Les Réseaux Neurones Multi-Couches (MLP, 1D-CNN, 2D-CNN, RNN) – Formulations Mathématiques, explications et applications

Auteur : Abdel YEZZA, Ph.D

Date : Avr. 2026

Réf. utilisée du même auteur :
1. Référence Mathématique des Modèles et Métriques ML
2. Machine Learning Cross-Validation (CV)

Repository GITHUB : https://github.com/ayezza/mlp_basique

NOTE : des erreurs involontaires peuvent glisser dans ce document, il appartient au lecteur de bien vérifier l'exactitude du contenu avant de procéder à toute expérimentation ou utilisation dans des projets d'entreprises – toute utilisation dans un cadre professionnel uniquement doit référencer l'auteur.

Table des matières

A.1 Le réseau MLP (Multi-Layers Perceptron)

A1.1 Qu'est-ce qu'un MLP ?

Un perceptron multicouche est la forme la plus simple d'un réseau de neurones. Il est formé :

1. d'une **couche d'entrée** (Input) des variables descriptives (features),
2. **d'une ou plusieurs couches intermédiaires non-visibles** effectuant des transformations multiples, avec à chaque couche une **transformation** basée sur la couche précédente suivie d'une **activation** pour le transfert de l'information vers la couche suivante
3. et **d'une couche de sortie** fournissant la fonction de coût à minimiser (fonction de perte) ainsi que les prédictions résultantes (valeurs réelles régressives ou probabilités (donc classes) prédites).

A1.2 Notations standards générales utilisées

Symbôle	Signification
L	Total des couches d'apprentissage utilisées par le modèle
l	L'index des couches allant de 0 à L-1 ($l = 0, 1, \dots, L-1$)
n_l	Total des neurones utilisés dans la couche l
$\mathbf{x} \in \mathbb{R}^{(n_0)}$	Cette variable représente le vecteur en entrée (Input Vector) du point de vue mathématique, ce qui l'équivalent des variables descriptives du point de vue jeu de données. La constante $n_0$représente la dimension de ce vecteur augmenté de 1 représentant *le coefficient du biais (b)*.
$\mathbf{a}^{((l))} \in \mathbb{R}^{(n_l)}$	Représente le vecteur d'activation à la sortie de la couche l, avec $\mathbf{a}^{((0))} = \mathbf{x}$ **(Fonction identité)**
$\mathbf{z}^{((l))} \in \mathbb{R}^{(n_l)}$	Représente la variable vecteur d'activation à la sortie de la couche (cachée) l (logit) utilisant l'une des fonctions d'activation indiquées ci-dessous
$W^{((l))} \in \mathbb{R}^{(n_l \times n_{(l-1)})}$	Représente la matrice des poids utilisés (initialement)/calculés (par optimisation de la fonction coût à la sortie à la fin de la passe retour utilisant la technique de gradient descendant (DG) ou celle du gradient descendant stochastique (SDG)) au niveau de la couche l
$\mathbf{b}^{((l))} \in \mathbb{R}^{(n_l)}$	Vecteur représentant les biais utilisés au niveau de la couche l
$\boldsymbol{\delta}^{((l))}$	Représente l'erreur ou le gradient au niveau de la couche l utilisant la notion du **delta de Kronecker**
η	Coefficient d'apprentissage (entre 0 et 1) servant à pénaliser la fonction de coût et ainsi ajuster le modèle pour éviter notamment le sur-apprentissage (Overfitting) ou le sous-apprentissage (Underfitting)
$\mathcal{L}$	Représente la fonction de perte, autrement dit mathématiquement parlant, la fonction coût à minimiser. A noter que toute maximisation peut être transformée en une minimisation ($\max J \Leftrightarrow \min -J$)
$\hat{\mathbf{y}}$	Représente le vecteur des probabilités en sortie du réseau (output) pour les modèles de classification ou le vecteur des valeurs prédites pour les modèles de régression minimisant la fonction de perte ($argmin(\mathcal{L})$).
$\mathbf{y}$	One-hot encoded true label pour les variables ayant des valeurs discrètes
$\odot$	Signe de l'opérateur du produit matriciel (élément-par-élément) de Hadamard à ne pas confondre avec le produit standard de 2 matrices
$\star$	Signe utilisé pour une convolution totale

A1.3 Fonctions d'activation les plus communes

I. ReLU

ReLU (Rectified Linear Unit) :

$$a(z) = max(0, z) = 1_{(z>0)} = \begin{cases} z & Si \ \ z > 0 \\ 0 & Si \ \ z \leq 0 \end{cases}$$

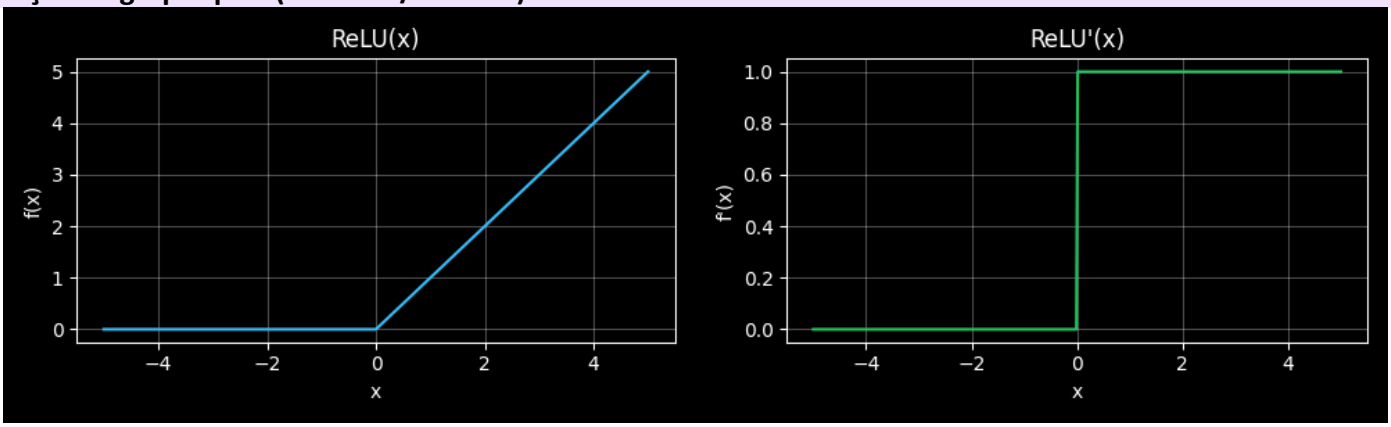
Dérivée :

$$a'(z) = \begin{cases} 1 & Si \ \ z > 0 \\ 0 & Si \ \ z < 0 \\ indéfinie & Si \ \ z = 0 \end{cases}$$

Propriétés principales :

1. **Usage le plus fréquent :** MLP, CNN aussi bien pour la classification que pour la régression
2. Rapide à calculer comme la ReLU (coût computational négligeable)
3. Pas de centre graphique (ne convient pas aux modèles probabilistes)
4. Utilisée comme fonction d'activation pour les couches intermédiaires (cachées)
5. Peut induire la disparition des neurones mourants, qui tombent à 0 et impliquent ainsi une perte de l'information (à cause du 0)

Aperçu des graphiques (Fonction/Dérivée) :



NOTE

Pour z = 0 on peut utiliser le sous-différentiel de Clarke (Gradient généralisé) : $\partial^C ReLU(0) = [0,1]$

II. Fonctions alternatives à ReLU

Afin d'éviter la disparition des neurones, on peut utiliser un certain nombre d'alternatives :

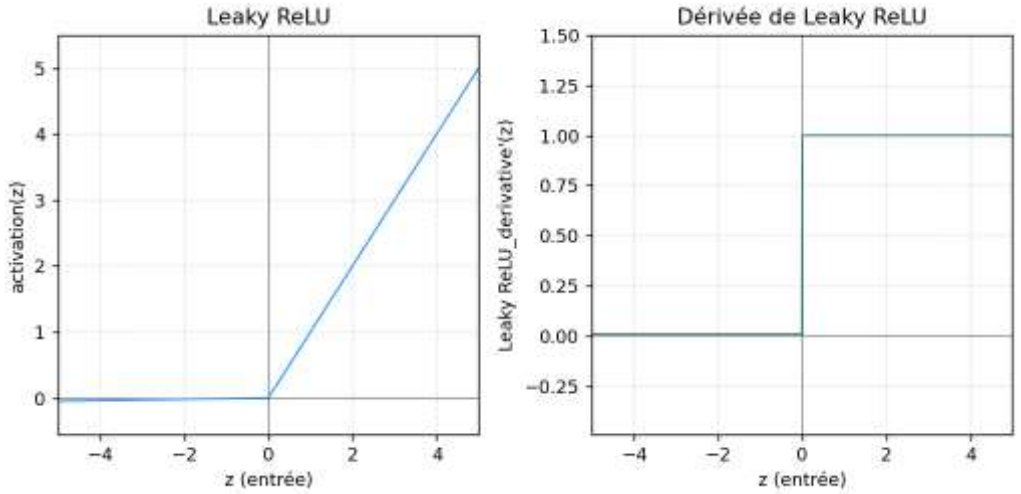
a. La fonction appelée Leaky ReLU :

$$LeakyReLU(z) = max(\alpha z, z) = \begin{cases} z & Si\ z > 0 \\ \alpha z & Si\ z \leq 0 \end{cases}$$

Avec généralement : $\alpha = 0.01\ ou\ 0.02$, voire appris dynamiquement pendant l'entrainement et dont la dérivée est :

$$LeakyReLU'(z) = \begin{cases} 1 & Si\ z > 0 \\ \alpha & Si\ z \leq 0 \end{cases}$$

Assurant ainsi toujours une valeur finie.

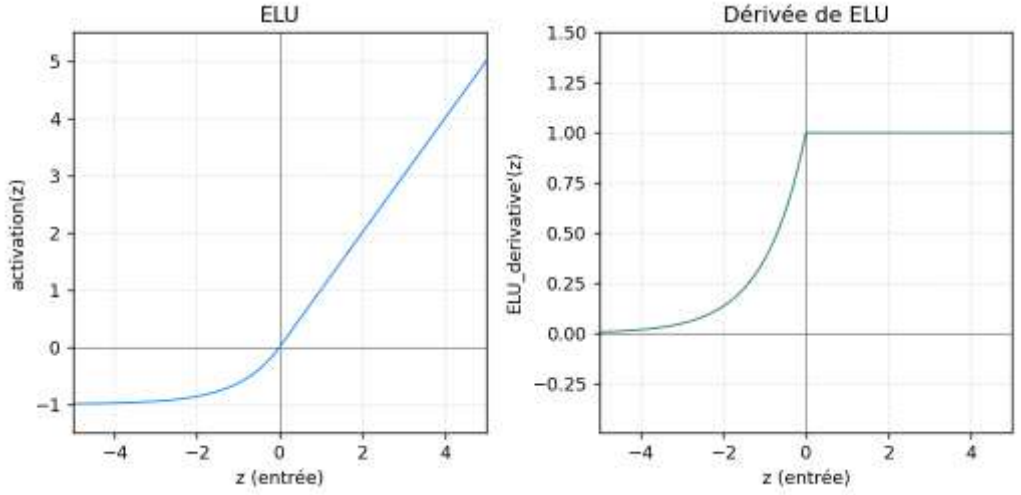


b. La fonction appelée ELU (E=Exponentiel):

$$ELU(z) = max(\alpha z, z) = \begin{cases} z & Si\ z > 0 \\ \alpha(e^z - 1) & Si\ z \leq 0 \end{cases}$$

Avec généralement : $\alpha = 1\ par\ défaut$, ou moins, voire appris dynamiquement pendant l'entrainement et dont la dérivée est :

$$ELU'(z) = \begin{cases} 1 & Si\ z > 0 \\ \alpha e^z & Si\ z \leq 0 \end{cases}$$

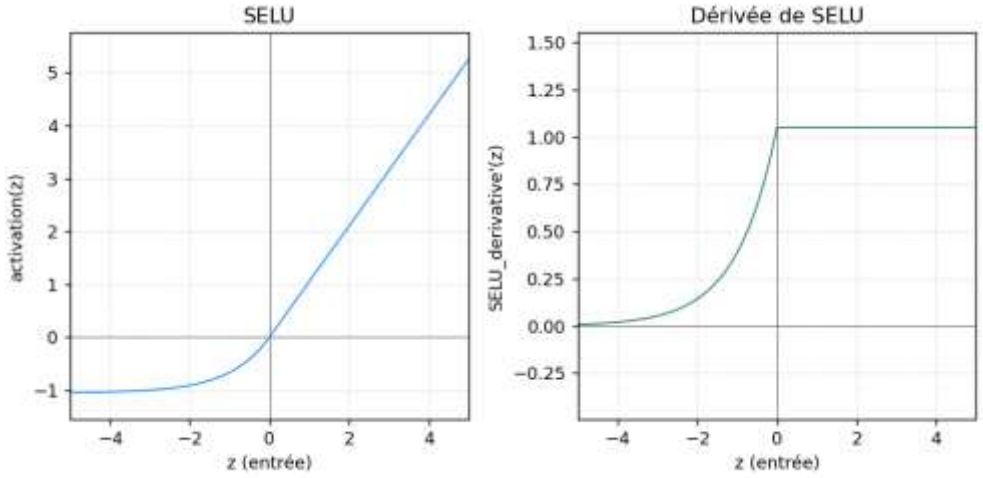


c. La fonction appelée SELU (S=Scaled):

$$\text{SELU}(z) = \begin{cases} \lambda z & \text{si } z > 0 \\ \lambda\alpha(e^z - 1) & \text{si } z \leq 0 \end{cases}$$

Avec généralement : $\lambda \approx 1.0507, \alpha \approx 1.6733$, voire appris dynamiquement pendant l'entrainement et dont la dérivée est :

$$\text{SELU}'(z) = \begin{cases} \lambda & \text{si } z > 0 \\ \lambda\alpha e^z & \text{si } z \leq 0 \end{cases}$$

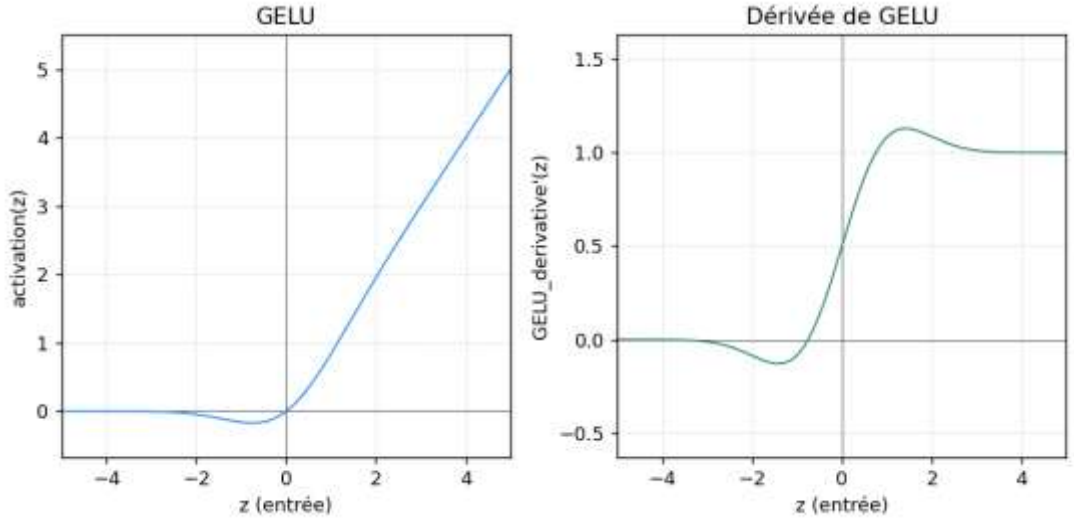


d. **La fonction appelée GELU (GE=Gaussien Error):**

$$GELU(z) = z\,\Phi(z)$$

où $\Phi(z) = P(X \leq z)$ est la CDF de la loi normale standard $X \sim \mathcal{N}(0,1)$, ce qui donne approximativement:

$$\text{GELU}(z) \approx 0.5\,z\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}(z + 0.044715\,z^3)\right)\right)$$



III. Sigmoïde

Sigmoïde (Fonction Logistique) : désignée par σ

$$a(z) = \sigma(z) = \frac{1}{1 + e^{-z}}$$

Dérivée :

$$a'(z) = a(z).\big(1 - a(z)\big)$$

Usages appropriés :

1. La couche de sortie pour la classification binaire (True/False, 1/0)
2. Les entrées dans **LSTM**

La fonction **SoftPlus** peut être utilisée comme alternative à **Sigmoïde** garantissant une sortie positive pas nécessairement comprise dans [0, 1] :

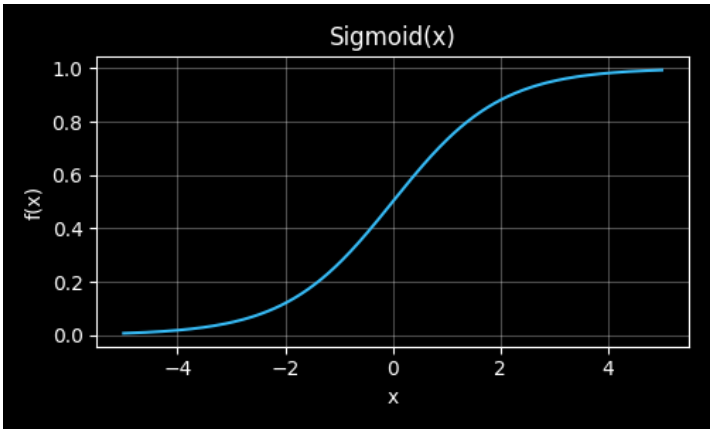
$$SoftPlus(z) = log(1 + e^z), \qquad SoftPlus'(z) = \sigma(z)$$

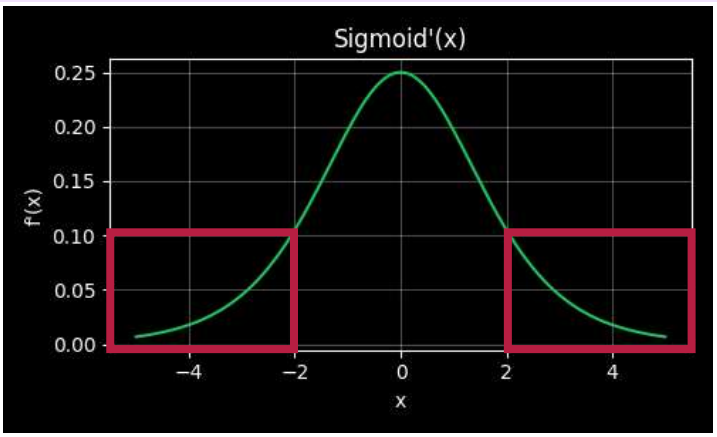
Dérivée : Sa dérivée est justement la fonction **Sigmoïde**.

