	radiateur x_3	chauffe-eau x_4
$x_i = 0$	éteint 0 W	éteint 0 W	éteint 0 W	éteint 0 W
$x_i = 1$	veille 100 W	puissance moitié 50 W	puissance moitié	maintient à température 50 W
$x_i = 2$	allumé 200 W	puissance maximale 100 W	puissance maximale	puissance maximale

Tableau 2 - Modes de fonctionnement

La valeur du paramètre Δ permettant de minimiser les pertes totales, notée Δ_{opt} , a été évaluée empiriquement pour 14 combinaisons entre les différents modes de fonctionnement des quatre éléments de l'installation électrique étudiée. Ces valeurs sont présentées dans le tableau 3. Le paramètre Δ s'exprime en volts et est positif. Lorsque la valeur -1 est indiquée pour Δ_{opt} , cela signifie que pour minimiser les pertes totales, il est préférable de désactiver le correcteur de facteur de puissance et de connecter l'installation directement au réseau électrique.

	box-TV x_1	éclairage x_2	radiateur x_3	chauffe-eau x_4	$\Delta_{opt}(V)$
Données d'entraînement	0	0	0	1	-1
	0	0	1	1	-1
	0	0	2	0	-1
	0	0	0	2	-1
	0	0	1	2	-1
	0	1	0	0	0,18
	2	2	0	0	0,32
	2	2	2	0	0,49
Données test	2	2	2	2	-1
	0	0	1	0	-1
	0	0	2	1	-1
	0	0	2	2	-1
	1	0	0	0	0,22
	2	2	1	0	0,42

Tableau 3 - Valeurs optimales du paramètre Δ pour quatorze combinaisons différentes

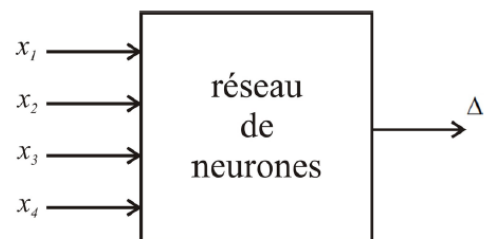


Figure 15 - Entrées-sortie du réseau de neurones.

Remarque : A la fin de ce sujet, nous appliquerons le réseau de neurones dans un contexte différent.

Traitement par IA

On souhaite dans cette partie contrôler le paramètre Δ à l'aide d'un réseau de neurones. L'objectif est qu'il fournisse la consigne Δ_{opt} en fonction des différentes combinaisons (x_1, x_2, x_3, x_4) . Le contrôleur associé au correcteur de facteur de puissance doit permettre en temps réel :

- Soit de désactiver le correcteur de facteur de puissance en connectant l'installation électrique du studio directement au réseau ($\Delta = -1$)
- Soit de fixer la valeur de Δ lorsque le correcteur de puissance est activé

Question 1: Proposer un mode de communication pour que les différents appareils électriques de l'installation puissent transmettre leurs modes de fonctionnement au contrôleur. Aucun détail technique n'est attendu pour cette question

Question 2: Préciser les avantages et inconvénients du choix d'un réseau de neurones pour modéliser la commande

Question 3: Préciser, en justifiant la réponse, en quoi le choix d'une méthode d'apprentissage supervisé est pertinent dans ce problème

Le perceptron

Le réseau de neurones choisi dans cette étude est un perceptron multicouche à une première couche cachée (couche d'entrée) à 4 neurones ($i \in (1,2,3,4)$), une seconde couche cachée à 4 neurones ($i' \in (1',2',3',4')$) et une couche de sortie à un neurone ($i'' \in (1'')$). Le système prend en entrée un vecteur à 4 composantes (x_1, x_2, x_3, x_4) et la donnée de sortie est Δ . Chaque neurone prend en entrée toutes les sorties de la couche précédente et renvoie une seule sortie. La figure 16 présente cette structure et la figure 17 donne les détails d'un neurone (i).

