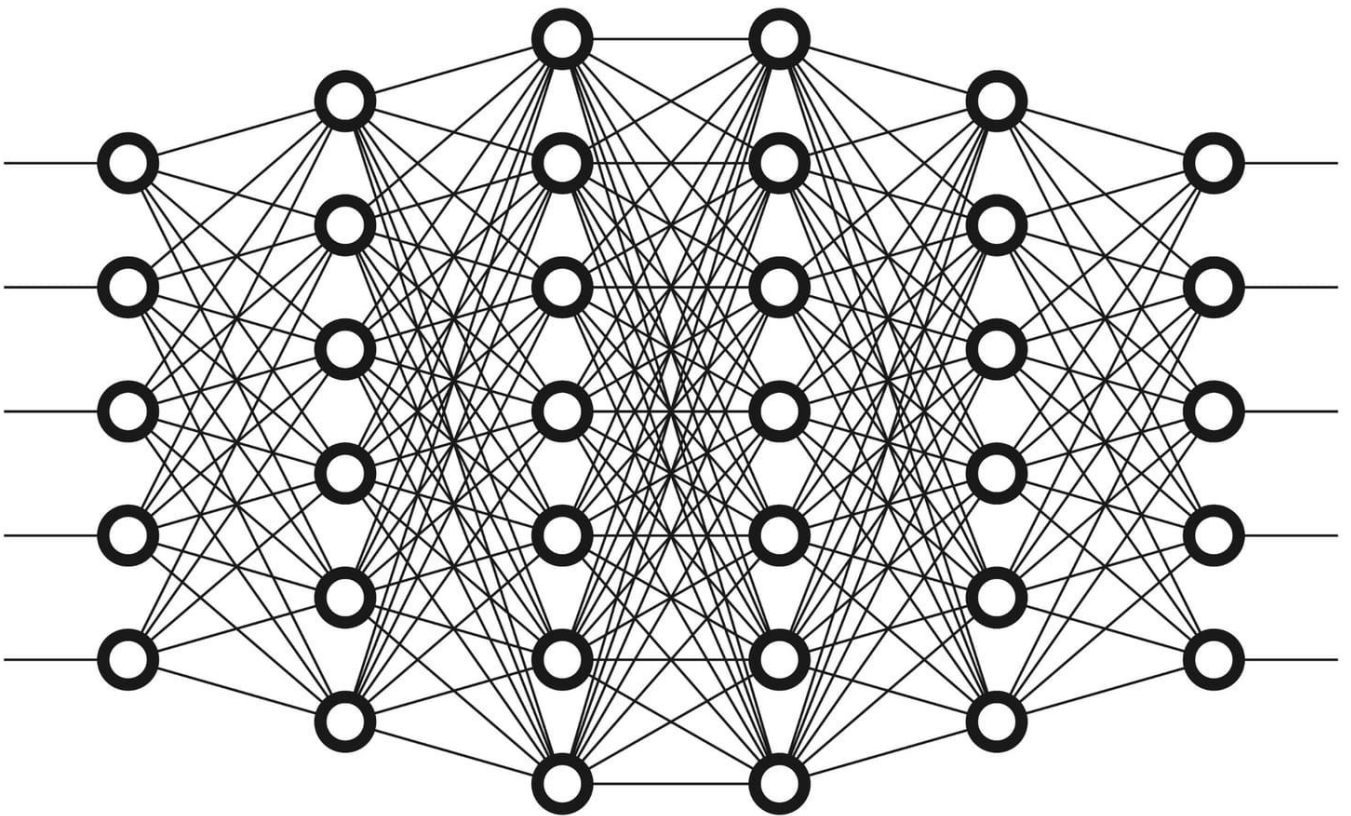


# Machine-Learning : Projet-2

Réseaux De Neurones



AL NATOUR Mazen et HERVOUET Léo

ML 2024-2025

# Table des matières

<b>Introduction :</b>	3
I. Perceptron Simple	4
A. Stratégie de Développement	4
B. Fonctionnement pour le cas du OU	5
C. Figure de la frontière de décision	6
II. Apprentissage Widrow	7
A. Stratégie de Développement	7
B. Ensemble Test 1	8
1. Graphiques des étapes d'apprentissage	8
2. Erreur en fonction des itérations	9
3. Tests avec initialisations différentes	10
C. Ensemble Test 2	11
1. Graphiques des étapes d'apprentissage	11
2. Erreur en fonction des itérations	13
3. Tests avec initialisations différentes	14
III. Perceptron Multicouche	15
A. Stratégie de Développement	15
B. Résultats	16
IV. Apprentissage Multicouche	17
A. Stratégie de Développement	17
B. Erreur en fonction des itérations	17
C. Droites séparatrices	18
V. Classification par Full-Connected	20
A. Stratégie de Développement	20
B. Comparaison des caractéristiques	20
C. Paramétrage optimal	25
D. Évaluation par classe d'images	25
E. Comparaison avec K-Plus-Proches-Voisins (KppV)	25
VI. Classification par Deep Learning	26
A. Architecture et stratégie d'apprentissage :	26
B. Comparaison des caractéristiques du réseau	27
C. Comparaison avec Full-Connected	28
D. Data Augmentation	28
E. Stratégies pour limiter l'overfitting	29
F. Transfert Learning	30
<b>Conclusion :</b>	31