Propriétés principales :

1. Intervalle des valeurs :]0, 1[
2. Les valeurs prises sont interprétées en tant que probabilités
3. Fonction lisse et différentiable partout avec une dérivée capée à 0,25
4. $a(-z) = 1 - a(z) \left(\underset{(Eq.)}{\Longleftrightarrow}\right) a(-z) + a(z) = 1$
5. $lim_{z \to \infty} a(z) = 1, \quad lim_{z \to -\infty} a(z) = 0$
6. $a(0) = 0.5$ (unique point de réflexion)
7. Pour des valeurs de |z| trop importantes le gradient peut disparaître (aspect appelé vanishing gradient - voir zones rouges ci-dessous) !
8. Sorties non centrées en 0 (problème pour la convergence)
9. Le calcul de l'exponentiel peut exploser et devenir coûteux dans certaines situations

Aperçu des graphiques (Fonction/Dérivée) :





IV. Tanh

Tanh (Tangente Hyperbolique) :

$$a(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{e^{2z} - 1}{e^{2z} + 1} = 2a(2z) - 1$$

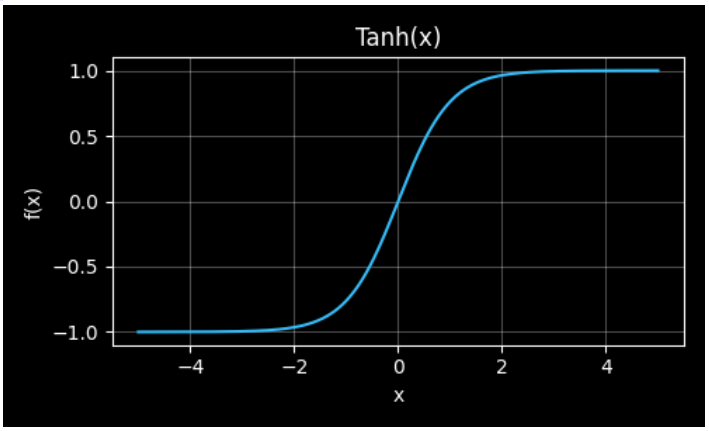
Dérivée :

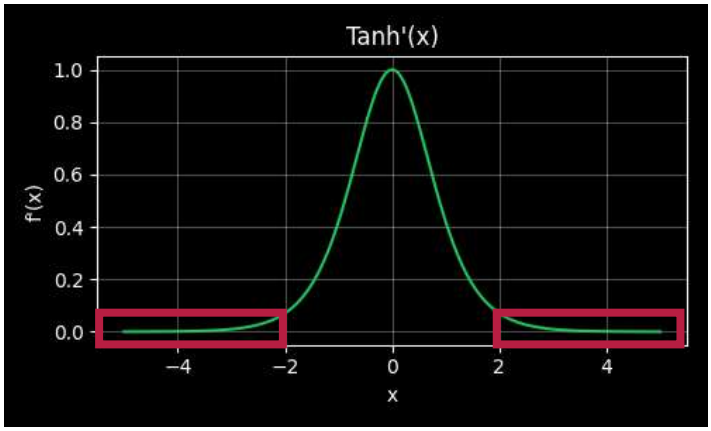
$$a'(z) = 1 - a^2(z)$$

Propriétés principales :

1. Intervalle des valeurs : $]-1,\ 1[$
2. Sortie plus centrée en 0 que la fonction **Sigmoïde** (voir la suite)
3. Son gradient est aussi plus fort que la **Sigmoïde** (cela se voit dans sa dérivée vs celle de la Sigmoïde) :
   $(1 - x^2)\ vs\ x(1 - x)$ → voir les courbes correspondantes
4. Pour des valeurs de |z| trop importantes le gradient peut disparaître des radars comme c'est le cas de la Sigmoïde
5. Assez coûteux comme la **Sigmoïde** à cause de la fonction exponentiel

Aperçu des graphiques (Fonction/Dérivée) :





V. Fonction Identité

Linéaire : utilisé pour la régression

$$a(z) = z$$

Dérivée :

$$a'(z) = 1$$

VI. Softmax

Softmax :

Contrairement aux fonctions d'activation précédentes à variable réelle, la Softmax quant à elles, est une fonction à variable vectorielle.

Pour chaque vecteur de dimension K, $z = (z_1, z_2, \dots, z_K)$:

$$a(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} = \frac{e^{z_i - \max_j(z_j)}}{\sum_{j=1}^{K} e^{z_j - \max_j(z_j)}}$$

C'est cette dernière formule d'équilibrage qui est généralement utilisée afin de réduire l'amplitude de l'exponentiel lors des calculs et ainsi éviter l'explosion vers l'infini pour les processeurs. La démonstration est directe :

En posant la constante $C = (\max_j)(z_j)$, et en multipliant le numérateur et le dénominateur de $a(\boldsymbol{z})_i$ par $e^{(-C)}$, on peut écrire pour chaque i :

$$a(\boldsymbol{z})_i = \frac{e^{z_i} \cdot e^{-C}}{e^{-C} \sum_{j=1}^{K} e^{z_j}} = \frac{e^{z_i - C}}{\sum_{j=1}^{K} e^{z_j - C}} = \frac{e^{z_i - \max_j(z_j)}}{\sum_{j=1}^{K} e^{z_j - \max_j(z_j)}}$$

Dérivée Jacobienne (matrice de dimension $K \times K$) :

$$J_{ij} = \frac{\partial a_i}{\partial z_j} = a(\boldsymbol{z})_i \left(\delta_{ij} - a(\boldsymbol{z})_j\right)$$

$$\text{où } \delta_{ij} = \begin{cases} 1 & si\ i = j \\ 0 & sinon \end{cases} \text{ est le delta de Kronecker.}$$

Ce qui donne explicitement :

$$J_{ij} = \begin{cases} a(\boldsymbol{z})_i \left(1 - a(\boldsymbol{z})_j\right) & si\ i = j \\ -a(\boldsymbol{z})_i \cdot a(\boldsymbol{z})_j & si\ i \neq j \end{cases}$$

Autrement dit, sur la diagonale de la matrice Jacobienne, la dérivée se comporte comme la Sigmoïde.

Propriétés principales :

1. Fonction multivariables scalaires ayant un vecteur comme variable
2. Varie strictement entre 0 et 1 pour chaque composante
3. $\sum_{i=1}^{K} a(\boldsymbol{z}_i) = 1$ *(distribution de probabilité)* autrement dit, il s'agit d'une distribution probabiliste avec des écarts amplifiés par l'exponentiel
4. Elle est Différentiable (Matrice **Jacobienne**)
5. Elle est invariante à la translation, autrement dit : $\mathrm{a}(\boldsymbol{z}) = \mathrm{a}(\boldsymbol{z} + c)$ *pour tout vecteur constant* c

Démonstration de la dérivée Jacobienne:

Posons : $S = \sum_{k=1}^{K} e^{z_k}$.

Alors en appliquant la règle de dérivée d'une fraction : $\left(\frac{f}{g}\right)' = \frac{f'g - g'f}{g^2}$, on peut écrire :

$$(1) \qquad \frac{\partial a_i}{\partial z_j} = \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{S}\right) = \frac{\frac{\partial}{\partial z_j}(e^{z_i}) \cdot S - \frac{\partial S}{\partial z_j} \cdot e^{z_i}}{S^2}$$

Or, on peut écrire selon que *i* soit égal ou différent de *j* :

$$\frac{\partial}{\partial z_j}(e^{z_i}) = e^{z_i}\,\delta_{ij}, \qquad \frac{\partial S}{\partial z_j} = e^{z_j}$$

En utilisant le symbole appelé communément le **delta de Kronecker** : $\delta_{ij} = \begin{cases} 1 & si\ i = j \\ 0 & sinon \end{cases}$, on peut écrire

$$(2) \qquad J_{ij} = \frac{e^{z_i}\,\delta_{ij} \cdot S - e^{z_j} \cdot e^{z_i}}{S^2} = \frac{e^{z_i}}{S}\left(\delta_{ij} - \frac{e^{z_j}}{S}\right)$$

$$= a(\boldsymbol{z})_i \left(\delta_{ij} - a(\boldsymbol{z})_j\right)$$

Démonstration de la dérivée forme compacte :

Pour simplifier, on va omettre le *z* et écrire :

Forme compacte de la dérivée de Softmax (la plus utilisée) :

$$J = diag(\boldsymbol{a}) - \boldsymbol{a}\boldsymbol{a}^T$$

où :

$$\boldsymbol{a}\boldsymbol{a}^T = \begin{pmatrix} a_1a_1 & \cdots & a_1a_K \\ \vdots & \ddots & \vdots \\ a_Ka_1 & \cdots & a_Ka_K \end{pmatrix} \in \mathbb{R}^{K\times K}$$

$$a_i = Softmax(z_i) = e^{(z_i)} / \left(\sum (j=1)^K e^{(z_j)}\right) \forall i \in 1,2,\dots,K$$

Usages appropriés :

1. Dans la couche de sortie pour les modèles de classification multi-classes
2. Elle est aussi utilisée entre autres dans le **mécanisme d'attention propre aux Transfomers** et les **KNN-LM**

$$J_ij = a_i(\delta_ij - a_j), \forall i,j \in 1,2,\dots,K$$

Le **delta de Kronecker** nous permet d'écrire :

$$\left(diag(\boldsymbol{a})\right)_{ij} = \delta_{ij}a_i, \forall i,j \in 1,2,\dots,K$$

A partir de la définition de la matrice aa^T, on peut écrire :

$$(\boldsymbol{a}\boldsymbol{a}^T)_{ij} = a_ia_j, \forall i,j \in 1,2,\dots,K$$

Par conséquent,

$$(diag(\boldsymbol{a}) - \boldsymbol{a}\boldsymbol{a}^T)_{ij} = \delta_{ij}a_i - a_ia_j = a_i(\delta_{ij} - a_j) = J_{ij}$$

La fonction **Softmax** étant **multidimensionnelle**, on va donner un exemple en 3 dimensions pour illustrer sa forme 3D dans l'espace $\left(z_1, z_2, softmax(z_1, z_2)\right)$.

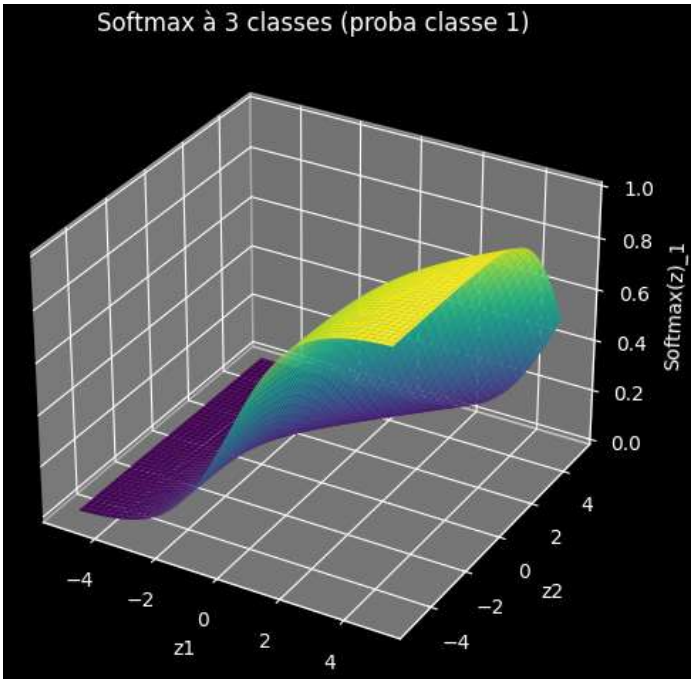
Exemple en Python en 3D (R^3) avec l'utilisation de la constante C :

$C = \underset{(j=1,2,3)}{(max)}\left(z_j\right)$, *voir dans le code ci-dessous la fonction* → ***def softmax(z):***

```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm

# Sélectionner un style dark pour les graphiques
plt.style.use("dark_background")


def softmax(z):
    """
    But: Calcule la fonction softmax de z.
    z: array de forme (..., C) où C est le nombre de classes
    Retourne: array de même forme que z, contenant les probabilités de chaque classe
    """

    C = np.max(z, axis=-1, keepdims=True)
    e = np.exp(z - C)
    return e / e.sum(axis=-1, keepdims=True)

# Exemple d'utilisation de la fonction softmax
# On fixe z3 = 0, on fait varier z1, z2
z1 = np.linspace(-5, 5, 100)
z2 = np.linspace(-5, 5, 100)
# On crée une grille de points (z1, z2)
# et on ajoute z3 = 0 pour former des vecteurs de logits à 3 classes
Z1, Z2 = np.meshgrid(z1, z2)
```

```
Z = np.stack([Z1, Z2, np.zeros_like(Z1)], axis=-1)  # dim = (N,N,3)
print("Z shape:", Z.shape)

S = softmax(Z)  # (N,N,3)
S1 = S[..., 0]  # <=> S1 = S[:, :, 0] prob de la classe 1
print("Softmax(z) shape:", S.shape)


# Visualisation de la probabilité de la classe 1 en fonction de z1 et z2
fig = plt.figure(figsize=(7,5))
ax = fig.add_subplot(111, projection='3d')
ax.plot_surface(Z1, Z2, S1, cmap=cm.viridis, edgecolor='none')
ax.set_xlabel("z1")
ax.set_ylabel("z2")
ax.set_zlabel("Softmax(z)_1")
ax.set_title("Softmax à 3 classes (proba classe 1)")
plt.tight_layout()
plt.show()
print("Exemples de softmax(z) pour quelques points z:")
for z in [[0,0,0], [5,0,0], [0,5,0], [-5,0,0], [0,-5,0], [2.5,-2.5,0], [-2.5,-2.5,0],
[2.5,2.5,0], [-2.5,2.5,0]]:
    print(f"z={z} => softmax(z)={softmax(np.array(z))}")
    print(f"Somme des probas: {softmax(np.array(z)).sum()}\n")
```

OUTPUT :



Z shape: (100, 100, 3)
Softmax(z) shape: (100, 100, 3)
Exemples de softmax(z) pour quelques points z:
z=[0, 0, 0] => softmax(z)=[0.33333333 0.33333333 0.33333333]
Somme des probas: 1.0

z=[5, 0, 0] => softmax(z)=[0.98670329 0.00664835 0.00664835]
Somme des probas: 1.0000000000000002

z=[0, 5, 0] => softmax(z)=[0.00664835 0.98670329 0.00664835]
Somme des probas: 1.0000000000000002

z=[-5, 0, 0] => softmax(z)=[0.00335766 0.49832117 0.49832117]
Somme des probas: 1.0

z=[0, -5, 0] => softmax(z)=[0.49832117 0.00335766 0.49832117]
Somme des probas: 1.0

z=[2.5, -2.5, 0] => softmax(z)=[0.91842297 0.00618829
0.07538875]
Somme des probas: 1.0000000000000002

z=[-2.5, -2.5, 0] => softmax(z)=[0.07050946 0.07050946
0.85898108]
Somme des probas: 0.9999999999999999

z=[2.5, 2.5, 0] => softmax(z)=[0.48028779 0.48028779
0.03942442]
Somme des probas: 1.0

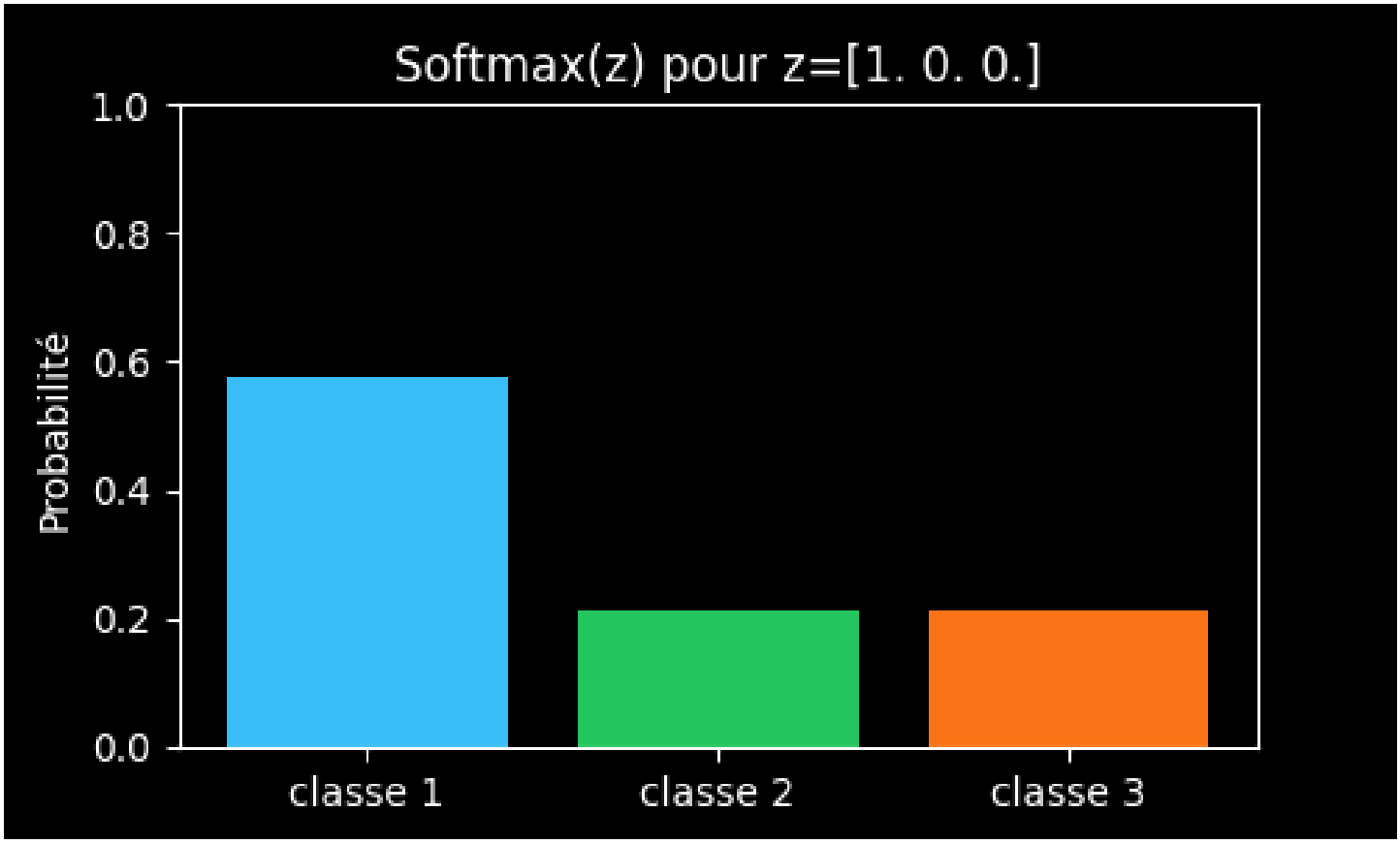
z=[-2.5, 2.5, 0] => softmax(z)=[0.00618829 0.91842297
0.07538875]
Somme des probas: 1.0000000000000002