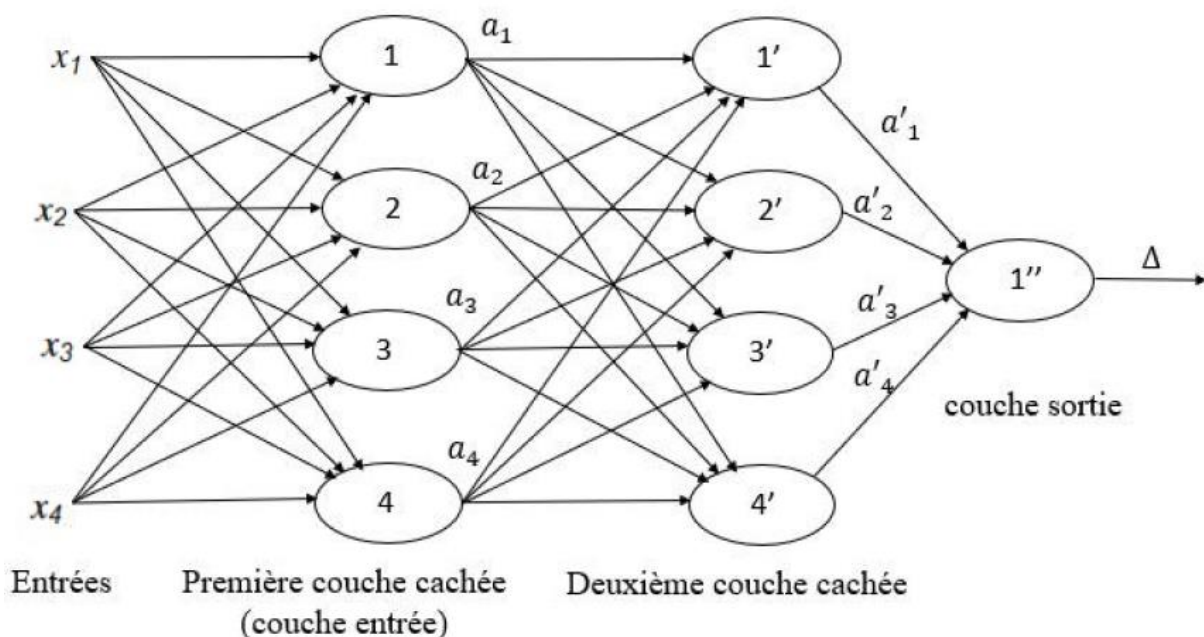


Figure 16 – Structure du réseau de neurones

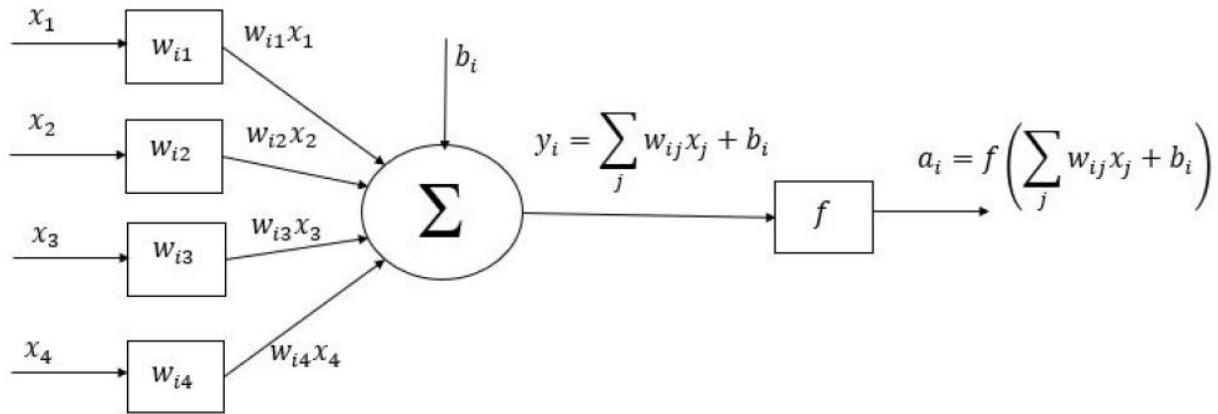


Figure 17 – Structure du neurone (i)

Nous noterons :

- $w_{ij} \in \mathbb{R}$: Le poids entre l'entrée j et le neurone (i) (w_{11} poids de l'entrée x_1 du neurone (1))
- $b_i \in \mathbb{R}$: Le biais du neurone (i)
- $f \in \mathbb{R}$: La fonction d'activation

Le processus d'apprentissage pour former le réseau de neurones consiste à appliquer une méthode itérative pour modifier les valeurs w_{ij} et b_i . On choisit les valeurs initiales aléatoirement.

La fonction d'activation choisie est une sigmoïde définie par $f: \mathbb{R} \rightarrow \mathbb{R}, x \rightarrow f(x) = \frac{1}{1+e^{-x}}$. On utilisera la fonction exp à utiliser dans le module numpy en Python dans la suite. Cette fonction s'applique sur un tableau (array).