## Introduction :

Dans le cadre de ce projet, nous explorerons plusieurs algorithmes d'apprentissage supervisé, en commençant par des modèles simples tels que les perceptrons et en progressant vers des architectures plus complexes, incluant des réseaux neuronaux multicouches et des méthodes de classification avancées. Chaque étape fera l'objet d'une réflexion approfondie pour élaborer des stratégies adaptées aux spécificités de chaque approche, tout en testant les résultats obtenus dans des cas variés.

L'objectif est de comprendre le comportement et l'efficacité de ces algorithmes dans divers environnements d'apprentissage en analysant l'impact des choix d'initialisation, des itérations et des caractéristiques des données sur la convergence des modèles. Nous chercherons également à optimiser les paramètres pour des performances optimales.

Le rapport suivra une progression logique. Nous étudierons dans un premier temps un perceptron simple en décrivant son fonctionnement et en analysant ses limites de décision. Ensuite, nous étudierons l'apprentissage basé sur Widrow et le testerons sur différents ensembles de données. Enfin, nous examinerons de plus près les réseaux multicouches en explorant leurs performances dans les tâches de classification, en particulier les architectures entièrement connectées et d'apprentissage profond. Une attention particulière sera accordée aux stratégies d'optimisation, aux méthodes permettant de limiter le surapprentissage et aux techniques de transfert de connaissances.

Ce travail mettra en lumière les étapes clés de notre approche, identifiera les défis à relever et évaluera les solutions que nous mettrons en œuvre pour atteindre le double objectif de comprendre l'algorithme et de l'appliquer efficacement à des problèmes concrets.

# I. Perceptron Simple

## A. Stratégie de Développement

Le perceptron simple qui repose sur une règle d'apprentissage très simple. Il consiste à ajuster les poids synaptiques en fonction des erreurs observées lors de la classification des points d'entrée. L'objectif est de trouver une frontière de décision linéaire qui sépare correctement les classes des données.

### ➤ Implémentation

L'implémentation du perceptron simple commence par la définition d'une fonction `perceptron_simple`. Cette fonction prend en entrée :

- **inputs** : un tableau numpy représentant les points de données à classer.
- **weights** : un tableau numpy contenant les poids synaptiques du neurone, incluant le seuil.
- **activation\_function** : un entier qui détermine la fonction d'activation à utiliser (0 pour la fonction signe et 1 pour la fonction tanh).

Le calcul du produit scalaire entre les points d'entrée et les poids est réalisé à l'aide de `np.dot(inputs, weights[1:])`, suivi de l'ajout du seuil (`poids[0]`). Ensuite, en fonction de la fonction d'activation choisie, le résultat est activé avec la fonction signe (`np.sign`) ou tangente hyperbolique (`np.tanh`).

### ➤ Choix de l'initialisation des poids

Dans l'exemple, les poids initiaux sont définis manuellement : **`weights_OR = np.array([-0.5, 1, 1])`**

Ce vecteur de poids correspond à un biais (le seuil, ici -0.5) et à deux poids associés aux entrées `x1x_1x1` et `x2x_2x2`. Ces poids sont choisis de manière à résoudre le problème du OU, comme illustré ci-dessous.

### ➤ Fonction d'activation

Le perceptron utilise deux options pour la fonction d'activation : la fonction **signe** ou la fonction **tanh**. Dans ce cas, la fonction signe est utilisée, car elle est bien adaptée à la classification binaire :

- Signe de la sortie : 1 si le résultat est positif, -1 sinon.