Animation des probabilités de 3 classes :

```
import os
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

plt.style.use("dark_background")


def softmax(z):
    """
    But: Calcule la fonction softmax de z.
    z: array de forme (..., C) où C est le nombre de classes
    Retourne: array de même forme que z, contenant les probabilités de chaque classe
    """

    C = np.max(z, axis=-1, keepdims=True)
    e = np.exp(z - C)
    return e / e.sum(axis=-1, keepdims=True)

fig, ax = plt.subplots(figsize=(5,3))
bars = ax.bar([0,1,2],[0.33,0.33,0.34], color=["#38bdf8","#22c55e","#f97316"])
ax.set_ylim(0,1)
ax.set_xticks([0,1,2])
ax.set_xticklabels(["classe 1","classe 2","classe 3"])
ax.set_ylabel("Probabilité")
ax.set_title("Évolution de Softmax(z) quand z varie")

def update(frame):
    # z tourne autour d'un cercle dans ℝ³
    t = frame/50
    z = np.array([np.cos(t), np.sin(2*t), 0.5*np.sin(3*t)])
    p = softmax(z)
    for b, v in zip(bars, p):
        b.set_height(v)
    ax.set_title(f"Softmax(z) pour z={np.round(z,2)}")
    return bars

ani = FuncAnimation(fig, update, frames=200, interval=80, blit=False)
ani.save(os.getcwd() + "/softmax/softmax_animation.gif", fps=15)   # GIF: no FFmpeg needed
plt.show()
```

OUTPUT :

Cliquer sur l'image GIF pour voir l'animation

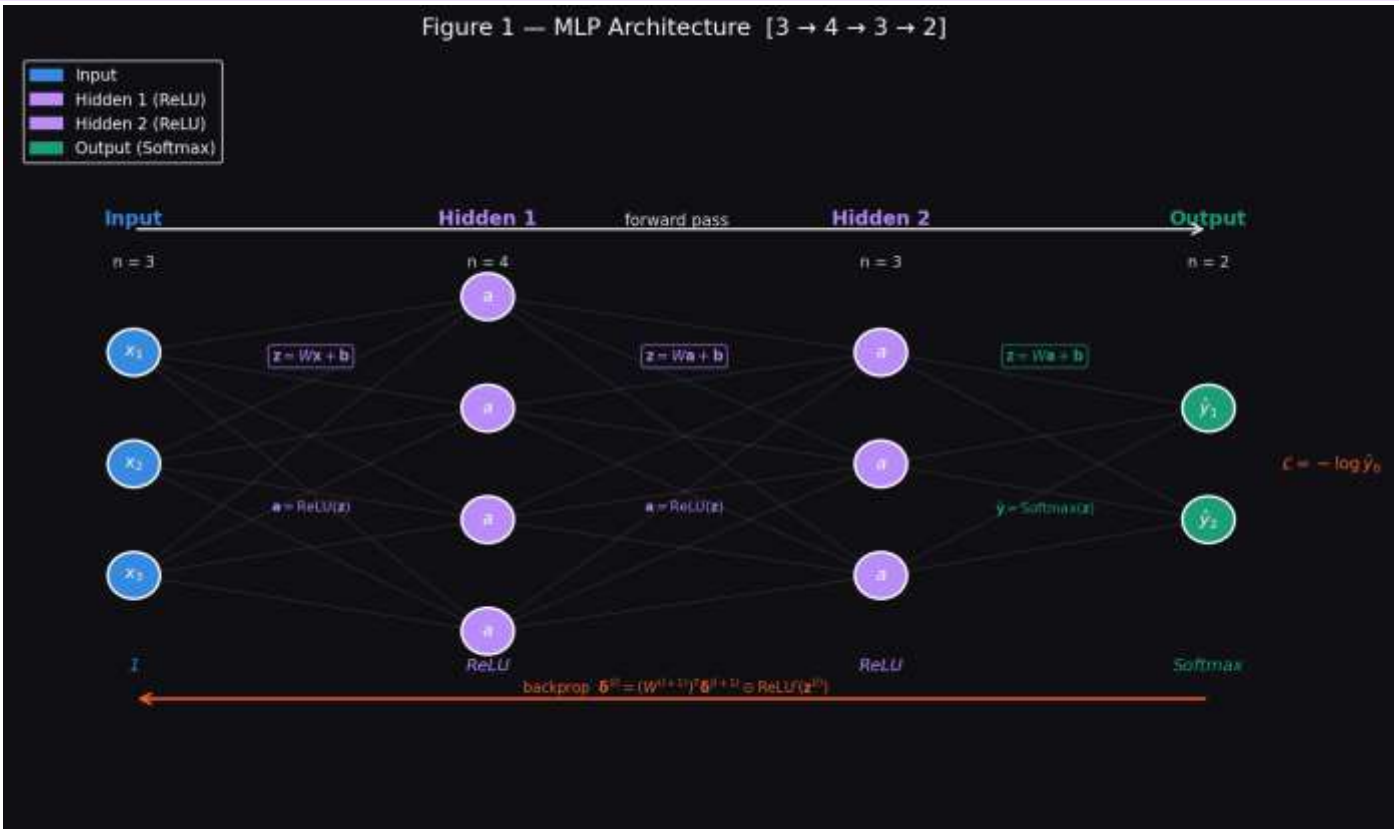

Softmax(z) pour z=[1. 0. 0.]
Probabilité
1.0
0.8
0.6
0.4
0.2
0.0
classe 1
classe 2
classe 3

VII. Résumé des fonctions d'activation les plus communes

Activation	Formule	Sortie	Dérivabilité	Usage typique régression
ReLU	$a(z) = max(0, z)$	$[0, +\infty)$	Oui (sauf 0)	Couches cachées standard
LeakyReLU	$LeakyReLU(z) = max(\alpha z, z)$	$\mathbb{R}$	Oui	Caches, évite neurones morts
ELU	$ELU(z) = max(\alpha z, z) = z$ Si $z > 0$, $\alpha(e^z - 1)$ Sinon	$(-\alpha, +\infty)$	Oui	Caches, plus lisse
SELU	$SELU(z) = max(\alpha z, z) = \lambda z$ Si $z > 0$, $\lambda\alpha(e^z - 1)$ Sinon	$\mathbb{R}$	Oui	Réseaux auto-normalisants
GELU	$GELU(z) \approx 0.5z\left(1 + tanh\left(\sqrt{(2/\pi)}(z + 0.044715z^3)\right)\right)$	$\mathbb{R}$	Oui	Profonds, style Transformer
Tanh	$a(z) = \left(e^z - e^{(-z)}\right)/\left(e^z + e^{(-z)}\right) = (e^{2z} - 1)/(e^{2z} + 1) = 2a(2z) - 1$	$[-1,1]$	Oui	Données centrées
Sigmoid	$a(z) = \sigma(z) = 1/(1 + e^{-z})$	$[0, 1]$	Oui	Sorties bornées/ classification binaire
Softplus	$SoftPlus(z) = log(1 + e^z)$	$(0, +\infty)$	Oui	Sorties ≥ 0
Softmax	$a(z)_i = e^{z_i}/\sum_j e^{z_j}$	$[0, 1]$	Oui	distribution probabiliste
Linéaire	$a(z) = z$	$\mathbb{R}$	Oui	Sortie régression

A1.4 Le schéma d'un modèle MLP simple

Architecture MLP simple (Classification)

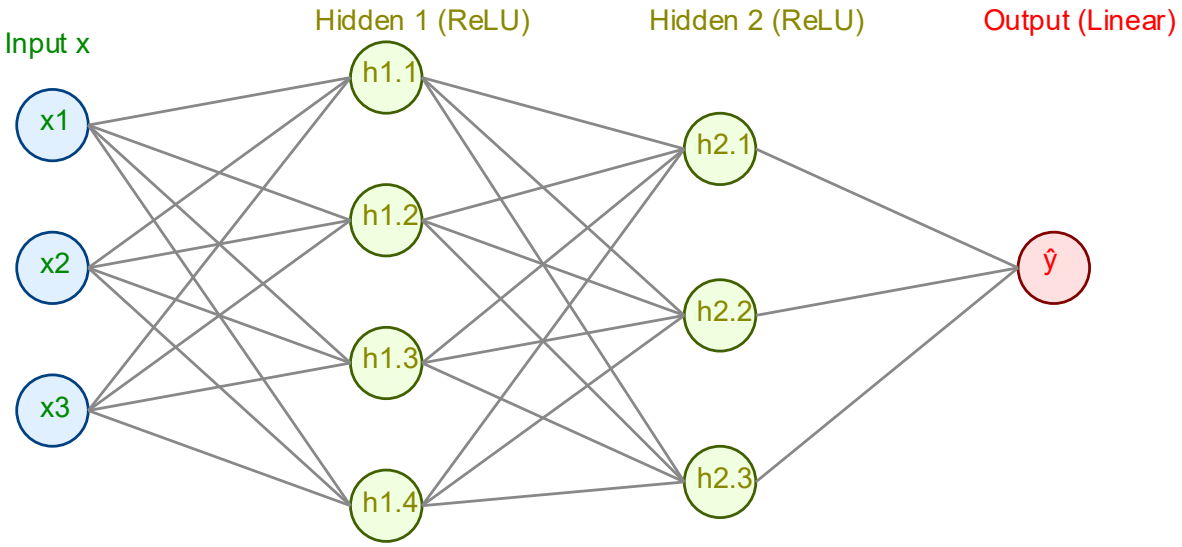


$$L = 3(4\ couches\ en\ débutant\ \text{à}\ 0) - l = 0,1,2,3$$

$$\text{Input}(n_0 = 3) \rightarrow \text{Hidden 1}(n_1 = 4,\ \text{ReLU}) \rightarrow \text{Hidden 2}(n_2 = 3,\ \text{ReLU}) \rightarrow \text{Output}(n_3 = 2,\ \text{Softmax})$$

Cliquer pour voir l'animation:

Architecture MLP simple (Régression)



$$L = 3\ (4\ couches\ en\ débutant\ \text{à}\ 0) - l = 0,1,2,3$$

I. *Définitions et notations*

1. **Matrice & Vecteur :** Soit $A \in \mathbb{R}^{(n \times p)}$. On note $A^{((i))}$ la i-ème colonne de A, pour $i \in 1, \ldots, p$. Chaque $A^{((i))}$ est considéré comme étant un vecteur de dimension n, i.e., $A^{((i))} \in \mathbb{R}^n$. Cette désignation sera utilisée ci-dessous pour l'ensemble des variables du réseau : $z, W, a\ et\ b$. Une matrice formée d'une seule colonne ($A \in$

$\mathbb{R}$^$(n \times 1)$), peut être considérée tout simplement comme un vecteur dont la dimension est égale au nombre de lignes n.

2. Notions utilisées lors de déroulement d'une traversée avant (FORWARD PASS) et d'une traversée retour (BACKPROPAGATION) d'un réseau :

Terme	Définition	Scope des data	Valeurs typiques
EPOCH	C'est une passe totale sur toutes les données du jeu d'entrainement (Training set. Tous les **batches** faisant partie d'une EPOCH (épopée) sont procédée une seule fois.	Uniquement le training set	10 – 300 (NNs) 1 (ML classiques)
BATCH	Il s'agit d'un sous-ensemble du jeu d'entrainement procédé par l'ensemble des étapes (**Steps**) de traitement à l'aller (Passe avant) et au retour (Passe arrière).	Pendant le training	32, 64, 128, 256, …
STEP/ ITERATION	Il s'agit d'une étape pour mettre à jours les poids W associés aux variables ***X faisant partie du même seul batch***. Donc : $\#\ steps\ =\ \lfloor N\_train\ set\ /\ batch\_size \rfloor$	Pendant le training	$\lfloor N_{\text{train}}/\text{batch\_size} \rfloor$
VALIDATION	Au sein d'une EPOCH avant même que les poids et les biais soient mis à jour par la passe arrière. Cette phase effectue une validation du modèle sur le set de validation. Elle est utilisée pour principalement arrêter les EPOCH s'il n'y a plus d'amélioration au niveau de l'Accuracy et/ou au niveau de la marge d'erreur (validation loss), ce qu'on appelle le « early stopping », arrêt prématuré, car le modèle ne semble plus s'améliorer.	Jeu de validation uniquement	Généralement à chaque 1–5 epochs
TEST SET	C'est la dernière évaluation du modèle une fois les poids et les biais établis (donc le modèle fixé) sur des données que ce dernier n'a jamais traitées.	Jeu de test uniquement	1 seule fois à la toute fin

II. Enchainement des traitements (EPOCHs, batches…)

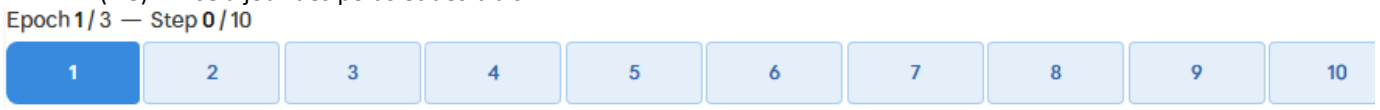
Le schéma suivant illustre ces termes et l'enchainement des traitements :

1. **Dimension du dataset :** 10 000 données
2. **Répartition :**
   a) **Jeu de données d'entrainement (training set) :** 70% (7 000 échantillons)
   b) **Jeu de données de validation (validation set) :** 15% (1 500 échantillons)
   c) **Jeu de données de test (test set) :** 15% (1 500 échantillons)
3. **Nombre d'epochs fixé au préalable :** n_epochs = 3
4. **Taille de chaque batch (Passe AVANT + Passe ARRIERE) :** 700 échantillons ➔ **Donc :**
   a) **On doit traiter :** 7 000/700 = 10 batches/epoch (en pratique on fixe le # de batches pour déterminer la taille de chaque échantillon)
   b) **Total des étapes (Steps) :** n_epochs x 10 = 30 étapes en tout

Le schéma suivant illustre visuellement les dimensionnements et les détails de l'enchainement décrit ci-dessus.

1

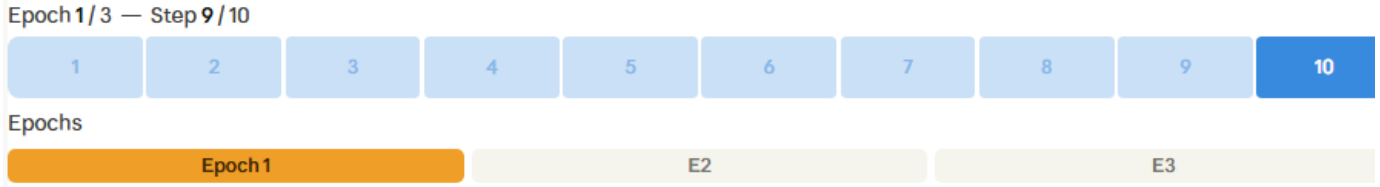
- **EPOCH 1** = Passe sur tous les **7 000 échantillons du training set**.
- batch_size=700 ➔ N d'étapes = 10 pour entrainer le modèle sur les 7 000 échantillons
- Pour chaque batch : **Passe AVANT** → Evaluation de la fonction perte → **Passe ARRIERE** → Gradient Descendant (DG) : mise à jour des poids et des biais



- **Step 1 terminé :** [0…699] échantillons

- **Step 2 terminé :** [700…1399] échantillons

…



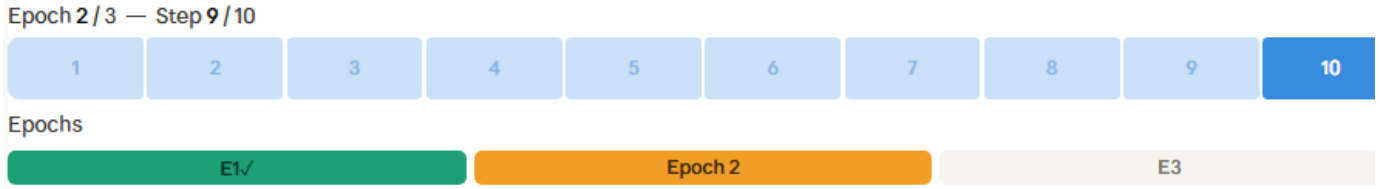
- **Step 10 terminé :** [6300…6999] échantillons

✓ **EPOCH 1 terminée.** Tous les batchs ont été traités ➔ **7 000 échantillons ont été visités**.
✓ Enclenchement de la **Validation** ➔ **val_loss** and **val_acc** calculées, *sans mise à jour des poids*.

2

…

- **EPOCH 2 : Traitement des 10 batchs de l'EPOCH 2**

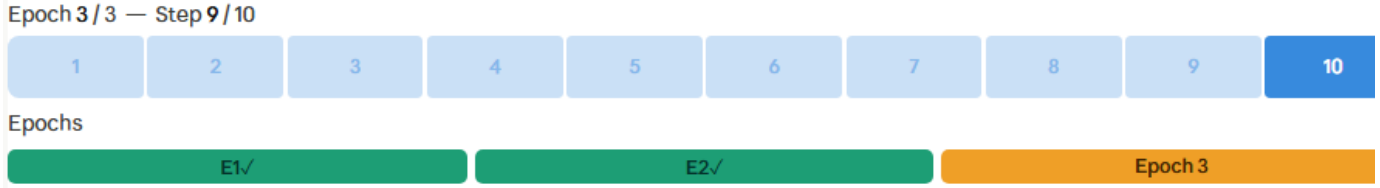


✓ **EPOCH 2 terminée.** Tous les batchs ont été traités ➔ **7 000 échantillons ont été visités**.
✓ Enclenchement de la **Validation** ➔ **val_loss** and **val_acc** calculées, *sans mise à jour des poids*.

3

- **EPOCH 3 : Traitement des 10 batchs de l'EPOCH 3**

…



✓ **EPOCH 3 terminée.** Tous les batchs ont été traités ➔ **7 000 échantillons ont été visités**.
✓ Enclenchement de la **Validation** ➔ **val_loss** and **val_acc** calculées, *sans mise à jour des poids*.

NOTE : Selon les critères imposés, il est possible que le traitement s'arrête à n'importe quelle EPOCH et à n'importe quelle Step

A1.5 Architecture (MLP classification) en résumé

$[\![\boldsymbol{Input} \to n]\!]\_0 = 3,\ \boldsymbol{Hidden\ 1} \to n\_1 = 4,\quad [\![\boldsymbol{Hidden\ 2} \to n]\!]\_2 = 3,\ [\![\boldsymbol{Output} \to n]\!]\_3 = 2$

Couche	Dimension	Fonction d'activation
Input	3	—
Hidden 1	4	$ReLU(z) = max(0,\ z) \quad \to \boldsymbol{Dérivée}: \quad \mathrm{ReLU}'(z) = \mathbf{1}_{(z>0)}.$
Hidden 2	3	ReLU
Output	2	$\mathrm{Softmax}(\mathbf{z})_k = \frac{e^{z_k - \max_j(z_j)}}{\sum_j e^{z_j - \max_j(z_j)}}$, dérivée partielle : $\frac{\partial \mathrm{SMax}(\mathbf{z})_k}{\partial z_j} = \mathrm{SMax}(\mathbf{z})_j\left(\delta_{kj} - \mathrm{SMax}(\mathbf{z})_j\right)$

Input : $\mathbf{x} = [1,\ 4,\ 5]^T$, $\mathbf{y} = [1,\ 0]^T$ (classes 0/1).
Coefficient d'apprentissage η généralement utilisé variant entre 0,001 juqu'à 0,1. Dans notre cas on utilisera $\eta = 0.1$.

A1.6 Le principe des passes AVANT et ARRIERE (FORWARD + BACKPROPAGATION)

- La **passe avant (FORWARD PASS)** sert à définir le problème d'optimisation au travers la fonction de perte comme fonction objective à minimiser.