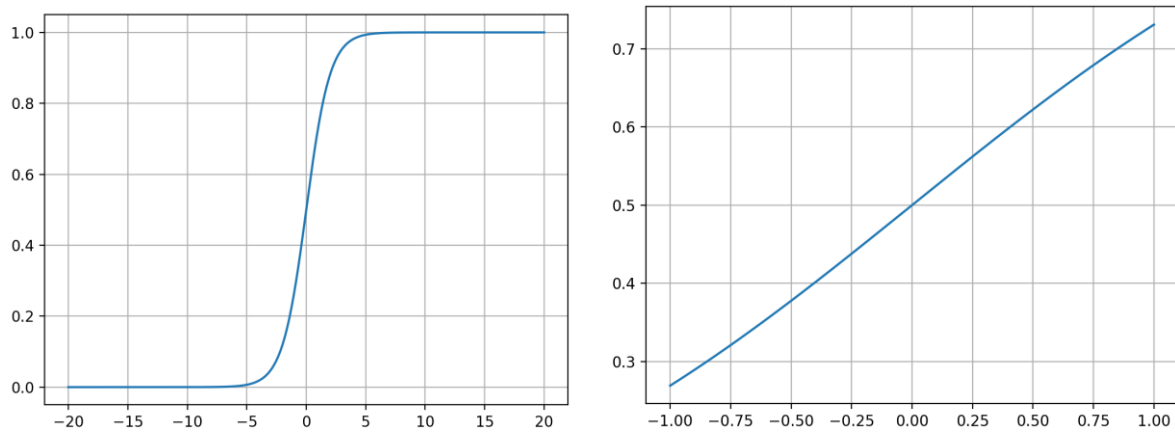


Figure 18 - Tracé de la fonction f

Vous avez à disposition un code élève si vous souhaitez réaliser la programmation de ce sujet.

Question 4: Après avoir calculé la dérivée de f , proposer les fonctions f et f_prime en langage Python

Phase d'inférence

Notations :

- Les grandeurs scalaires seront notées en minuscules, par exemple w
- Les grandeurs matricielles seront notées en majuscules, par exemple W
- L'indice k fait référence à la couche, par exemple W_2 pour la seconde couche

Cette phase consiste à calculer la sortie Δ à partir d'un vecteur d'entrée X connu. La sortie du dernier neurone peut s'écrire ainsi :

$$\Delta = f(W_3 f(W_2 f(W_1 X + B_1) + B_2) + B_3)$$

Question 5: Donner les dimensions des arrays W_2 , W_3 , B_2 et B_3

Pour la couche d'entrée, on propose l'écriture matricielle suivante : $Y_1 = W_1 X + B_1$; $A_1 = f(Y_1)$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = W_1 \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Question 6: Donner l'expression de la matrice W_1

Question 7: Proposer la fonction Python `inference_couche(X,W,B)` renvoyant le vecteur des sorties de la couche A

On prend :

$$A_1 = \begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} ; \quad Y_1 = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} ; \quad X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} ; \quad B_1 = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$W_1 = \begin{bmatrix} w_{11} & w_{12} & w_{13} & w_{14} \\ w_{21} & w_{22} & w_{23} & w_{24} \\ w_{31} & w_{32} & w_{33} & w_{34} \\ w_{41} & w_{42} & w_{43} & w_{44} \end{bmatrix} = \begin{bmatrix} 0,3 & 0,2 & 0,25 & -0,04 \\ 0,4 & -0,3 & 0,6 & -0,3 \\ -0,4 & 0,3 & -0,6 & -0,4 \\ -1 & -0,75 & -0,1 & -0,5 \end{bmatrix}$$

Notons que, pour cette première phase d'apprentissage qu'on appelle souvent l'initialisation, les valeurs des poids ont été générées aléatoirement et les biais choisis nuls.

Question 8: En vous aidant de la figure 18, calculer $A_1 = f(Y_1)$ dans ce cas

En reproduisant le même calcul pour les couches suivantes, nous calculerons Δ .

On suppose le code suivant existant :

```
X = [[0],  
      [1],  
      [0],  
      [0]]  
X = np.array(X)  
  
W1 = [[ 0.3, 0.2, 0.25, -0.04],  
      [ 0.4, -0.3, 0.6, -0.3],  
      [-0.4, 0.3, -0.6, -0.4],  
      [-1, -0.75, -0.1, -0.5]]  
W1 = np.array(W1)  
  
B1 = [[0.0],  
      [0],  
      [0],  
      [0]]  
B1 = np.array(B1)  
  
W2 = W1.copy()  
B2 = B1.copy()  
W3 = W1[0,:].copy().reshape(1,4)  
B3 = np.array([[0.]])
```

Nous avons pris soin de créer des arrays aux bonnes dimensions, composés de flottants, en réalisant des copies pour rendre chaque matrice indépendante des autres. Elles seront utilisées comme variables globales dans la suite.

Une annexe « Fonctions Python utiles » est à votre disposition en fin de sujet. Vous pourrez y piocher les fonctions souhaitées.

Question 9: Proposer la fonction Python `inference(X)` renvoyant le flottant Δ associé

On trouve $\Delta = 0,59 \text{ V}$. Ce résultat signifie que pour l'éclairage allumé et tous les autres appareils éteints, le réseau de neurones prévoit un réglage de $\Delta = 0,59 \text{ V}$. Cependant, le tableau 3 indique que pour cette combinaison, le réglage optimal est $\Delta_{opt} = 0,18 \text{ V}$.

Il faut donc maintenant modifier les valeurs des poids et biais pour que la sortie du réseau de neurones se rapproche de Δ_{opt} et ce, pour toutes les combinaisons d'entrée possibles.

Rétropropagation

NDLR : J'ai reformulé grandement cette partie par rapport au sujet de concours pour qu'elle soit plus facilement compréhensible et programmable entièrement.