- La **passe arrière (BACKPROPAGATION)** sert via l'utilisation de la technique du gradient descendant à déterminer les classes associées dans notre cas (0 ou 1) ainsi que les coefficients optimaux approximatifs à être utilisés par le modèle pour les prédictions.

- Chaque passe avant suivie d'une passe arrière, représente un cycle appelé une époque (EPOCH) :
  a. Chaque époque (EPOCH) est exécutée sur deux sous-ensembles, le **jeu d'entrainement (training set)** et le **jeu de validation (validation set)** avec généralement une proportion relative au jeu de données de 70% à 80% pour le « trainig set » et 30% à 20% pour le « validation set ». Le nombre d'époques est fixé au préalable

  b. Dans le cas où on souhaite tester le modèle avec les paramètres retenus (ici ***W_best*** et ***b_best***), on réserve un sous-jeu de données réservé aux tests (voir un tel scénario dans la section : Enchainement des traitements (EPOCHs, batches…) sur lequel on teste notre modèle, autrement dit, on a :
     i. **trainig set** → 70% → pendant les EPOCHs
     ii. **validation set** → 15%→ pendant les EPOCHs
     iii. **testing set** → 15% → après la sortie des EPOCHs avec les paramètres retenus (les meilleurs)

  c. C'est lors de l'application du modèle sur les données du jeu d'entrainement que les paramètres *W* et *b* sont estimés via l'application de la méthode gradient descendant. Ces derniers sont utilisés et appliqués sur le jeu de données réservé à la validation (phase ii) afin de déterminer les métriques de performances (Valeur de l'Accuracy, Valeur de la perte (loss)) → val_acc et val_loss

  d. Une fois les meilleurs paramètres du modèle établis (ici ***W_best*** et ***b_best***), on le teste sur le **« testing set** » afin de mesurer sa capacité de prédiction sur des données qu'il n'a jamais connues !

A1.7 Détails du fonctionnement du réseau MLP

Les différentes variables qui interviennent dans les calculs ci-dessous n'ont pas les mêmes types et dépendent de la dimension de x (dataset, features), de *W* (poids) et des biais (*b*); il est important de résumer le type de chaque variable impliquée :

Variables en entrée/sortie (Pour chaque couche d'ordre l)	**Type**	**Dimension**
X	Vecteur	$n_0 \times 1$
$W^{(l)}$	Matrice	$n_l \times n_{l-1}$
$\mathbf{a}^{(l-1)}$	Vecteur colonne	$n_{l-1} \times 1$
$\mathbf{b}^{(l)}$	Vecteur colonne	$n_l \times 1$
$delta\ de\ Kronecke$: $\boldsymbol{\delta}^{(l)}$	Vecteur colonne	$n_l \times 1$

Cela définit implicitement les dimensions des différentes dérivées partielles (voir les démonstrations plus bas) que l'on va utiliser lors de l'application du gradient descendant et dont le résumé est le tableau suivant :

Dérivées impliquées (Pour chaque couche d'ordre l)	**Type**	**Dimension**
Fonction de perte $\mathcal{L}$:		
$\frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(L)}} = \hat{\mathbf{y}} - \mathbf{y}$	Vecteur colonne	$n_l \times 1$
$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$	Vecteur colonne	$n_l \times 1$
$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \boldsymbol{\delta}^{(l)} \cdot \left(\mathbf{a}^{(l-1)}\right)^T$	Matrice	$n_l \times n_{l-1}$

I. Dimensions des couches de notre exemple

- **Couche 0 (Input)** : $\mathbf{x} = (x_1, x_2, x_3)^T = (1,4,5)^T \in \mathbb{R}^{n_0}$
  Remarque : on peut aussi ajouter la constante 1 comme 4ème composant par laquelle le biais *b* est multiplié à l'entrée de la 1ère couche cachée (**Hidden 1**). Autrement dit, le vecteur des poids *W* est augmenté d'une dimension avec le biais *b* comme valeur du dernier composant.
- **Couche 1 (Hidden 1, ReLU)** : $W^{(1)} \in \mathbb{R}^{4\times3}$, $\mathbf{b}^{(1)} \in \mathbb{R}^4$
- **Couche 2 (Hidden 2, ReLU)** : $W^{(2)} \in \mathbb{R}^{3\times4}$, $\mathbf{b}^{(2)} \in \mathbb{R}^3$
- **Couche 3 (Output, Softmax)** : $W^{(3)} \in \mathbb{R}^{2\times3}$, $\mathbf{b}^{(3)} \in \mathbb{R}^2$

On note, pour chaque couche l :

1. $\boldsymbol{z}^{((l))}$: pré-activation (somme pondérée par les poids *W*)
2. $\boldsymbol{a}^{((l))}$: activation (sortie de la couche par application de la fonction **ReLU** ou **Softmax** selon l'ordre de la couche)

II. Passe avant (Forward Pass) :

Résumé :

Pour chaque couche $l = 1, \dots, L$:

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

$$\mathbf{a}^{(l)} = \begin{cases} \mathrm{ReLU}\left(\mathbf{z}^{(l)}\right) & \text{si } l < L \\ \mathrm{Softmax}\left(\mathbf{z}^{(L)}\right) & \text{si } l = L \end{cases}$$

Appliquer la fonction de perte (Cross-entropy loss) sur le output ($l = L$) :

$$\mathcal{L} = -\sum_{k} y_k \log \hat{y}_k = -\log \hat{y}_{\text{vraie classe}}$$

En détails :

Couche 1 (dimension 3 → dimension 4, ReLU)

$$\mathbf{z}^{(1)} = W^{(1)}\,\mathbf{a}^{(0)} + \mathbf{b}^{(1)} \in \mathbb{R}^4, \qquad \mathbf{a}^{(0)} = \mathbf{x} = (1,4,5)^T$$

$$z_i^{(1)} = \sum_{j=1}^{3} W_{ij}^{(1)}\, x_j + b_i^{(1)}, \qquad a_i^{(1)} = \mathrm{ReLU}\left(z_i^{(1)}\right) = \max\left(0, z_i^{(1)}\right)$$

Couche 2 (4 → 3, ReLU)

$$\mathbf{z}^{(2)} = W^{(2)}\,\mathbf{a}^{(1)} + \mathbf{b}^{(2)} \in \mathbb{R}^3$$

$$z_k^{(2)} = \sum_{i=1}^{4} W_{ki}^{(2)}\, a_i^{(1)} + b_k^{(2)}, \quad k = 1,2,3, \qquad a_k^{(2)} = \max\left(0, z_k^{(2)}\right)$$

Couche Output (3 → 2 Softmax)→ Fonction de perte (cross-entropy) appliquée

$$\mathbf{z}^{(3)} = W^{(3)}\,\mathbf{a}^{(2)} + \mathbf{b}^{(3)} \in \mathbb{R}^2$$

$$z_i^{(3)} = \sum_{k=1}^{3} W_{ik}^{(3)}\, a_k^{(2)} + b_i^{(3)}, \quad i = 1,2$$

$$\hat{y}_m = \mathrm{softmax}\left(\mathbf{z}^{(3)}\right)_m = \frac{e^{z_m^{(3)}}}{\sum_{i=1}^{2} e^{z_i^{(3)}}}$$

$$\mathcal{L} = -\sum_{m=1}^{2} y_m \log \hat{y}_m = -\log \hat{y}_{\text{classe vraie}}$$

➔Résumé de la phase FORWARD (Passe **AVANT**)

N°	Étape	Vecteur associé aux perceptrons de la couche	Activation (a)
0	Input	$\mathbf{x} = (x_1, x_2, x_3)^T \in \mathbb{R}^3$	—
1	Input → Hidden 1	$\mathbf{z}^{(1)} = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$	$\mathbf{a}^{(1)} = \mathrm{ReLU}(\mathbf{z}^{(1)})$
2	Hidden 1 → Hidden 2	$\mathbf{z}^{(2)} = W^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}$	$\mathbf{a}^{(2)} = \mathrm{ReLU}(\mathbf{z}^{(2)})$
3	Hidden 2 → Output	$\mathbf{z}^{(3)} = W^{(3)}\mathbf{a}^{(2)} + \mathbf{b}^{(3)}$	$\hat{\mathbf{y}} = \mathrm{Softmax}(\mathbf{z}^{(3)})$
4	Perte à minimiser	$\mathcal{L} = -\sum_{m=1}^{2} y_m \log \hat{y}_m$	

III. Passe arrière, Rétropropagation (BACKPROPAGATION)

Le but est de déterminer les valeurs (approximatives) des coefficients *W* et des biais *b* pour lesquelles la fonction de perte atteint son minimum (i.e. solution optimale). A bien noter qu'à chaque passe avant, la fonction de perte n'est pas la même et de même pour la solution optimale (approximative) associée et déterminée via la passe arrière.

Pour cela, on a besoin principalement de deux dérivées partielles :

a) **Delta en sortie** (**delta de Kronecker** $\boldsymbol{\delta}^{(L)}$) :

$$\boldsymbol{\delta}^{(L)} = \hat{\mathbf{y}} - \mathbf{y}$$

b) **Gradients des poids** (règle de dérivation en chaîne) **:**

$$\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial W^{(l)}} = \boldsymbol{\delta}^{(l)} \cdot \left(\mathbf{a}^{(l-1)}\right)^T$$

car $\mathbf{z}^{(l)} = W^{(l)}\,\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \Rightarrow \frac{\partial \mathbf{z}^{(l)}}{\partial W^{(l)}} = \mathbf{a}^{(l-1)}$.

c) **Gradient des biais :**

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}^{(l)}} \cdot \frac{\partial \mathbf{z}^{(l)}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)} \cdot 1 = \boldsymbol{\delta}^{(l)}$$

d) **Propagation récursive du delta vers l'arrière** :

$$\boldsymbol{\delta}^{(l-1)} = \left(\left(W^{(l)}\right)^T \boldsymbol{\delta}^{(l)}\right) \odot f'\left(\mathbf{z}^{(l-1)}\right), \quad l = L, L-1, \ldots, 2$$

Le symbole ⊙ désigne l'opérateur de multiplication élément par élément des deux matrices.

$\frac{\partial \mathcal{L}}{\partial z_j^{(L)}} = \sum_{i=1}^{K} \frac{\partial \mathcal{L}}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{\partial z_j^{(L)}}$ (L = 3), par application de la dérivée partielle en chaine

Or :

$$\frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \frac{\partial}{\partial \hat{y}_i}(-y_i \log \hat{y}_i) = -\frac{y_i}{\hat{y}_i}$$

$$\frac{\partial \hat{y}_i}{\partial z_j^{(L)}} = \hat{y}_i\left(\delta_{ij} - \hat{y}_j\right) \quad \text{(dérivée Softmax)}$$

D'où :

$$\frac{\partial \mathcal{L}}{\partial z_j^{(L)}} = -\sum_{i=1}^{K} \frac{y_i}{\hat{y}_i} \cdot \hat{y}_i\left(\delta_{ij} - \hat{y}_j\right) = -\sum_{i=1}^{K} y_i\left(\delta_{ij} - \hat{y}_j\right) = \hat{y}_j - y_j$$

←Résumé de la phase BACKPROPAGATION (Passe **ARRIERE**)

Étape	Transition des couches $(L = 3)$	Mise à jour des variables
0	**Output** $(l = L)$	Delta de sortie à utiliser dans les étapes suivantes : $\boldsymbol{\delta}^{(L)} = \hat{\mathbf{y}} - \mathbf{y}$
1	**Output → Hidden 2** $(l = L - 1)$	Pour chaque l, utiliser successivement les formules:
2	**Hidden 2 → Hidden 1** $(l = L - 2)$	a) **Calculer le delta de Kronecker de l'étape** l : $\boldsymbol{\delta}^{(l)} = \left(W^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)} \odot ReLU'\left(\mathbf{z}^{(l)}\right)$ b) **Calculer les gradients :** $\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \boldsymbol{\delta}^{(l)}\left(\mathbf{a}^{(l-1)}\right)^T, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$ c) **Mettre à jour les poids et les biais du modèle :** $\mathrm{W}^{(l+1)} \leftarrow \mathrm{W}^{(l)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathrm{W}^{(l)}}$ $\mathbf{b}^{(l+1)} \leftarrow \mathbf{b}^{(l)} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}}$

Le coefficient $\boldsymbol{\eta}$ appelé le **learning rate** (pas d'apprentissage), sert à corriger le pas de la descente du gradient et donc l'amplitude des valeurs obtenues. En pratique et généralement, il est fixé entre 0,001 et 0,1, voire ajusté dynamiquement, car il influe sur le nombre d'itérations, i.e., nombre des FORAWRD+BACKPROPAGATION nécessaire pour converger selon les critères imposés (val_loss ↘ et val_accuracy ↗ sur un # prédéfini d'EPOCHs ou d'itérations à fixer au préalable).

Le même principe s'applique pour un réseau MLP dont le nombre de couches intermédiaires cachées est > 2, dont la dimension des variables du jeu de données (features) est > 3 et dont les fonctions d'activation intermédiaires et finale sont différentes de celles utilisées dans notre réseau exemple MLP de classification. Vous avez aussi la possibilité d'utiliser vos propres fonctions d'activation à des fins d'expérimentation ou dans l'optique d'améliorer les performances obtenues par le modèle ; voir dans la suite une telle implémentation.

A1.8 Application des Passes avant et arrière (FORWARD + BACKPROPAGATION) - Classification

Référez-vous aux 2 parties : (Résumé de la phase FORWARD (Passe avant) et (Résumé de la phase BACKPROPAGATION (Passe arrière).

Le lien du code Python se trouve dans l'espace GITHUB : https://github.com/ayezza/mlp_basique

Le principe de l'exemple est simple :

Architecture – Paramètres - Training/Validation (Passes AVANT + ARRIERE)

Etape	Brique concernée	Eléments associés
Entrées		

A	**Architecture du réseau :**
	• 3 features en Input
	• 4 neurones dans Hidden 1
	• 3 neurones dans Hidden 2
	• 2 neurones dans output (les probas des 2 classes)
		`LAYER_SIZES = [3, 4, 3, 2]`
		`LAYER_NAMES = ["Input", "Hidden 1", "Hidden 2", "Output"]`
		`LAYER_ACTS = ["-", "ReLU", "ReLU", "Softmax"]`
		`activations = [relu, relu, softmax]`

B	**Hyperparamètres pour le training/validation :**
	(voir les commentaires en # des variables)
	2 critères sont imposés :
	3. **Critère 1 :** Si la fonction de perte n'atteigne pas un seuil minimum
	4. **Critère 2 :** Si aucune amélioration de l'Accuracy pendant un nombre d'EPOCHs maximum
	Si l'un des critères n'est pas satisfait on arrête l'entrainement.
		`LR = 0.05 # taux d'apprentissage`
		`MAX_EPOCHS = 100 # nombre maximum d'epochs`
		`LOSS_TOL = 1e-4 # seuil de convergence de la perte (critère 1)`
		`ACC_PATIENCE = 5 # patience pour le plateau d'accuracy (critère 2)`

C	**Jeu de données (généré aléatoirement) avec une classification binaire :**
	Le Seed n'étant pas fixe, les résultats ne sont pas les mêmes d'une exécution à une autre !) ; c'est ce qui définira les paramètres du modèle sur la base des données en entrée.
		`X, Y = make_dataset(n_per_class=30, seed=None) # 60 échantillons, 2 classes`
		
		`# Split train (70 %) / validation (30 %)`
		`n_train = int(0.7 * len(X))`
		`X_train, Y_train = X[:n_train], Y[:n_train]`
		`X_val, Y_val = X[n_train:], Y[n_train:]`

D	**Initialisation des poids et des biais :**	`np.random.seed(42)` `weights = [np.random.normal(0, 0.5, (n_out,n_in))`

	Au tout 1er passage dans le réseau MLP les poids et les biais sont initialisés d'une manière aléatoire	`for n_in, n_out in zip(LAYER_SIZES[:-1], LAYER_SIZES[1:])]`
		`biases = [np.random.normal(0, 0.1, n_out)`
		`for n_out in LAYER_SIZES[1:]]`

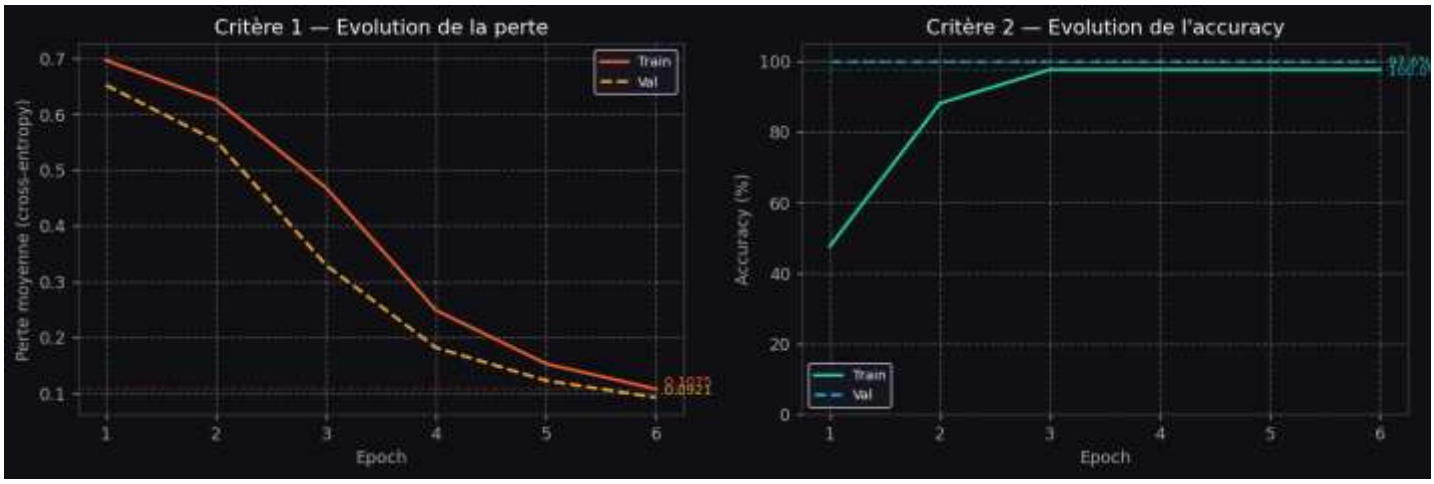
Entraînement/ Validation	Entrainer le train set et valider sur le validation set en déroulant les EPOCHs et en appliquant les critères d'arrêt à chaque époque (voir **B** ci-dessus)	`weights, biases, history = train(`
		`weights, biases, activations, X_train, Y_train,`
		`lr = LR,`
		`max_epochs = MAX_EPOCHS,`
		`loss_tol = LOSS_TOL,`
		`acc_patience = ACC_PATIENCE,`
		`verbose_first_epoch = True,`
		`X_val = X_val,`
		`Y_val = Y_val,`
		`)`

Graphes	Tracer les courbes de tendance des pertes et des Accuracy obtenues pour le training et la validation en fonction des EPOCHs	`plot_history(history)`

Sortie partielle en texte

```
 Dataset total  : 60 échantillons, 3 features, 2 classes
  Train         : 42 échantillons  (21 x cl-0  |  21 x cl-1)
  Validation    : 18 échantillons  (9 x cl-0  |  9 x cl-1)

  Architecture : Input(3) -> Hidden 1(4) -> Hidden 2(3) -> Output(2)
  lr=0.05  |  loss_tol=0.0001  |  acc_patience=5
…
============================================================
  BOUCLE D'ENTRAÎNEMENT (suite)
============================================================
   Epoch    Perte(tr)   Δperte(tr)   Acc(tr)  |   Perte(val)   Δperte(v)  Acc(val)   Pat.
  ------------------------------------------------------------------------------------------
       1     0.696457          inf     47.6%  |     0.652190         inf    100.0%      0
       2     0.625115     7.13e-02     88.1%  |     0.552410    9.98e-02    100.0%      0
       3     0.468068     1.57e-01     97.6%  |     0.329396    2.23e-01    100.0%      1
       4     0.248531     2.20e-01     97.6%  |     0.181447    1.48e-01    100.0%      2
       5     0.152596     9.59e-02     97.6%  |     0.122567    5.89e-02    100.0%      3
       6     0.107458     4.51e-02     97.6%  |     0.092139    3.04e-02    100.0%      4

  ============================================================
  ARRÊT à l'epoch 6 : Critère 2 — accuracy val stable à 100.0% depuis 5 epochs
  Perte finale    (train) : 0.107458   Acc : 97.6%  (41/42)
  Perte finale    (val)   : 0.092139   Acc : 100.0%  (18/18)
  ============================================================
```

Graphes : évolution de la perte et de l'Accuracy par EPOCH



A1.9 Application des Passes sur des datasets réputés - Classification

Nous allons appliquer le même code, mais cette fois sur des vrais jeux de donnés de référence académique que nous allons charger à partir de la librairie **klearn.datasets**. Pour ce faire :

1. Se référer au repository (les datasets sont déjà extraits) : https://github.com/ayezza/mlp_basique
2. Faire un clone via GIT (en ligne de commande) ou via GITHUB Desktop en mode UI sur votre machine
3. Charger les datasets (format Numpy compréssé **.npz**) en exécutant **generate_dataset_clf.py**:

```
(a) data, dataset_name = load_iris(),          "iris"
(b) data, dataset_name = load_wine(),          "wine"
(c) data, dataset_name = load_breast_cancer(), "breast_cancer"
(d) data, dataset_name = load_digits(),        "digits"
```

4. Pour tester chacun des datasets :
   - **(a)** Compléter le fichier des paramètres JSON **params_clf.json**, ex., pour le fameux dataset **IRIS** (voir les résultats plus bas) :

```
{
    "parameters": {
        "NPZ_DATASET": "iris.npz",
        "LAYER_SIZES": [4, 16, 32, 3],
        "LAYER_NAMES": ["input", "hidden1", "hidden2", "output"],
        "LAYER_ACTS": ["-", "ReLU", "ReLU", "Softmax"],
        "LR": 0.001,
        "MAX_EPOCHS": 1000,
        "LOSS_TOL": 1e-8,
        "ACC_PATIENCE": 20,
        "BATCH_SIZE": 5,
        "WARMUP_EPOCHS": 20,
        "RANDOM_SEED": 42
    }
}
```

5. Lancer le script : **mlp_classif_model.py** qui prendra automatiquement le dataset indiqué ainsi que les autres paramètres associés **(à adapter pour chaque dataset !)**

Voici les résultats obtenus (peuvent être différents des vôtres !) matérialisés par l'évolution de la perte (**loss**) et de **l'Accuracy train set** vs **validation set** :

Les performances obtenues sont très honorables avec des durées des traitements (EPOCH/BATCHES) négligeables

Description du dataset sur UCI : Forest Cover Type Dataset

Extrait du output :

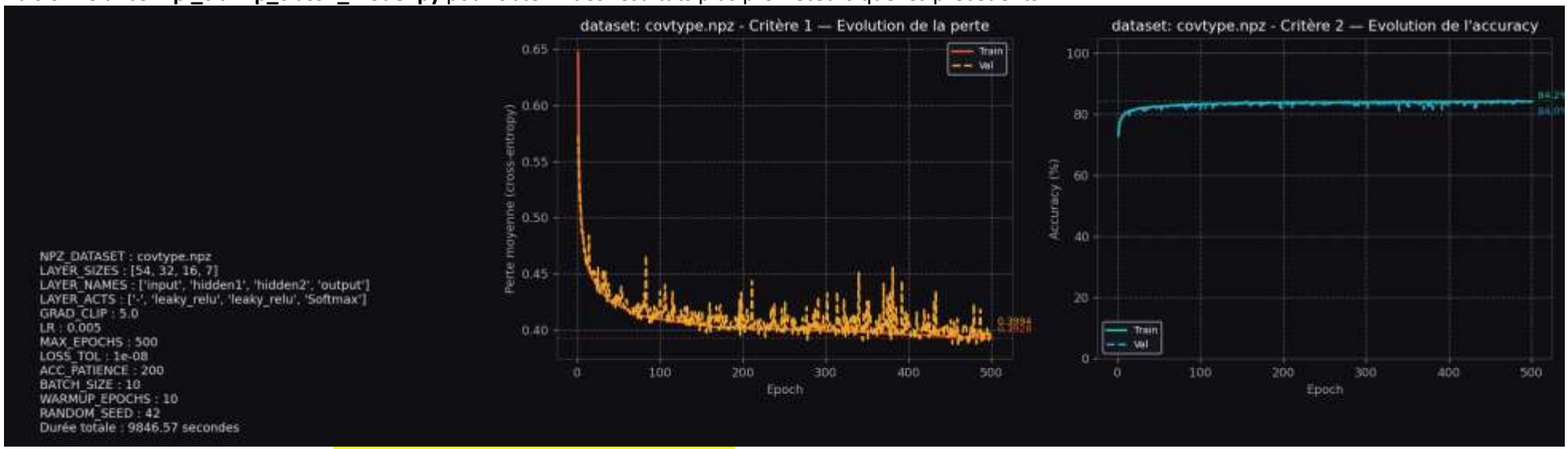
```
Paramètres chargés depuis 'params_clf.json' :
    NPZ_DATASET : covtype.npz
    LAYER_SIZES : [54, 16, 32, 7]
    LAYER_NAMES : ['input', 'hidden1', 'hidden2', 'output']
    LAYER_ACTS : ['-', 'ReLU', 'ReLU', 'Softmax']
    LR : 0.001
    MAX_EPOCHS : 500
    LOSS_TOL : 1e-08
    ACC_PATIENCE : 200
    BATCH_SIZE : 10
    WARMUP_EPOCHS : 10
    RANDOM_SEED : 42

  Dataset chargé depuis 'covtype.npz' : 581012 échantillons, 54 features, 7 classes
  Dataset total  : 581012 échantillons, 54 features, 7 classes  [normalisé Z-score]
  Train          : 406708 échantillons  (148175 x cl-0  |  198338 x cl-1  |  25047 x cl-2  |  1919 x cl-3  |  6669 x cl-4  |  12168 x cl-5  |  14392 x cl-6)
  Validation     : 174304 échantillons  (63665 x cl-0  |  84963 x cl-1  |  10707 x cl-2  |  828 x cl-3  |  2824 x cl-4  |  5199 x cl-5  |  6118 x cl-6)

  Architecture : input(54) -> hidden1(16) -> hidden2(32) -> output(7)
  lr=0.001  |  loss_tol=1e-08  |  acc_patience=200  |  batch_size=10  |  warmup=10

============================================================
  BOUCLE D'ENTRAÎNEMENT  (lr=0.001, max=500 epochs, mini-batch (batch=10))
============================================================

  Critère 1 : |perte|   < 1e-08  (convergence de la perte)
  Critère 2 : accuracy stable depuis 200 epochs  (plateau)
  Warmup    : 10 epochs  (critères inactifs pendant cette période)

Epoch    Perte(tr)        Δperte(tr)   Acc(tr)  |   Perte(val)  perte(v)     Acc(val)   Pat.
      1    0.622188          inf      73.9%     |   0.555647        inf      76.4%      0
      2    0.533932      8.83e-02     77.2%     |   0.514871    4.08e-02     78.1%      0
      3    0.499604      3.43e-02     78.8%     |   0.486930    2.79e-02     79.3%      0
      4    0.475893      2.37e-02     79.8%     |   0.468956    1.80e-02     80.2%      0
…
    490    0.299089      8.37e-04     88.7%     |   0.326221    1.30e-02     88.0%     77
    500    0.298571      1.20e-04     88.7%     |   0.294832    1.13e-02     88.9%     87

  ARRÊT à l'epoch 500 : max epochs (500) atteint
  Perte finale    (train) : 0.298571   Acc : 88.7%  (360837/406708)
  Perte finale    (val)   : 0.294832   Acc : 88.9%  (155009/174304)
```

Graphes : Vous disposez de l'ensemble des paramètres si vous souhaitez effectuer des compilations distinctes et améliorer ainsi les performances du modèle.

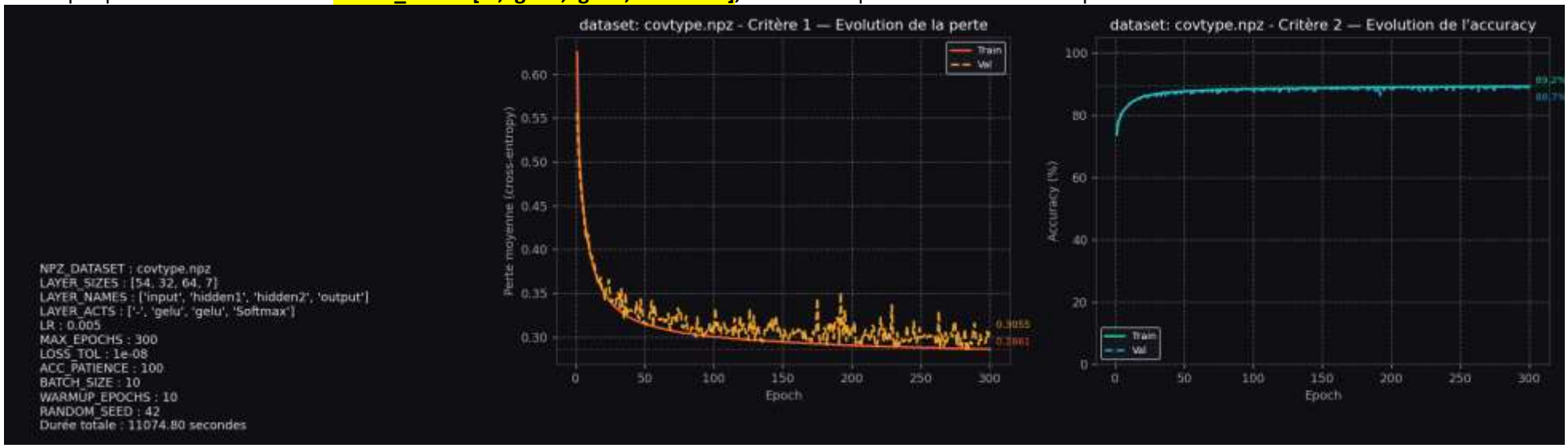
Prennons le même dataset, en sachant qu'il est désiquilibré en termes de répartition des 7 classes à prédire, et afin d'éviter l'écrasement de certains points ayant une valeur d'activation **ReLU** nulle, nous allons la remplacer par la fonction moins tranchante **leaky_relu (avec alpha=0.01)** en remplaçant tout simplement dans le fichier **params_clf.json** la ligne suivante indiquant les activations par couche surlignée en jaune :

```
LAYER_ACTS : ['-', 'leaky_relu', 'leaky_relu', 'Softmax']
```

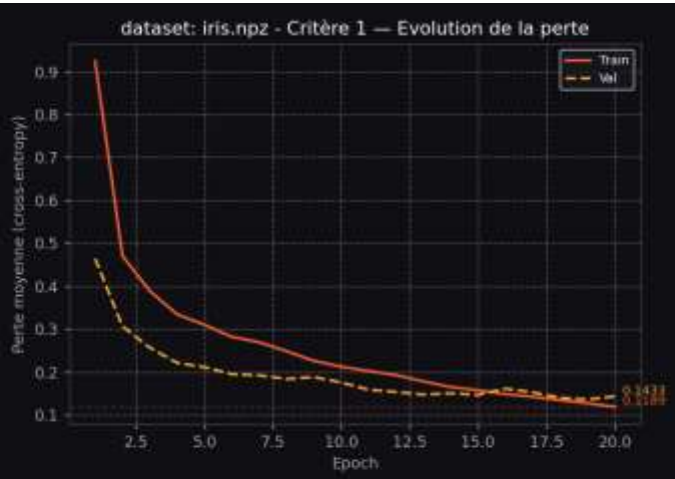
Puis on relance **mpl_clamlp_classif_model.py** pour obtenir des résultats plus prometeurs que les précédents :

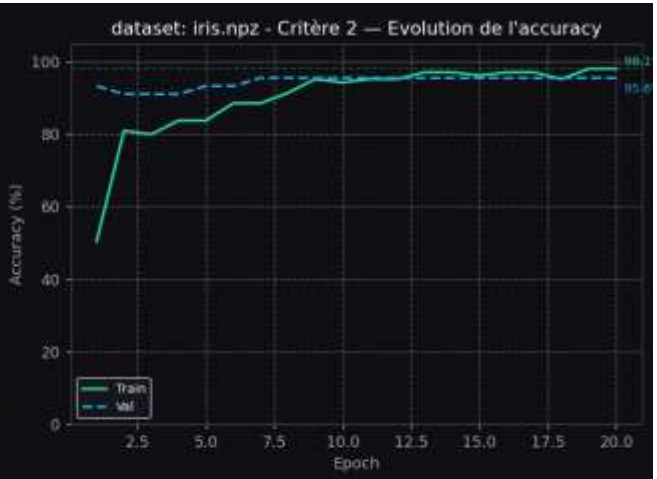


Si on opte pour l'activation GELU : **LAYER_ACTS : ['-', 'gelu', 'gelu', 'Softmax'],** on obtient quasimment les mêmes performances :

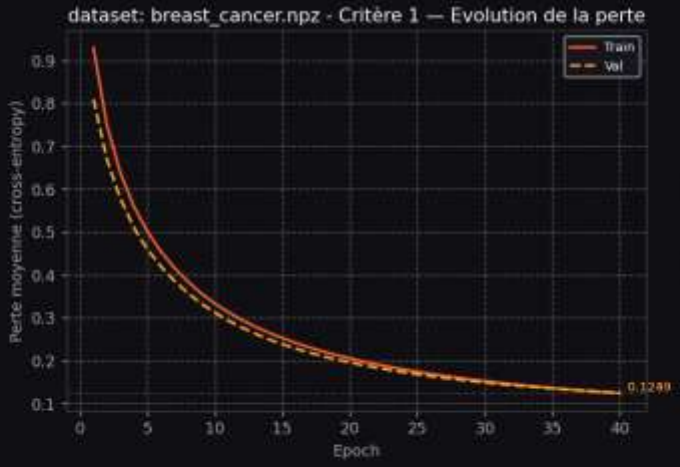


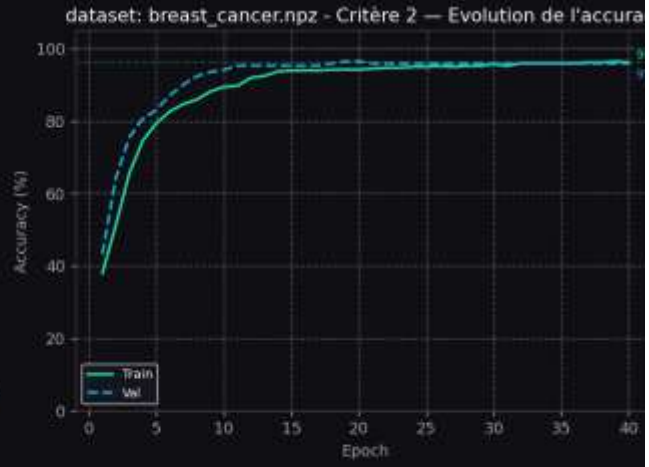
NPZ_DATASET : iris.npz
LAYER_SIZES : [4, 16, 32, 3]
LAYER_NAMES : ['input', 'hidden1', 'hidden2', 'output']
LAYER_ACTS : ['-', 'ReLU', 'ReLU', 'Softmax']
LR : 0.05
MAX_EPOCHS : 100
LOSS_TOL : 1e-08
ACC_PATIENCE : 10
BATCH_SIZE : 10
WARMUP_EPOCHS : 10
RANDOM_SEED : 42
Durée totale : 0.11 secondes



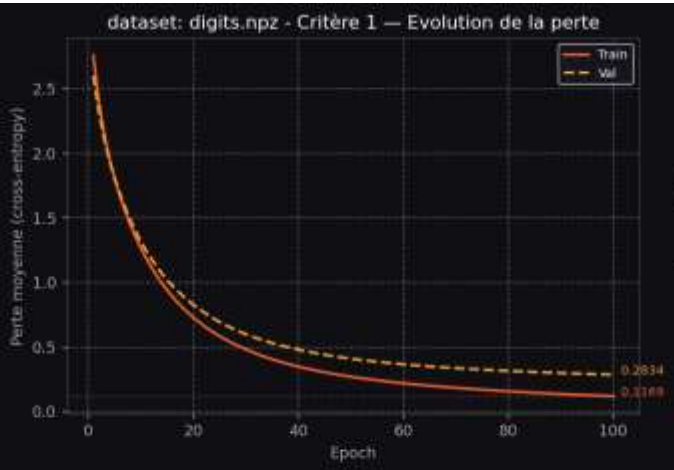


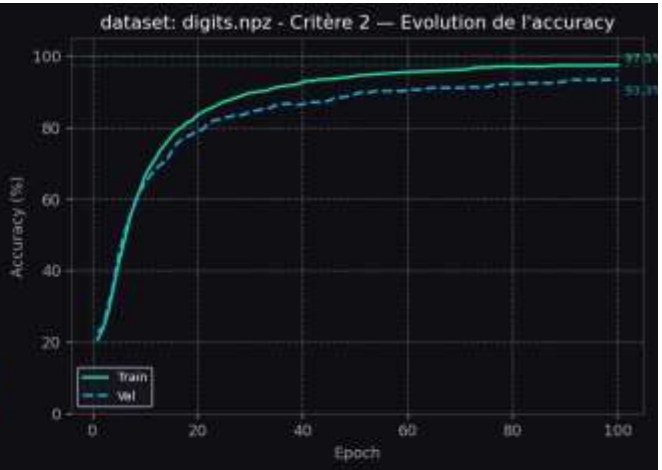
NPZ_DATASET : breast_cancer.npz
LAYER_SIZES : [30, 16, 32, 2]
LAYER_NAMES : ['input', 'hidden1', 'hidden2', 'output']
LAYER_ACTS : ['-', 'ReLU', 'ReLU', 'Softmax']
LR : 0.001
MAX_EPOCHS : 100
LOSS_TOL : 1e-08
ACC_PATIENCE : 20
BATCH_SIZE : 5
WARMUP_EPOCHS : 20
RANDOM_SEED : 42
Durée totale : 0.75 secondes