Pour toute combinaison $X_l = (x_1, x_2, x_3, x_4)$ des entrées à la ligne l du tableau 3, on souhaite minimiser l'erreur entre la sortie calculée par inférence avec le réseau de neurones Δ_l et le réglage optimal attendu Δ_l^{opt} noté t_l dans la suite. Dans le tableau 3, nous disposons de 9 données d'entraînement, soit 9 paires $\{X_l, t_l\}$ pour chacune des $N = 9$ lignes concernées.

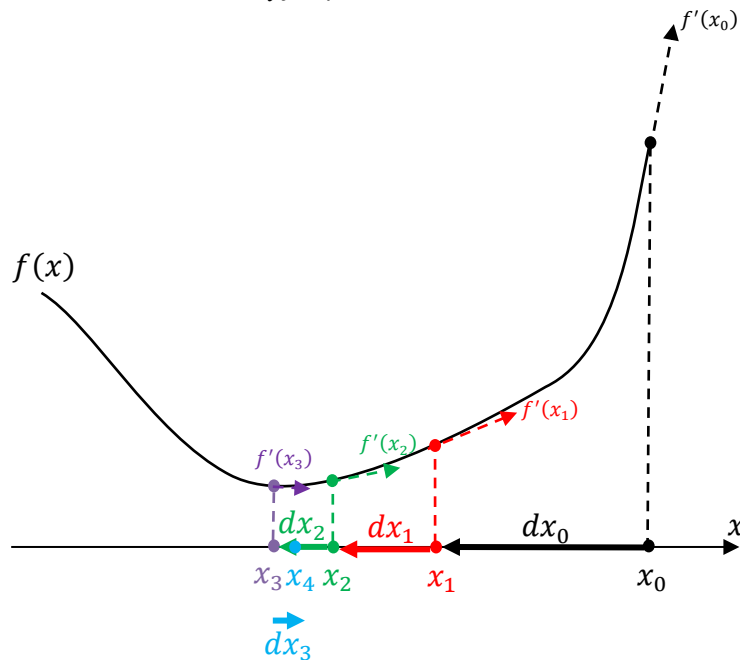
Le calcul de l'erreur (fonction de coût) $\mathcal{L}_l = (\Delta_l - t_l)^2$ sur chacune de ces lignes va être utilisé pour calculer de « meilleurs » valeurs des poids et biais, ce qui correspond à la phase dite « d'entraînement » du réseau de neurones. On parle de « rétropropagation » du gradient de la fonction de coût dans tout le réseau depuis la couche de sortie jusqu'à la couche d'entrée pour mettre à jour les W_i et B_i . Cette procédure, réalisée pour toutes les données d'entrée, s'appelle un « epoch ».

L'epoch sera alors répété sur un grand nombre d'itérations jusqu'à obtenir un critère acceptable d'erreur quadratique moyenne totale $\mathcal{L}_{tot}$:

$$\mathcal{L}_{tot} = \frac{1}{N} \sum_{l=1}^N (\Delta_l - t_l)^2 = \frac{1}{N} \sum_{l=1}^N \mathcal{L}_l$$

La mise à jour des valeurs des W_i et B_i est réalisée à l'aide de la méthode de la descente du gradient. Le principe est le suivant : Soit $\mathcal{L}_l$ le gradient de l'erreur en sortie. Supposons connaître la dérivée $\frac{\partial \mathcal{L}_l}{\partial w_{ij}}$.

En écrivant $w_{ij} = w_{ij} - \alpha \frac{\partial \mathcal{L}_l}{\partial w_{ij}}$ avec α un réel appelé « taux d'apprentissage », on peut mettre à jour w_{ij} de manière itérative en diminuant $\mathcal{L}_l$ jusqu'à atteindre un minimum.



Nous allons dans un premier temps chercher les expressions « locales » pour une couche de $\frac{\partial \mathcal{L}_l}{\partial a_i}, \frac{\partial \mathcal{L}_l}{\partial b_i}, \frac{\partial \mathcal{L}_l}{\partial w_{ij}}$ et $\frac{\partial \mathcal{L}_l}{\partial x_j}$ avant de passer de manière matricielle « globale ».

On rappelle que :

$$a_i = f(y_i) \quad ; \quad y_i = \sum_{j=1}^4 (w_{ij}x_j) + b_i$$

Pour cette question, on ne s'intéresse qu'à la dernière couche.

Question 10: Donner l'expression de $\frac{\partial \mathcal{L}_l}{\partial a_i}$ en fonction de a_i et t_l

Pour une ligne des entrées, on peut donc calculer l'erreur en sortie de la dernière couche puis son gradient. On pourra alors calculer le gradient en entrée de cette couche à l'aide de ses paramètres ; celui-ci sera utilisé comme gradient en sortie de la couche précédente, et ainsi de suite.

Pour les questions suivantes, on s'intéresse à n'importe quelle couche. On simplifie l'écriture $\mathcal{L}_l$ pour $\mathcal{L}$, erreur en sortie du réseau pour la ligne l des entrées.