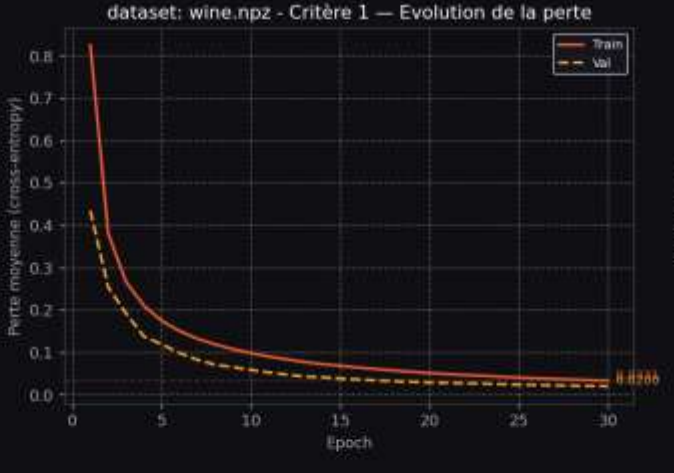


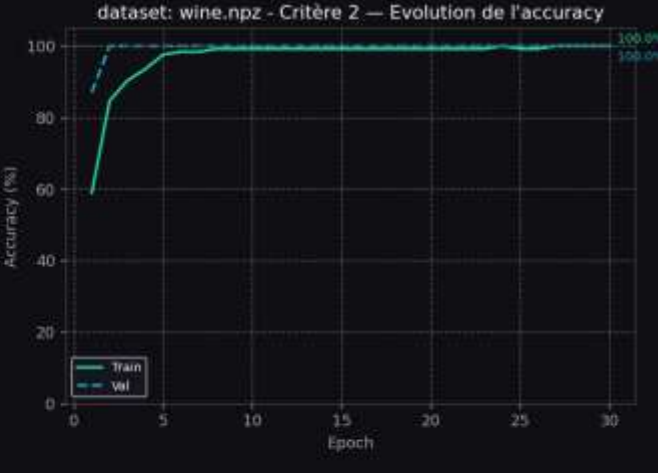
NPZ_DATASET : digits.npz
LAYER_SIZES : [64, 32, 64, 10]
LAYER_NAMES : ['input', 'hidden1', 'hidden2', 'output']
LAYER_ACTS : ['-', 'ReLU', 'ReLU', 'Softmax']
LR : 0.001
MAX_EPOCHS : 100
LOSS_TOL : 1e-08
ACC_PATIENCE : 10
BATCH_SIZE : 10
WARMUP_EPOCHS : 20
RANDOM_SEED : 42
Durée totale : 5.70 secondes





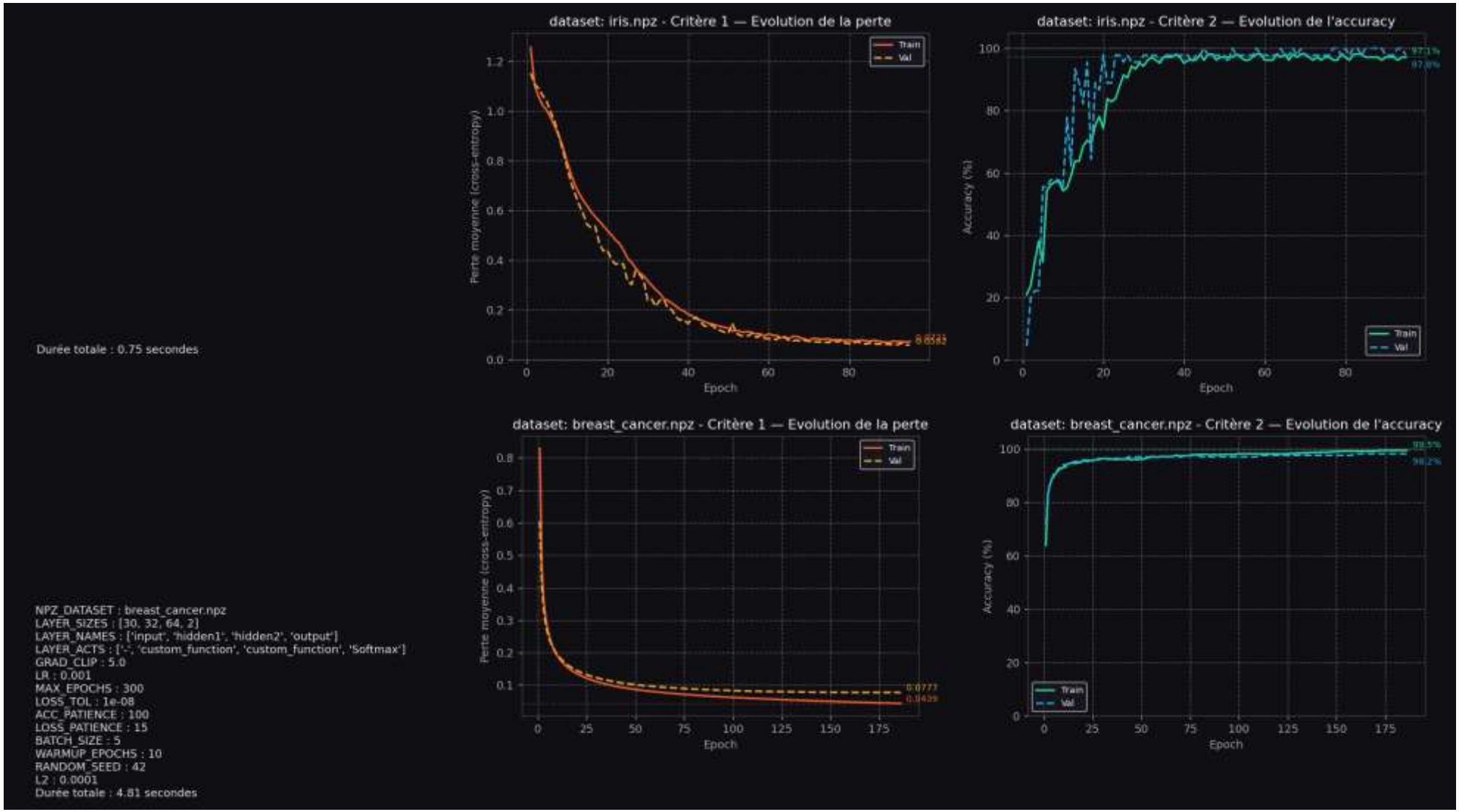
NPZ_DATASET : wine.npz
LAYER_SIZES : [13, 32, 64, 3]
LAYER_NAMES : ['input', 'hidden1', 'hidden2', 'output']
LAYER_ACTS : ['-', 'ReLU', 'ReLU', 'Softmax']
LR : 0.01
MAX_EPOCHS : 100
LOSS_TOL : 1e-08
ACC_PATIENCE : 10
BATCH_SIZE : 5
WARMUP_EPOCHS : 20
RANDOM_SEED : 42
Durée totale : 0.20 secondes



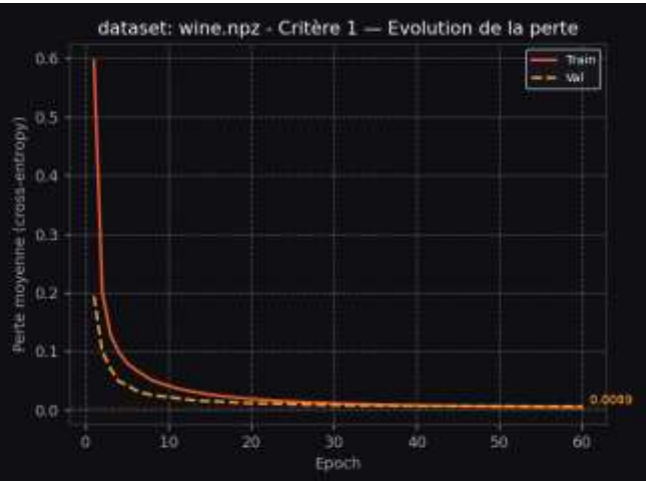


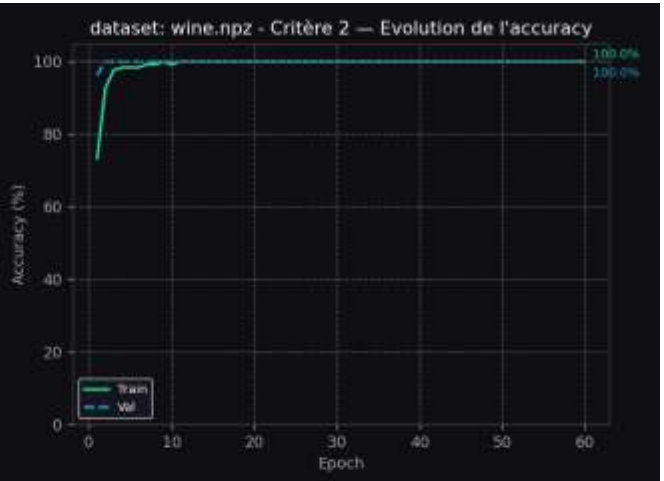
IV. Résultats de l'application de la fonction d'activation custom

Les résultats obtenus ci-dessous sont légèrement meilleurs que ceux obtenus en appliquant les fonctions d'activation standards les plus utilisées :

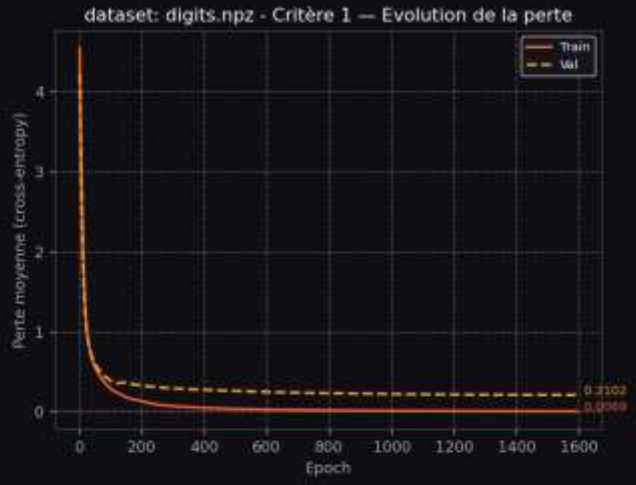


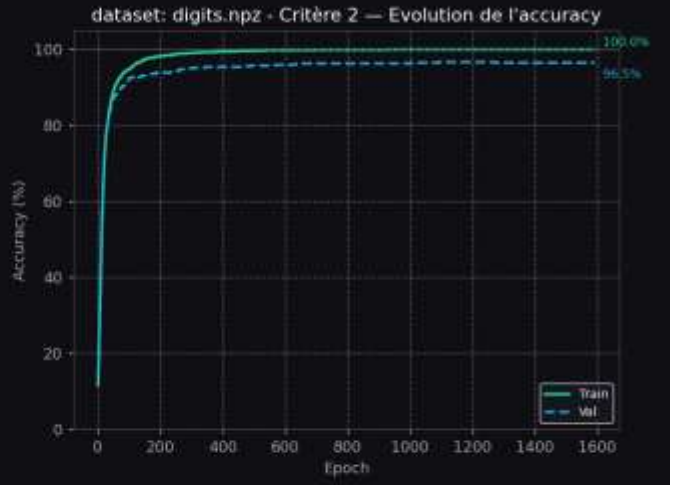
NPZ_DATASET : wine.npz
LAYER_SIZES : [13, 32, 64, 3]
LAYER_NAMES : ['input', 'hidden1', 'hidden2', 'output']
LAYER_ACTS : ['-', 'custom_function', 'custom_function', 'Softmax']
GRAD_CLIP : 5.0
LR : 0.01
MAX_EPOCHS : 200
LOSS_TOL : 1e-08
ACC_PATIENCE : 50
BATCH_SIZE : 2
WARMUP_EPOCHS : 10
RANDOM_SEED : 42
LOSS_PATIENCE : 20
L2 : 0.01
Durée totale : 0.61 secondes

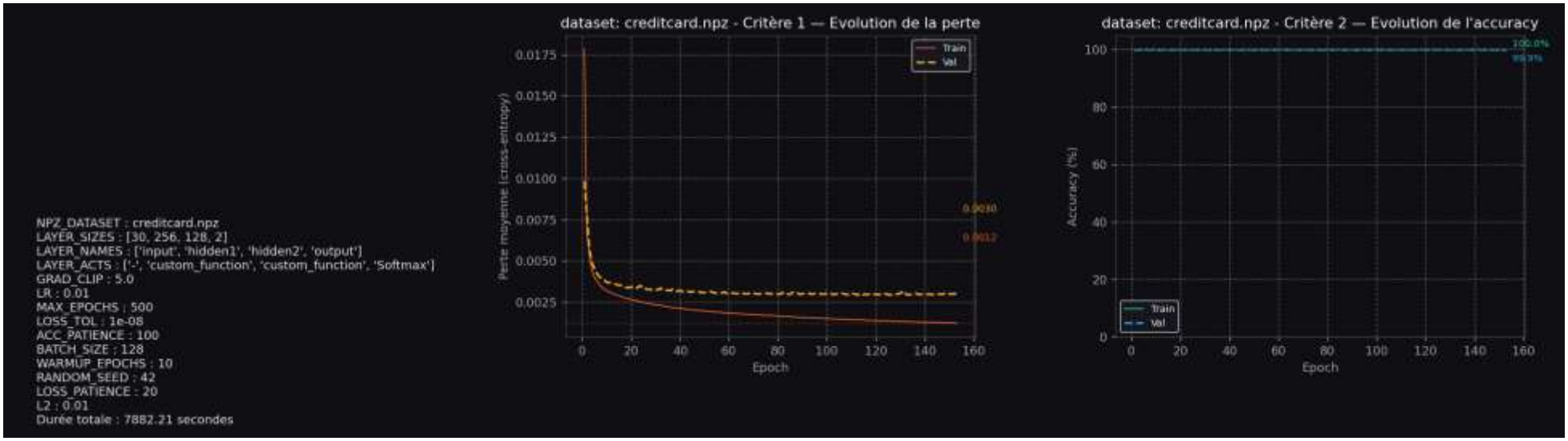




NPZ_DATASET : digits.npz
LAYER_SIZES : [64, 512, 256, 10]
LAYER_NAMES : ['input', 'hidden1', 'hidden2', 'output']
LAYER_ACTS : ['-', 'custom_function', 'custom_function', 'Softmax']
GRAD_CLIP : 5.0
LR : 0.0001
MAX_EPOCHS : 2000
LOSS_TOL : 1e-08
ACC_PATIENCE : 500
BATCH_SIZE : 5
WARMUP_EPOCHS : 20
RANDOM_SEED : 42
Durée totale : 1130.93 secondes







Epoch	Perte(tr)	Δperte(tr)	Acc(tr)	Perte(val)	Δperte(v)	Acc(val)	Rec(val)	F1(val)	Pat.
1	0.017839	inf	99.8%	0.009812	inf	99.9%	79.9%	85.2%	0
2	0.006633	1.12e-02	99.9%	0.007757	2.06e-03	99.9%	77.2%	83.2%	0
3	0.005381	1.25e-03	99.9%	0.005758	2.00e-03	99.9%	84.9%	88.1%	0
4	0.004482	8.98e-04	99.9%	0.004918	8.39e-04	99.9%	84.9%	89.2%	0
5	0.004039	4.44e-04	99.9%	0.004566	3.53e-04	99.9%	86.6%	90.1%	0
6	0.003791	2.47e-04	99.9%	0.004263	3.03e-04	99.9%	86.6%	90.2%	0
7	0.003595	1.96e-04	99.9%	0.004048	2.15e-04	99.9%	86.6%	90.5%	0
…									
70	0.001760	8.44e-06	100.0%	0.003022	2.91e-05	99.9%	88.6%	92.0%	13
80	0.001665	1.12e-05	100.0%	0.002988	1.84e-05	100.0%	87.9%	92.2%	23
90	0.001576	3.45e-06	100.0%	0.002997	2.46e-05	100.0%	87.6%	92.3%	33
100	0.001518	1.60e-05	100.0%	0.003019	3.10e-05	99.9%	87.9%	91.8%	43
110	0.001429	2.23e-05	100.0%	0.003048	1.04e-04	100.0%	87.2%	92.0%	53
120	0.001370	1.59e-05	100.0%	0.002976	6.79e-07	99.9%	88.2%	91.9%	63
130	0.001334	5.30e-06	100.0%	0.003013	1.29e-05	99.9%	87.2%	91.9%	73
140	0.001269	7.44e-06	100.0%	0.002957	1.87e-05	99.9%	87.9%	91.7%	83
150	0.001233	6.37e-06	100.0%	0.002991	7.69e-05	99.9%	87.9%	91.5%	93

ARRÊT à l'epoch 153 : Critère 3 – perte val en hausse depuis 20 epochs consécutifs

Perte finale (train) : 0.001213 Acc : 100.0% Rec : 93.7% F1 : 96.5% (199319/199364)

Perte finale (val) : 0.002984 Acc : 99.9% Rec : 87.9% F1 : 92.0% (85400/85443)

Entraînement terminé en 7882.21 secondes

A1.10 Application des Passes sur des datasets réputés pour Régression

Nous allons appliquer le module réservé aux modèles de régression sur des vrais jeux de donnés de référence académique que nous allons charger à partir de la librairie **klearn.datasets**. Pour ce faire :

6. Se référer au repository (les datasets sont déjà extraits) : https://github.com/ayezza/mlp_basique
7. Faire un clone via GIT (en ligne de commande) ou via GITHUB Desktop en mode UI sur votre machine
8. Charger les datasets (format Numpy compréssé **.npz**) en exécutant **generate_dataset_reg.py**:

```
(a)data, dataset_name = load_diabetes(),        "diabetes"
(b)data, dataset_name = load_wine(),          "wine"
(c)data, dataset_name = load_breast_cancer(), "breast_cancer"
(d)data, dataset_name = load_digits(),        "digits"
```

9. Pour tester chacun des datasets :

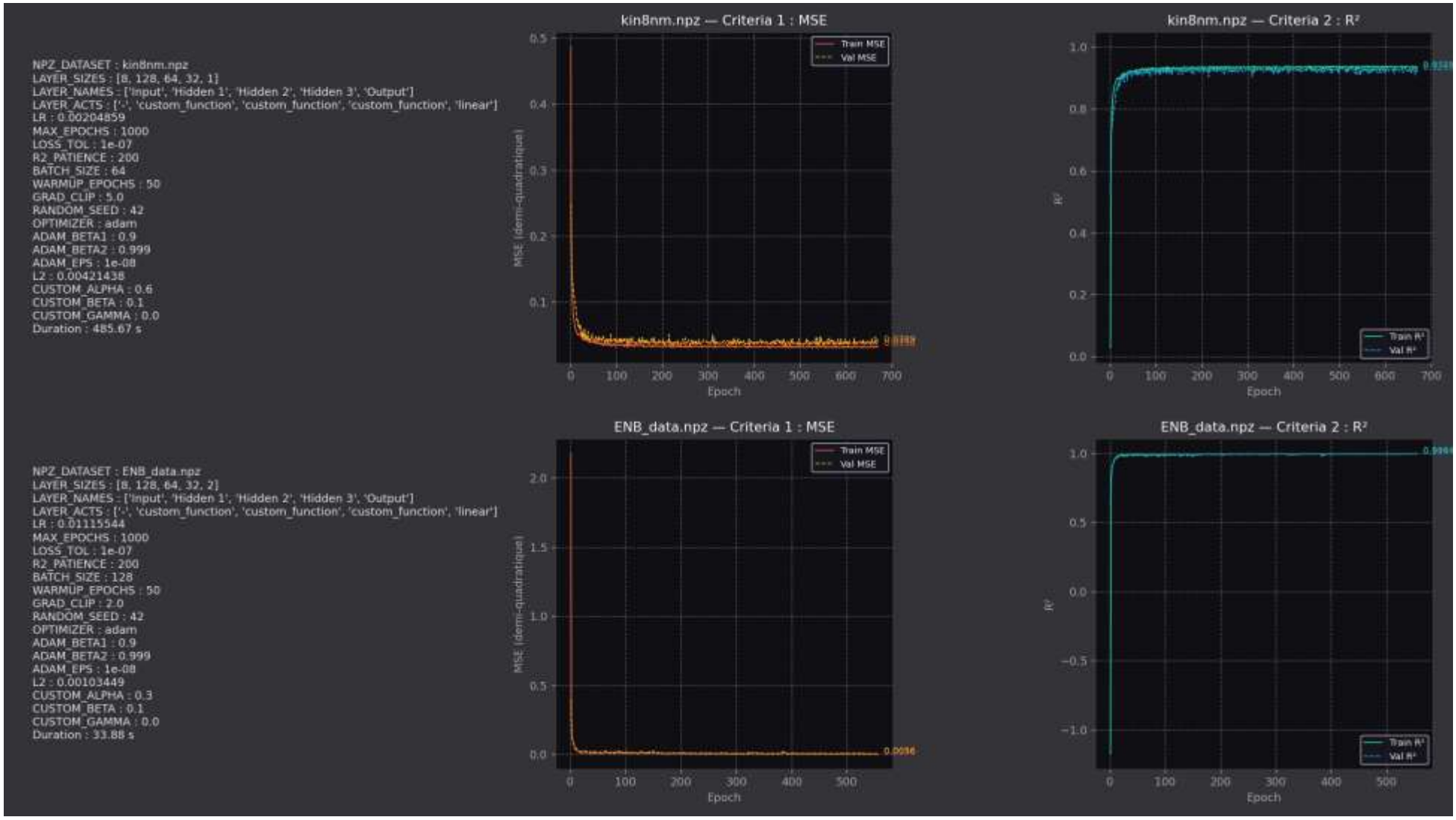
   (a) Compléter le fichier des paramètres JSON **params_reg.json**, ex., pour le fameux dataset **diabetes**:

```
{
    "parameters": {
        "NPZ_DATASET": "diabetes.npz",
        "LAYER_SIZES": [10, 128, 256, 1],
        "LAYER_NAMES": ["Input", "Hidden 1", "Hidden 2", "Output"],
        "LAYER_ACTS":  ["-", "ReLU", "ReLU", "linear"],
        "LR": 0.005,
        "MAX_EPOCHS": 100,
        "LOSS_TOL": 1e-7,
        "R2_PATIENCE": 10,
        "BATCH_SIZE": 5,
        "WARMUP_EPOCHS": 10,
        "GRAD_CLIP": 5.0,
        "RANDOM_SEED": 42,
        "OPTIMIZER": "adam",
        "ADAM_BETA1": 0.9,
        "ADAM_BETA2": 0.999,
        "ADAM_EPS": 1e-8,
        "L2": 0.01
    }
}
```

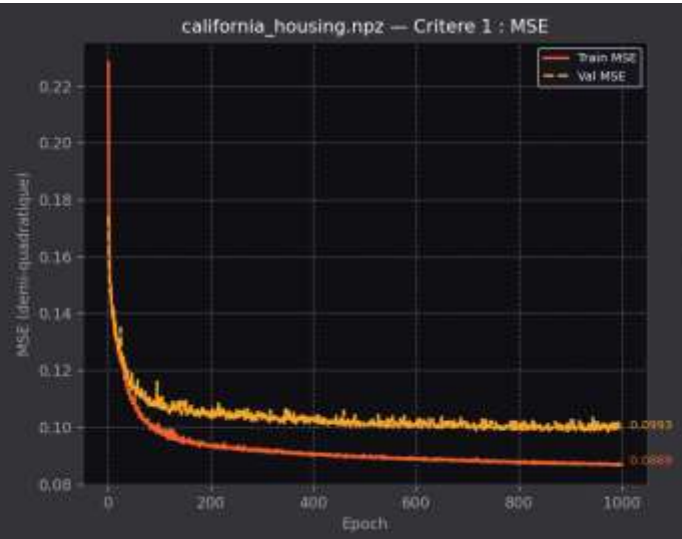
   (b) Lancer le script : **mlp_reg_model.py** qui prendra automatiquement le dataset indiqué ainsi que les autres paramètres associés **(à adapter pour chaque dataset !)**

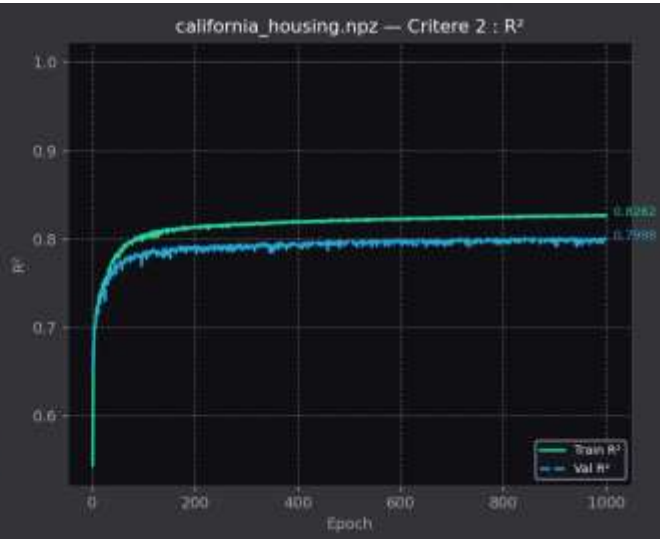
Voici les résultats obtenus (peuvent être différents des vôtres !) matérialisés par l'évolution de la perte (**loss**) et du score **R^2**, **train set** vs **validation set** :

Les performances obtenues sont très honorables avec des durées des traitements (EPOCH/BATCHES) raisonnables, voire négligeables

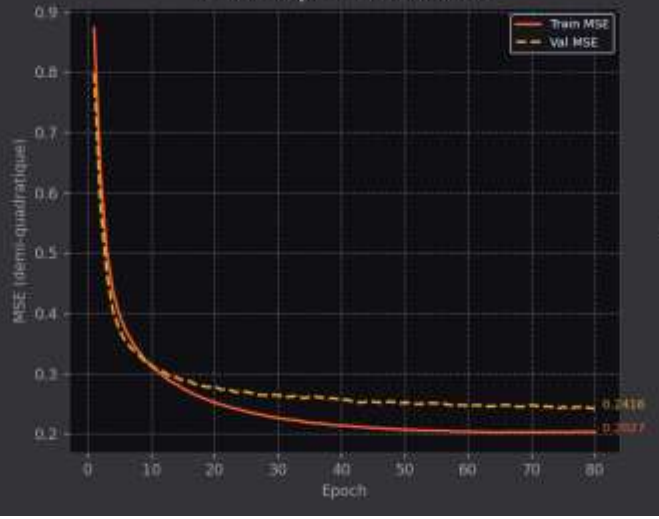


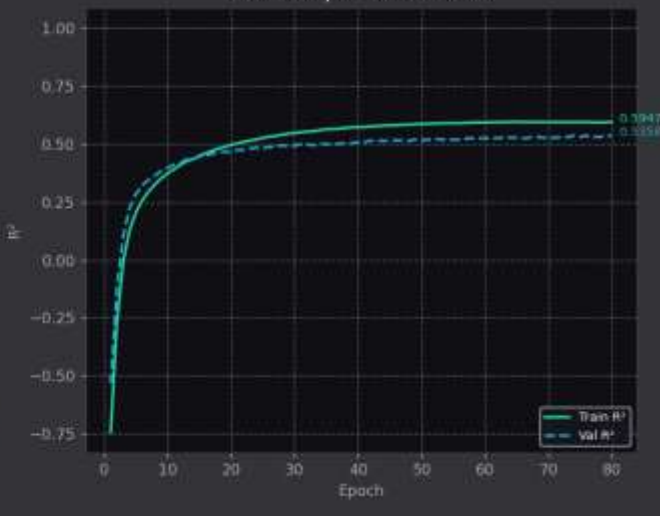
NPZ_DATASET : california_housing.npz
LAYER_SIZES : [8, 256, 128, 1]
LAYER_NAMES : ['Input', 'Hidden 1', 'Hidden 2', 'Output']
LAYER_ACTS : ['-', 'ReLU', 'ReLU', 'linear']
LR : 0.0001
MAX_EPOCHS : 1000
LOSS_TOL : 1e-07
R2_PATIENCE : 300
BATCH_SIZE : 20
WARMUP_EPOCHS : 10
GRAD_CLIP : 5.0
RANDOM_SEED : 42
OPTIMIZER : adam
ADAM_BETA1 : 0.9
ADAM_BETA2 : 0.999
ADAM_EPS : 1e-08
L2 : 0.001
Duree : 1043.77 s



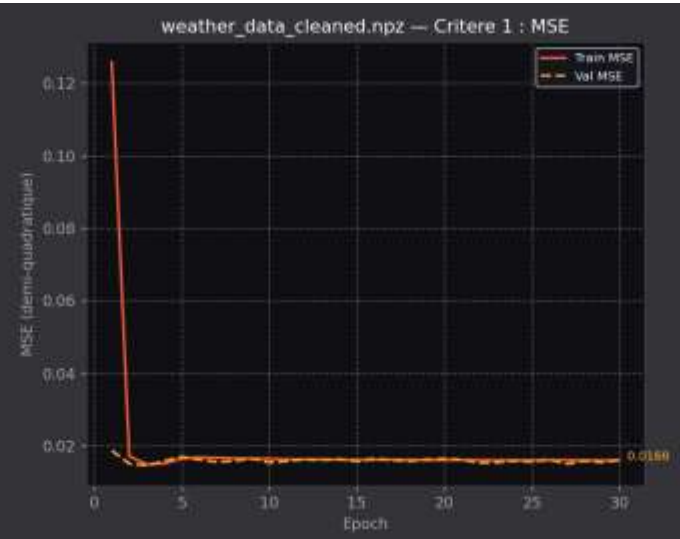


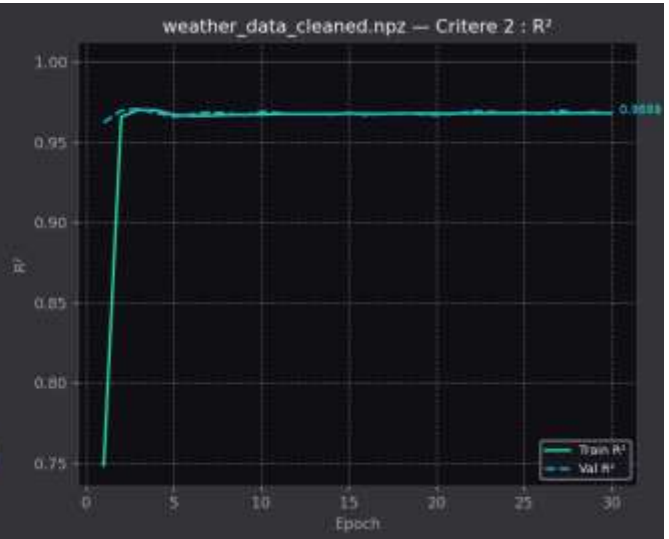
NPZ_DATASET : diabetes.npz
LAYER_SIZES : [10, 128, 64, 1]
LAYER_NAMES : ['Input', 'Hidden 1', 'Hidden 2', 'Output']
LAYER_ACTS : ['-', 'ReLU', 'ReLU', 'linear']
LR : 5e-05
MAX_EPOCHS : 80
LOSS_TOL : 1e-07
R2_PATIENCE : 20
BATCH_SIZE : 5
WARMUP_EPOCHS : 10
GRAD_CLIP : 5.0
RANDOM_SEED : 42
OPTIMIZER : adam
ADAM_BETA1 : 0.9
ADAM_BETA2 : 0.999
ADAM_EPS : 1e-08
L2 : 0.1
Duree : 1.74 s



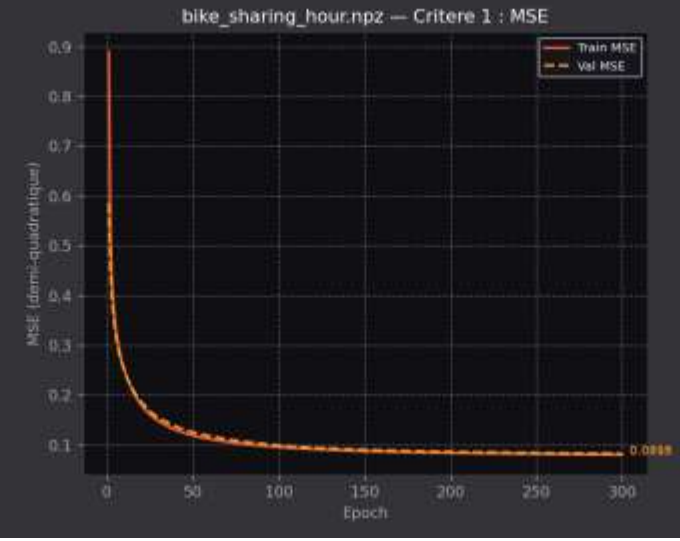


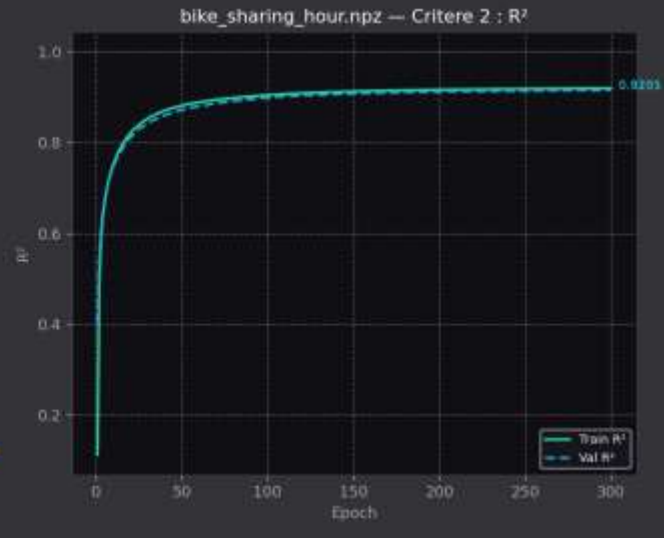
NPZ_DATASET : weather_data_cleaned.npz
LAYER_SIZES : [8, 32, 16, 1]
LAYER_NAMES : ['Input', 'Hidden 1', 'Hidden 2', 'Output']
LAYER_ACTS : ['-', 'ReLU', 'ReLU', 'linear']
LR : 0.0001
MAX_EPOCHS : 500
LOSS_TOL : 1e-07
R2_PATIENCE : 20
BATCH_SIZE : 20
WARMUP_EPOCHS : 10
GRAD_CLIP : 5.0
RANDOM_SEED : 42
OPTIMIZER : adam
ADAM_BETA1 : 0.9
ADAM_BETA2 : 0.999
ADAM_EPS : 1e-08
L2 : 0.1
Duree : 98.25 s

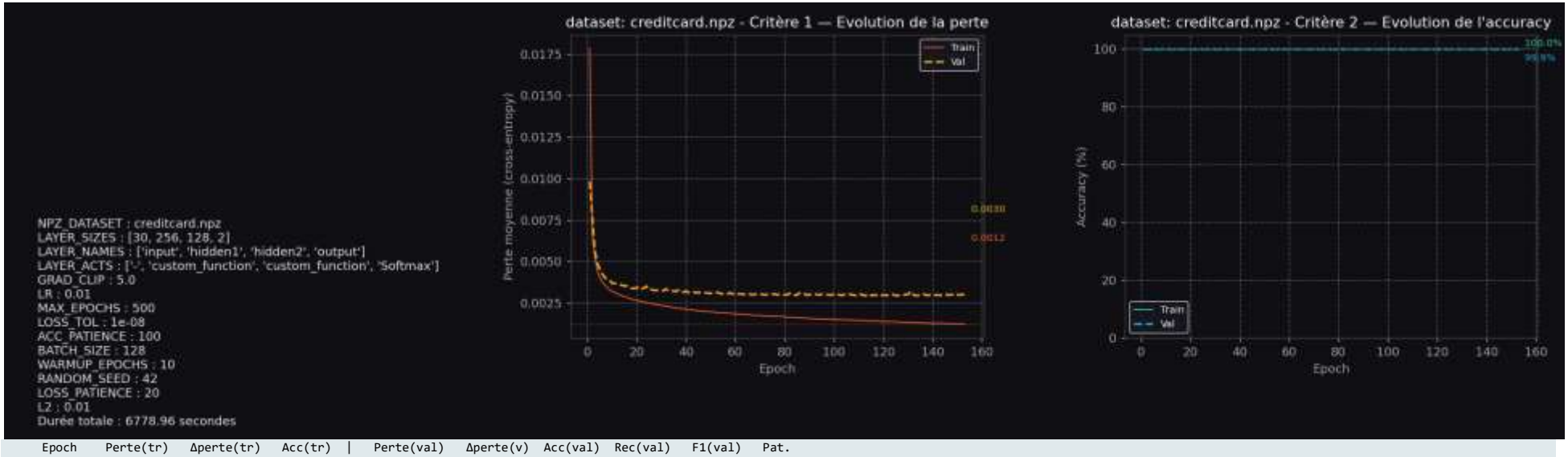




NPZ_DATASET : bike_sharing_hour.npz
LAYER_SIZES : [17, 256, 128, 2]
LAYER_NAMES : ['Input', 'Hidden 1', 'Hidden 2', 'Output']
LAYER_ACTS : ['-', 'ReLU', 'ReLU', 'linear']
LR : 5e-05
MAX_EPOCHS : 300
LOSS_TOL : 1e-07
R2_PATIENCE : 50
BATCH_SIZE : 64
WARMUP_EPOCHS : 10
GRAD_CLIP : 5.0
RANDOM_SEED : 42
OPTIMIZER : adam
ADAM_BETA1 : 0.9
ADAM_BETA2 : 0.999
ADAM_EPS : 1e-08
L2 : 0.005
Duree : 334.30 s