Question 11: Montrer que $\frac{\partial \mathcal{L}}{\partial b_i} = \frac{\partial \mathcal{L}}{\partial a_i} f'(y_i)$

Question 12: Montrer que $\frac{\partial \mathcal{L}}{\partial w_{ij}} = \frac{\partial \mathcal{L}}{\partial a_i} f'(y_i) x_j$

Un x_k d'entrée de couche influence tous les neurones de celle-ci. Autrement dit, la fonction de coût $\mathcal{L}$ est une fonction de a_1, a_2, a_3 et a_4 , eux-mêmes tous fonctions de x_k . On admet l'expression de la règle de la chaîne suivante :

$$\frac{\partial \mathcal{L}(a_1(x_k), a_2(x_k), a_3(x_k), a_4(x_k))}{\partial x_k} = \frac{\partial \mathcal{L}}{\partial a_1} \frac{\partial a_1}{\partial x_k} + \frac{\partial \mathcal{L}}{\partial a_2} \frac{\partial a_2}{\partial x_k} + \frac{\partial \mathcal{L}}{\partial a_3} \frac{\partial a_3}{\partial x_k} + \frac{\partial \mathcal{L}}{\partial a_4} \frac{\partial a_4}{\partial x_k} = \sum_{i=1}^4 \frac{\partial \mathcal{L}}{\partial a_i} \frac{\partial a_i}{\partial x_k}$$

Question 13: Montrer que $\frac{\partial \mathcal{L}}{\partial x_k} = \sum_{i=1}^4 \frac{\partial \mathcal{L}}{\partial a_i} f'(y_i) w_{ik}$

On souhaite maintenant établir les relations matricielles « globales » permettant de réaliser la rétropropagation sur une couche quelconque. Pour une couche, on introduit les matrices suivantes (sans indices i de couche pour alléger les notations), où X est le vecteur d'entrée et A le vecteur de sortie :

$$\frac{\partial \mathcal{L}}{\partial W} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_{11}} & \frac{\partial \mathcal{L}}{\partial w_{12}} & \frac{\partial \mathcal{L}}{\partial w_{13}} & \frac{\partial \mathcal{L}}{\partial w_{14}} \\ \frac{\partial \mathcal{L}}{\partial w_{21}} & \frac{\partial \mathcal{L}}{\partial w_{22}} & \frac{\partial \mathcal{L}}{\partial w_{23}} & \frac{\partial \mathcal{L}}{\partial w_{24}} \\ \frac{\partial \mathcal{L}}{\partial w_{31}} & \frac{\partial \mathcal{L}}{\partial w_{32}} & \frac{\partial \mathcal{L}}{\partial w_{33}} & \frac{\partial \mathcal{L}}{\partial w_{34}} \\ \frac{\partial \mathcal{L}}{\partial w_{41}} & \frac{\partial \mathcal{L}}{\partial w_{42}} & \frac{\partial \mathcal{L}}{\partial w_{43}} & \frac{\partial \mathcal{L}}{\partial w_{44}} \end{bmatrix} ; \quad \frac{\partial \mathcal{L}}{\partial X} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial x_1} \\ \frac{\partial \mathcal{L}}{\partial x_2} \\ \frac{\partial \mathcal{L}}{\partial x_3} \\ \frac{\partial \mathcal{L}}{\partial x_4} \end{bmatrix} ; \quad \frac{\partial \mathcal{L}}{\partial A} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial a_1} \\ \frac{\partial \mathcal{L}}{\partial a_2} \\ \frac{\partial \mathcal{L}}{\partial a_3} \\ \frac{\partial \mathcal{L}}{\partial a_4} \end{bmatrix} ; \quad \frac{\partial \mathcal{L}}{\partial B} = \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial b_1} \\ \frac{\partial \mathcal{L}}{\partial b_2} \\ \frac{\partial \mathcal{L}}{\partial b_3} \\ \frac{\partial \mathcal{L}}{\partial b_4} \end{bmatrix}$$

Avec ces relations, connaissant le gradient de la fonction coût en sortie de couche $E_A = \frac{\partial \mathcal{L}}{\partial A}$, il sera possible de mettre à jour les matrices W et B de la couche par descente de gradient ainsi :

$$\begin{cases} W = W - \alpha \frac{\partial \mathcal{L}}{\partial W} \\ B = B - \alpha \frac{\partial \mathcal{L}}{\partial B} \end{cases}$$

Puis on pourra renvoyer le gradient de la fonction coût en entrée de couche $E_X = \frac{\partial \mathcal{L}}{\partial X}$ qui servira de gradient en sortie de la couche précédente : $E_A^{c-1} = E_X^c$

On introduit les opérateurs suivants :

- « * » pour le produit terme à terme entre deux arrays (utilisable en Python avec *)
- « . » pour le produit matriciel (np.dot() en Python)

Question 14: Etablir les relations matricielles donnant l'expression de E_X , $\frac{\partial \mathcal{L}}{\partial W}$ et $\frac{\partial \mathcal{L}}{\partial B}$ en fonction de W , E_A , X , Y et f'

Les formules étant établies, on propose de réaliser le code Python permettant d'entraîner le réseau de neurones créé précédemment.

Nous allons proposer une fonction de rétropropagation qui, pour une couche, devra :

- Prendre en arguments X (vecteur d'entrée), W (matrice des poids), B (vecteur de biais), E_A (gradient du coût en sortie) et α (valeur du coefficient d'apprentissage)
- Calculer $E_W = \frac{\partial \mathcal{L}}{\partial W}$, $E_B = \frac{\partial \mathcal{L}}{\partial B}$ et $E_X = \frac{\partial \mathcal{L}}{\partial X}$
- Réaliser les mises à jour de W et B en place
- Renvoyer le vecteur gradient du coût en entrée $E_X = \frac{\partial \mathcal{L}}{\partial X}$

Question 15: Ecrire la fonction Python `retropropagation_couche(X,W,B,Ea,alpha)` attendue

Nous allons maintenant réaliser la mise à jour complète du réseau de neurones ($W_3, B_3, W_2, B_2, W_1, B_1$) pour une donnée $\{X_l, t_l\}$ du tableau 3 en réalisant la rétropropagation du gradient de la fonction de coût en sortie de 3^e couche à travers les 3 couches du réseau.

Question 16: Ecrire la fonction `epoch_ligne(X,Deltaopt,alpha)` réalisant la procédure epoch sur une ligne des entrées X connaissant la valeur Deltaopt de la ligne associée

On propose le code suivant :

```
Donnees = [
    [0, 0, 0, 1, -1],
    [0, 0, 1, 1, -1],
    [0, 0, 2, 0, -1],
    [0, 0, 0, 2, -1],
    [0, 0, 1, 2, -1],
    [0, 1, 0, 0, 0.18],
    [2, 2, 0, 0, 0.32],
    [2, 2, 2, 0, 0.49],
    [2, 2, 2, 2, -1],
    [0, 0, 1, 0, -1],
    [0, 0, 2, 1, -1],
    [0, 0, 2, 2, -1],
    [1, 0, 0, 0, 0.22],
    [2, 2, 1, 0, 0.42]
]
Donnees = np.array(Donnees)
Donnees_A = Donnees[:9]
Donnees_T = Donnees[9:]
```

L'argument LD des prochaines fonctions sera la liste `Donnees_A`.

Il faut maintenant réaliser un epoch complet, c'est-à-dire mettre à jour le réseau par la procédure précédente en utilisant toutes les données d'apprentissage.

Question 17: Ecrire la fonction `epoch(LD,alpha)` réalisant un epoch sur toutes les données d'apprentissage

Remarque : on pensera à transformer les données D en array vertical à deux dimensions avec reshape

Après réalisation d'un epoch, on souhaite quantifier l'erreur quadratique moyenne totale du réseau.

$$\mathcal{L}_{tot} = \frac{1}{N} \sum_{l=1}^N \mathcal{L}_l$$

Question 18: Ecrire la fonction `erreur_totale(LD)` calculant et renvoyant l'erreur quadratique moyenne totale après un epoch

Question 19: Créer la fonction `entrainement(LD,Nit,alpha)` réalisant Nit epoch et renvoyant les listes Lit du numéro de l'itération et Lerr de l'erreur totale associée

Question 20: Ecrire les lignes de code réalisant 100 epoch pour différentes valeurs de alpha et tracant l'erreur totale en fonction du nombre d'itérations réalisées

N'ayez crainte, les résultats obtenus ne sont pas ceux du sujet, on y vient.

Analyse des résultats

L'évolution de la fonction de coût total $\mathcal{L}_{tot}$ pendant la phase d'entraînement est donnée sur la figure 19. Elle comporte 100 itérations.

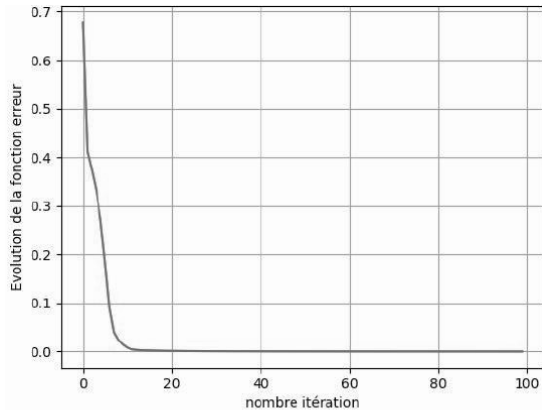


Figure 19 - Entraînement modèle

	box-TV x_1	éclairage x_2	radiateur x_3	chauffe-eau x_4	Δ_{opt} (V)	Δ (V)
Données d'entraînement	0	0	0	1	-1	-0,981
	0	0	1	1	-1	-0,979
	0	0	2	0	-1	-0,961
	0	0	0	2	-1	-0,983
	0	0	1	2	-1	-0,980
Données test	0	1	0	0	0,18	0,178
	2	2	0	0	0,32	0,341
	2	2	2	0	0,49	0,474
	2	2	2	2	-1	-0,970
	0	0	1	0	-1	-0,919
	0	0	2	1	-1	-0,977
	0	0	2	2	-1	-0,978
	1	0	0	0	0,22	0,015
	2	2	1	0	0,42	0,412

Tableau 5 - Valeurs optimales du paramètre Δ pour quatorze combinaisons différentes

NDLR : Cette courbe de la figure 19 n'a pas été obtenue sur le set des 9 données du tableau 5. Il est donc normal que vous ne trouviez pas la même chose que dans ce sujet.