Epoch	Perte(tr)	Δperte(tr)	Acc(tr)	Perte(val)	Δperte(v)	Acc(val)	Rec(val)	F1(val)	Pat.
1	0.017839	inf	99.8%	0.009812	inf	99.9%	79.9%	85.2%	0
2	0.006633	1.12e-02	99.9%	0.007757	2.06e-03	99.9%	77.2%	83.2%	0
3	0.005381	1.25e-03	99.9%	0.005758	2.00e-03	99.9%	84.9%	88.1%	0
4	0.004482	8.98e-04	99.9%	0.004918	8.39e-04	99.9%	84.9%	89.2%	0
5	0.004039	4.44e-04	99.9%	0.004566	3.53e-04	99.9%	86.6%	90.1%	0
6	0.003791	2.47e-04	99.9%	0.004263	3.03e-04	99.9%	86.6%	90.2%	0
7	0.003595	1.96e-04	99.9%	0.004048	2.15e-04	99.9%	86.6%	90.5%	0
8	0.003432	1.64e-04	99.9%	0.003922	1.26e-04	99.9%	86.6%	90.5%	0
9	0.003304	1.28e-04	99.9%	0.003886	3.64e-05	99.9%	86.6%	90.4%	0
10	0.003181	1.23e-04	100.0%	0.003692	1.94e-04	99.9%	86.6%	90.7%	0
11	0.003148	3.31e-05	100.0%	0.003677	1.49e-05	99.9%	86.9%	90.7%	0
12	0.003061	8.67e-05	100.0%	0.003666	1.09e-05	99.9%	86.9%	90.4%	1
13	0.002987	7.41e-05	100.0%	0.003630	3.65e-05	99.9%	86.6%	91.1%	2
14	0.002942	4.50e-05	100.0%	0.003567	6.26e-05	99.9%	87.2%	90.8%	0
15	0.002875	6.74e-05	100.0%	0.003521	4.61e-05	99.9%	87.2%	90.8%	1
16	0.002847	2.81e-05	100.0%	0.003528	6.35e-06	99.9%	87.2%	90.6%	2
17	0.002784	6.26e-05	100.0%	0.003452	7.52e-05	99.9%	87.2%	90.8%	3
18	0.002730	5.45e-05	100.0%	0.003376	7.59e-05	99.9%	87.2%	91.4%	4
19	0.002695	3.45e-05	100.0%	0.003379	2.68e-06	99.9%	86.2%	91.1%	0
20	0.002659	3.66e-05	100.0%	0.003407	2.78e-05	99.9%	85.2%	90.7%	1
30	0.002341	2.52e-05	100.0%	0.003289	2.45e-05	99.9%	87.2%	90.6%	11
40	0.002137	9.53e-06	100.0%	0.003198	3.77e-05	99.9%	85.9%	91.1%	0
50	0.001954	3.00e-05	100.0%	0.003084	9.40e-06	99.9%	88.2%	91.9%	7
60	0.001852	4.16e-07	100.0%	0.003101	7.35e-05	99.9%	88.6%	91.8%	3
70	0.001760	8.44e-06	100.0%	0.003022	2.91e-05	99.9%	88.6%	92.0%	13
80	0.001665	1.12e-05	100.0%	0.002988	1.84e-05	100.0%	87.9%	92.2%	23
90	0.001576	3.45e-06	100.0%	0.002997	2.46e-05	100.0%	87.6%	92.3%	33
100	0.001518	1.60e-05	100.0%	0.003019	3.10e-05	99.9%	87.9%	91.8%	43
110	0.001429	2.23e-05	100.0%	0.003048	1.04e-04	100.0%	87.2%	92.0%	53
120	0.001370	1.59e-05	100.0%	0.002976	6.79e-07	99.9%	88.2%	91.9%	63
130	0.001334	5.30e-06	100.0%	0.003013	1.29e-05	99.9%	87.2%	91.9%	73
140	0.001269	7.44e-06	100.0%	0.002957	1.87e-05	99.9%	87.9%	91.7%	83
150	0.001233	6.37e-06	100.0%	0.002991	7.69e-05	99.9%	87.9%	91.5%	93

```
======================================================================
ARRÊT à l'epoch 153 : Critère 3 - perte val en hausse depuis 20 epochs consécutifs
Perte finale    (train) : 0.001213   Acc : 100.0%   Rec : 93.7%   F1 : 96.5%   (199319/199364)
Perte finale    (val)   : 0.002984   Acc : 99.9%   Rec : 87.9%   F1 : 92.0%   (85400/85443)
======================================================================
Entrainement terminé en 6778.96 secondes
```

B.1 1D-CNN

B1.1 Quest-ce qu'un 1D-CNN

Le design d'un réseau de type **1D-CNN (Convolutional Neurone Network 1 Dimension)** est fait pour traiter des données séquentielles comme les **signaux**, les **séquences temporelles** ou du **texte**, données pour lesquelles l'ordre des éléments a du sens, et est important. Contrairement au réseau **MLP** pour lequel le input est analysé dans sa globalité sans prêter attention au séquencement (ordre des éléments), le **1D-CNN** examine morceau par morceau des parties sur la base d'un noyau (**Kernel**) dans l'objectif de déterminer des paternes locaux à chacune des positions. Ce principe de fonctionnement d'un **1D-CNN** est illustré dans les deux schémas suivants, le 1er statique et le 2ème présentant une animation du 1er.

B1.2 Fonctions propres à 1D-CNN

Ci-après des fonctions (transformations) standard que nous allons utiliser pour passer d'une couche à la suivante d'un ensemble de neurones :

Conv1D :

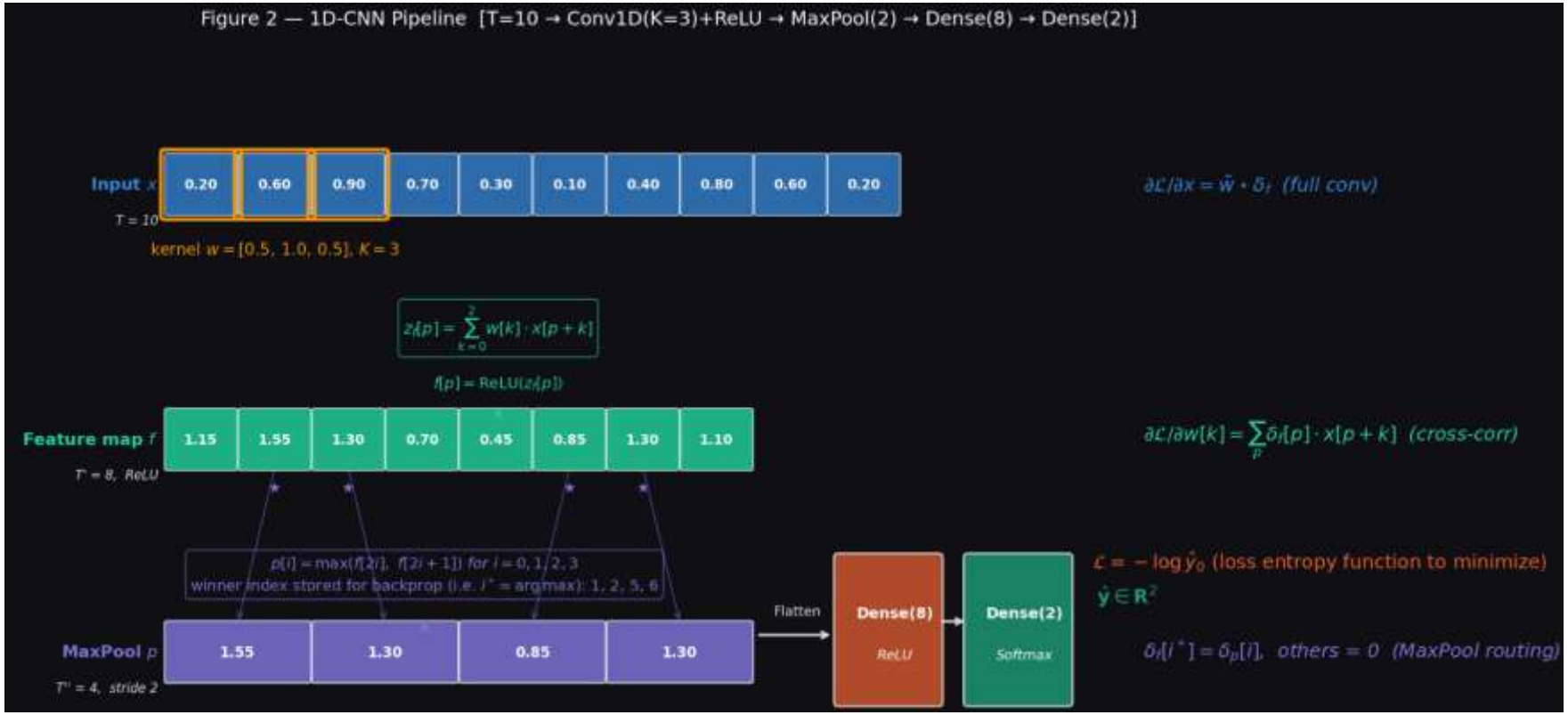
Signal d'entrée: $x \in R^{C\_in}$

Filtre (ou Kernel de dimension K) :

$$y_{c_{out},t} = \sum_{j=0}^{K-1} W_{c_{out},c_{in,j}} . x_{c_{in},ts+j-p}$$

B1.3 Le schéma d'un modèle 1D-CNN simple

Architecture 1D-CNN simple (Classification)



$L = 3$ (4 *couches en débutant* à 0) $- l = 0,1,2,3$

Cliquer pour voir l'animation avec l'ensemble des transformations et opérations :

B1.3 Architecture (1D-CNN classification) en résumé

Pipeline : $\boldsymbol{Input}\ (T = 10) \rightarrow \mathbf{Conv1D}(K = 3) + \mathrm{ReLU} \rightarrow \mathbf{MaxPool}(2) \rightarrow \mathbf{Dense}(8) \rightarrow \mathbf{Dense}(2) \rightarrow \boldsymbol{Output}$

B1.4 ➔ PASSE AVANT

PASSE AVANT

Couche	Dimension	Transformations
Input	10	▪ C'est une séquence numérique: [0.20, 0.60, 0.90, 0.70, ...] ▪ Comme le Kernel qui sera **appliqué via un produit scalaire** pour passer à la couche suivante, est de dimension 3 : ($[\mathbf{0.5}, \mathbf{1.0}, \mathbf{0.5}]$), on définit des parties séquentielles de l'input de même dimension chacune.
Conv1D (Mapping des features) + ReLU	8	▪ Cette couche est le résultat de l'opération à fenêtre glissante de dim 3 chacune, celle du Kernel $K = [\mathbf{0.5}, \mathbf{1.0}, \mathbf{0.5}]$ ▪ Pour chaque position nous calculons un seul nombre : le **produit scalaire** du noyau [0.5, 1.0, 0.5] par les 3 valeurs de la 1ère séquence de l'input [0.20,0.60,0.90], ex. 1er élément du mapping = $\begin{pmatrix}.5\\1.0\\.5\end{pmatrix}.\begin{pmatrix}.20\\.60\\.90\end{pmatrix} = \mathbf{0.5 \times 0.20 + 1.0 \times 0.60 + 0.5 \times 0.90 = 1.15}$ représentant le 1er élément du mapping des features… $$Pour\ p = 0,1,\dots,7: Z_f[p] = \sum_{k=0}^{K-1} w[k] \cdot x[p+k]$$ ▪ Comme $dim(Input) = 10, dim(Kernel) = 3$, alors la dimension de la couche de mapping $= 10 - 3 + 1 = 8$ ▪ Puis nous appliquons la fonction d'activation ***ReLU*** afin d'obtenir les valeurs du vecteur de la prochaine couche : $$Pour\ p = 0,1,\dots,7: f[p] = ReLU(Z_f[p]),$$ Comme toutes les valeurs sont > 0, alors l'activations ReLU ne change pas les valeurs obtenues au travers l'application du Kernel.
Max Pooling 1D	4	$$\boldsymbol{MaxPool(Pool\_size = 2, stride = 2)}$$ Comme son nom l'indique, il s'agit de retenir les activations les plus fortes pour chaque paire (2) des activations ReLU obtenues en parcourant séquentiellement d'un pas de 2 le vecteur obtenu : $$Pour\ p = 0,2,4,6: \boldsymbol{MaxPool}[p] = max\left(ReLU(Z_f[p]), ReLU(Z_f[p+1])\right)$$ *ou d'une manière générale* : $$\boldsymbol{MaxPool}[p] = max(f[2i], f[2i+1]), i = 0,\dots,N-1$$ $$(N\ est\ le\ \#\ de\ neurones, \mathrm{f}\ fonction\ d'activation)$$ Chaque élément retenu (le gagnant) est marqué par ⋆ dans le schéma ci-dessus et sera sauvegardé pour être utilisé lors de la phase de RETROPROPAGATION.
Dense (16) + ReLU	16	$$\boldsymbol{Dense(16, ReLU)}$$ Rappelons que **Dense (appelée Full Connected, car elle connecte tous les neurones de la couche précédente à chaque neurone de la couche suivante)** est tout simplement une transformation linéaire affine : $z = Wx + b$, avec • $x \in \mathbb{R}^{(n_{\text{in}})}$: vecteur d'entrée • $W \in \mathbb{R}^{(n_{\text{out}} \times n_{\text{in}})}$: matrice des poids

		• $b \in \mathbb{R}^{(n_{out})}$: vecteur des biais • $z \in \mathbb{R}^{(n_{out})}$: sortie linéaire Dans notre cas : $n_{in} = 4, n_{out} = 8$ $\boldsymbol{Dense \rightarrow z_{d1} = W_{D1}p + b_{d1}}$ Activation → **ReLU** → $\boldsymbol{a_{d1} = ReLU(z_{d1})}$
Dense (8) + SotfMax	8	$\boldsymbol{Dense(8, SoftMax)}$ Couche Dense avec : $n_{in} = 8, n_{out} = 2$ Fonction d'activation pour la sortie/entrée vers la fonction de perte : **SoftMax** $\boldsymbol{Dense \rightarrow z_{d2} = W_{D2}p + b_{d2}}$ Activation → $\hat{y} = \text{Softmax}(\mathbf{z}_{d2})$
Output	2	Fonction de perte: $\mathcal{L} = -\log\hat{y}_0$ (loss entropy)

B1.5 ← Passe ARRIERE (RETROPROPAGATION)

PASSE ARRIERE

Couche	Dimension	Transformations
Input	10	▪ C'est une séquence numérique: [0.20, 0.60, 0.90, 0.70, ...] ▪ Comme le Kernel qui sera **appliqué via un produit scalaire** pour passer à la couche suivante, est de dimension 3 : ($[\mathbf{0.5}, \mathbf{1.0}, \mathbf{0.5}]$), on définit des parties séquentielles de l'input de même dimension chacune.
Conv1D (Mapping des features) + ReLU	8	▪ Cette couche est le résultat de l'opération à fenêtre glissante de dim 3 chacune, celle du Kernel $K = [\mathbf{0.5}, \mathbf{1.0}, \mathbf{0.5}]$ ▪ Pour chaque position nous calculons un seul nombre : le **produit scalaire** du noyau [0.5, 1.0, 0.5] par les 3 valeurs de la 1ère séquence de l'input [0.20,0.60,0.90], ex. 1er élément du mapping = $\begin{pmatrix}.5\\1.0\\.5\end{pmatrix}.\begin{pmatrix}.20\\.60\\.90\end{pmatrix} = \mathbf{0.5 \times 0.20 + 1.0 \times 0.60 + 0.5 \times 0.90 = 1.15}$ représentant le 1er élément du mapping des features… $$Pour\ p = 0,1,\dots,7: Z_f[p] = \sum_{k=0}^{K-1} w[k] \cdot x[p+k]$$ ▪ Comme $dim(Input) = 10, dim(Kernel) = 3$, alors la dimension de la couche de mapping $= 10 - 3 + 1 = 8$ ▪ Puis nous appliquons la fonction d'activation ***ReLU*** afin d'obtenir les valeurs du vecteur de la prochaine couche : $$Pour\ p = 0,1,\dots,7: f[p] = ReLU(Z_f[p]),$$ Comme toutes les valeurs sont > 0, alors l'activations ReLU ne change pas les valeurs obtenues au travers l'application du Kernel.

Dense layers (standard MLP backprop — see §2.4):

$$\boldsymbol{\delta}_{d2} = \hat{\mathbf{y}} - \mathbf{y} \frac{\partial \mathcal{L}}{\partial W_{D2}} = \boldsymbol{\delta}_{d2}\, \mathbf{a}_{d1}^{T}$$

$$\boldsymbol{\delta}_{d1} = W_{D2}^{T}\, \boldsymbol{\delta}_{d2} \odot \mathrm{ReLU}'(\mathbf{z}_{d1}) \frac{\partial \mathcal{L}}{\partial W_{D1}} = \boldsymbol{\delta}_{d1}\, \mathbf{p}^{T}$$

Gradient arriving at pool layer:

$$\boldsymbol{\delta}_{\text{pool}} = W_{D1}^{T}\, \boldsymbol{\delta}_{d1} \in \mathbb{R}^{T''}$$

MaxPool backward (gradient routing):

$$\delta_{\text{feat}}[t] = \begin{cases} \delta_{\text{pool}}[i] & \text{if } t = i^{*} \text{ (winner index for pool cell } i) \\ 0 & \text{otherwise} \end{cases}$$

ReLU gate (applied after routing):

$$\delta_{\text{feat}}[p] \;\times\!= \; \mathbf{1}_{z_f[p]>0}$$

Conv1D — kernel gradient $\partial \mathcal{L} / \partial w$ (cross-correlation):

$$\frac{\partial \mathcal{L}}{\partial w[k]} = \sum_{p=0}^{T'-1} \delta_{\text{feat}}\,[p] \cdot x[p+k], k = 0, \ldots, K-1$$

Conv1D — input gradient $\partial \mathcal{L} / \partial x$ (full convolution with flipped kernel):

$$\frac{\partial \mathcal{L}}{\partial x[t]} = \sum_{k=0}^{K-1} \delta_{\text{feat}}\,[t-k] \cdot w[k] \;\equiv\; (\delta_{\text{feat}} \star \tilde{\mathbf{w}})[t]$$

where $\tilde{w}[k] = w[K-1-k]$ denotes kernel reversal.

Kernel update:

$$w[k] \leftarrow w[k] - \eta \cdot \frac{\partial \mathcal{L}}{\partial w[k]}$$

Étape	Transition des couches $(L = 3)$	Mise à jour des variables
0	**Output** $(l = L)$	Delta de sortie à utiliser dans les étapes suivantes : $\boldsymbol{\delta}_{d2} = \hat{\boldsymbol{y}} - \boldsymbol{y}$

1	**Output → Hidden 2 ($l = L - 1$)**	Pour chaque l, utiliser successivement les formules: a) **Calculer le delta de Kronecker de l'étape l :** $\boldsymbol{\delta}^{(l)} = \left(W^{(l+1)}\right)^T \boldsymbol{\delta}^{(l+1)} \odot ReLU'\left(\mathbf{z}^{(l)}\right)$ b) **Calculer les gradients :** $\frac{\partial \mathcal{L}}{\partial W^{(l)}} = \boldsymbol{\delta}^{(l)} \left(\mathbf{a}^{(l-1)}\right)^T, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(l)}} = \boldsymbol{\delta}^{(l)}$ c) **Mettre à jour les poids et les biais du modèle :** $\mathrm{W}^{(\mathrm{l}+1)} \leftarrow \mathrm{W}^{(\mathrm{l})} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathrm{W}^{(\mathrm{l})}}$ $\mathbf{b}^{(\mathrm{l}+1)} \leftarrow \mathbf{b}^{(\mathrm{l})} - \eta \cdot \frac{\partial \mathcal{L}}{\partial \mathbf{b}^{(\mathrm{l})}}$

C.1 2D-CNN

A venir…

D.1 RNN

A venir…