Le graphique montre que la phase d'entraînement, constituée d'une succession d'inférences et de rétropropagation, a permis de diminuer la fonction coût, c'est-à-dire l'erreur quadratique moyenne entre le résultat Δ du modèle et les valeurs cibles des données d'entraînement. Il y a désormais peu d'écart pour les données d'entraînement.

Sur le tableau 5, on peut alors comparer les valeurs de Δ cibles aux valeurs calculées sur les 5 données de test.

Question 21: Analyser les résultats. Préciser si ce réseau de neurones est capable de bien prédire la valeur optimale de Δ

Question 22: Pour améliorer le modèle, préciser en justifiant la réponse, s'il est préférable d'augmenter le nombre d'itérations de l'entraînement ou d'augmenter le nombre de données test

Nouveau contexte – Redondance de capteurs

L'article « A Review of Exhaust Gas Temperature Sensing Techniques for Modern Turbine Engine Controls » de juillet 2014 s'intéresse à la manière dont on mesure la température des gaz d'échappement dans les turbines à gaz.

Having several thermocouples serves two purposes: sensor redundancy - if one sensor fails there are other sensors still there to measure EGT, and averaging - the more point measurements there are, the closer the average sensor output will be to the actual gas path average temperature. Multiple thermocouples at a single axial location are typically bundled together in a harness.

Extrait page 6

Traduction : « Avoir plusieurs thermocouples sert à deux choses : assurer une redondance des capteurs — si un capteur tombe en panne, les autres continuent à mesurer l'EGT (température des gaz d'échappement) — et effectuer une moyenne. Plus il y a de points de mesure, plus la valeur moyenne fournie par les capteurs sera proche de la température moyenne réelle des gaz dans l'écoulement. Plusieurs thermocouples situés à une même position axiale sont généralement regroupés dans un faisceau. »

Plusieurs capteurs de température identiques (thermocouples EGT) sont utilisés afin d'assurer la redondance, permettant à la fois d'identifier un capteur défaillant et de déterminer une température par moyenne à partir de plusieurs mesures.

Considérons que l'on dispose de 4 capteurs de température. Chaque température mesurée T_i est normalisée en une valeur $x_i \in [0,1]$ par la transformation suivante (T_{min} et T_{max} étant les températures mesurables minimale et maximale) :

$$x_i = \frac{T_i - T_{min}}{T_{max} - T_{min}}$$

Programmons un réseau de neurones pour déterminer la moyenne des capteurs. Même si le réseau de neurones est une « usine à gaz » vis-à-vis d'un simple calcul de moyenne, cette application reste très intéressante : elle vous permet de bien comprendre comment construire un dataset, d'implémenter une fonction dont on connaît exactement le résultat, qui est indépendante de l'ordre des entrées, et enfin de visualiser les performances du réseau sur un cas parfaitement maîtrisé.

Soient les données normalisées suivantes présentes dans le code élèves :

$$\left(x_1, x_2, x_3, x_4, moy x_i\right) \quad ; \quad x_i \in [0,1[$$

```
from random import random as rd

def cree_dataset(N):
    dataset = []
    for i in range(N):
        x1,x2,x3,x4 = rd(),rd(),rd(),rd()
        d = (x1+x2+x3+x4)/4
        data = [x1,x2,x3,x4,d]
        dataset.append(data)
    return np.array(dataset)

N = 1000
Ind = int((75/100)*N)
Donnees = cree_dataset(N)
Donnees_A = Donnees[:Ind]
Donnees_T = Donnees[Ind:]
```

Question 23: Entraîner et vérifier le réseau créé précédemment sur 1000 lignes de ces données et vérifier qu'il converge en moins de 1000 itérations (alpha=0.01)

Suggestions :

- Initialiser les W_i et B_i aléatoirement en utilisant `np.random.rand(l,c)`
- Tracer la moyenne obtenue avec le réseau de neurones en fonction de la moyenne souhaitée $moy x_i$

Utilisation de Scikit-learn

Pour terminer, voyons comment utiliser le module scikit-learn pour construire un réseau de neurones performant automatiquement.

Si besoin, pour installer le module, tenter dans la console « `pip install scikit-learn` » ou « `python -m pip install scikit-learn` ».

Nous allons utiliser un réseau de neurones de type « Regressor » permettant de renvoyer une valeur continue, contrairement à un « Classifieur » qui renvoie un nombre fini de catégories.

Quelques rappels :

Import module et réseau de neurones multicouches	<code>from sklearn.neural_network import MLPRegressor</code>
Création du modèle	<code>rn = MLPRegressor(solver='sgd')</code>
Entraînement du modèle	<code>rn.fit(trn_x, trn_y)</code>
Prédiction Lx : Liste de 4 valeurs	<code>rn.predict([Lx])[0]</code>
Sauvegarde de modèle	<code>from joblib import dump</code> <code>dump(rn, 'rn.joblib')</code>
Chargement de modèle	<code>from joblib import load</code> <code>rn = load('rn.joblib')</code>
Score du modèle en %	<code>Score = rn.score(tst_x, tst_y)</code>

Avec :

- `_x` : Entrées : liste de listes de 4 valeurs x_i
- `_y` : Sortie : liste des moyennes des 4 entrées
- `trn_` : Données d'entraînement pour l'apprentissage
- `tst_` : Données de test pour le score et la vérification des performances
- `'sgd'` = Stochastic Gradient Descent

Question 24: Mettre en place le code permettant de créer le réseau de neurones, l'entraîner, vérifier son score et afficher ses performances (moyenne obtenue en fonction de moyenne souhaitée), avec l'objectif d'un score de minimum 99%

Vous pourrez explorer le réseau ainsi créé afin d'obtenir sa structure à l'aide de ces instructions :

```
rn.n_features_in_      # Couche d'entrée : Nombre d'entrées
rn.hidden_layer_sizes  # Couches cachées : Nb neurones
rn.n_outputs_          # Couche de sortie : Nb neurones
```

Pour information, on utilise « `hidden_layer_sizes` » pour modifier cette structure en précisant le nombre de neurones par couche cachée ($n_1 n_2, \dots$), et « `activation` » pour modifier la fonction d'activation. Consulter l'aide du module scikit-learn vous aidera à aller plus loin.

```
MLPRegressor(solver='sgd', hidden_layer_sizes=(4), activation='logistic')
```


ANNEXE 2 - FONCTIONS PYTHON UTILES

La bibliothèque numpy est supposée installée de la manière classique - `import numpy as np`.

np.dot (A, B) : retourne le produit de deux matrices $[A]$ et $[B]$.

A.T : retourne la transposée d'une matrice $[A]$.

len(X) : retourne le nombre d'éléments de l'objet X. Ce dernier peut être une liste ou un tableau numpy.

np.abs(X) : retourne un tableau dont les éléments correspondent aux modules des éléments du tableau X.

np.arange(a, b, p) : retourne un tableau 1D dont les éléments sont répartis avec un pas p entre a et b, a est le premier élément du tableau, le dernier élément est strictement inférieur à b. Si l'argument p n'est pas renseigné, il est pris par défaut égal à 1.

np.fft.rfft(X) : retourne, sous forme d'un tableau 1D de nombres complexes, la transformée de Fourier discrète du signal échantillonné correspondant à X, un tableau 1D dont les éléments doivent être réels. Les éléments du tableau retourné correspondent aux fréquences $n \frac{F_e}{N}$ avec $n \in \llbracket 0, \lfloor \frac{N}{2} \rfloor \rrbracket$, N le nombre d'éléments de X et F_e la fréquence d'échantillonnage. Enfin, le facteur multiplicatif $2/N$ doit être appliqué au tableau 1D retourné pour obtenir le spectre du signal continu correspondant au signal échantillonné contenu dans X.

np.linspace(a, b, N) : retourne un tableau 1D de N éléments répartis linéairement entre a et b, a est le premier élément du tableau, b le dernier.

plt.plot(X, Y) : permet de tracer le tableau 1D Y en fonction du tableau 1D X. Les deux tableaux doivent avoir le même nombre d'éléments. Des arguments optionnels permettent de choisir la couleur du tracé, l'épaisseur du trait, etc.

plt.xlim(x_min, x_max) : permet de limiter un tracé aux abscisses comprises entre x_min et x_max.

plt.xlabel('titre abscisses') : permet d'attribuer un titre à l'axe des abscisses. Des arguments optionnels permettent de choisir la taille de la police, la couleur du texte, etc.

plt.ylim(y_min, y_max) : permet de limiter un tracé aux ordonnées comprises entre y_min et y_max.

plt.ylabel('titre ordonnées') : permet d'attribuer un titre à l'axe des ordonnées. Des arguments optionnels permettent de choisir la taille de la police, la couleur du texte, etc.

Ajouts personnels :

- L'opérateur multiplication « $A*B$ » réalise le produit terme à terme de deux arrays de mêmes dimensions
- Soit A un array à une dimension. `A.reshape(-1,1)` renvoie (sans modifier A) un array à deux dimensions à une colonne et de même nombre de lignes que d'éléments de A
- `np.random.rand(l,c)` renvoie un array de l lignes et c colonnes rempli de flottants aléatoires dans $[0, 1[$