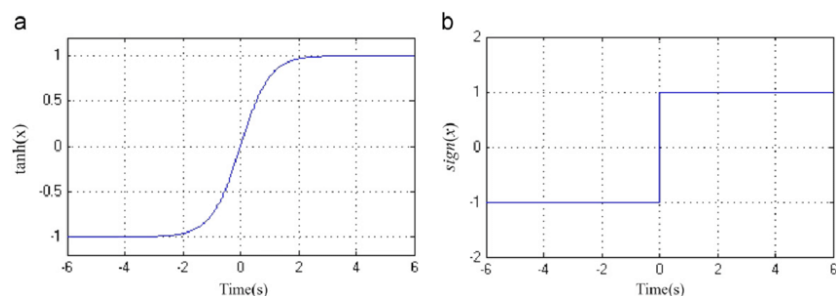


Figure 1 : Différence entre la fonction `tanh()` et `sign()`

### ➤ Critère d'arrêt

Dans cette implémentation, le perceptron ne continue pas l'apprentissage à travers les itérations. Une exécution unique est réalisée, et les poids sont définis de manière fixe. Un critère d'arrêt classique dans un cas d'apprentissage serait basé sur un nombre d'itérations maximal ou sur une convergence en fonction de l'erreur.

## B. Fonctionnement pour le cas du OU

Pour vérifier le fonctionnement du perceptron, nous avons testé l'algorithme sur un ensemble de données correspondant à la fonction logique OU. Les points d'entrée utilisés sont les suivants :

```
the_data_points = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
```

Ce tableau correspond à la table de vérité de la fonction OU, où chaque couple d'entrée est associé à une sortie définie par :

- $0 \text{ OR } 0 = 0$
- $0 \text{ OR } 1 = 1$
- $1 \text{ OR } 0 = 1$
- $1 \text{ OR } 1 = 1$

L'exécution du perceptron sur cet ensemble produit les résultats suivants :

```
results_OR = perceptron_simple(the_data_points, weights_OR, 0)  
print("results_OR : ", results_OR)
```

**Sortie obtenue :** *results\_OR : [-1 1 1 1]*

Ici, les résultats montrent que le perceptron classe correctement les points associés à la sortie du OU :

- Le point  $[0, 0]$  est classé comme -1 (faux).
- Les points  $[0, 1]$ ,  $[1, 0]$ , et  $[1, 1]$  sont classés comme 1 (vrai).

## C. Figure de la frontière de décision

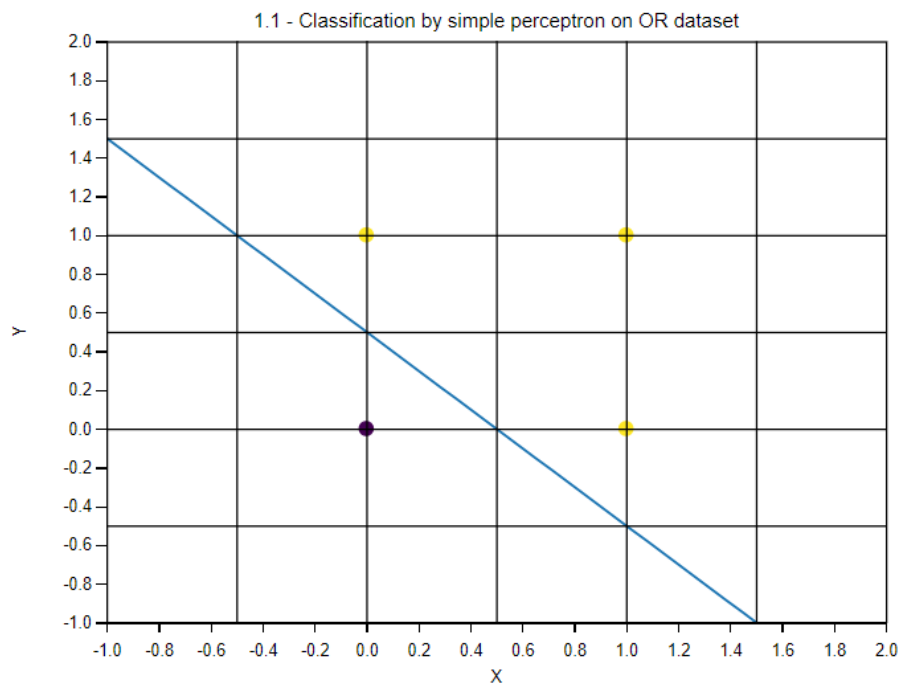


Figure 2 : Classification par perceptron Simple sur l'ensemble de données OR

La figure obtenue (Figure 2) montre les points d'entrée (en violet et jaune) ainsi que la droite de séparation calculée. Voici le code utilisé pour générer le graphique :

```
plot_with_class(  
    the_data_points, weights_OR, results_OR,  
    "1.1 - Classification by simple perceptron on OR dataset",  
    -1, 2  
).show()
```

Le graphique montre une droite de séparation qui divise correctement les points, avec une légère marge pour les points  $[0, 1]$ ,  $[1, 0]$ , et  $[1, 1]$  qui sont classés dans la classe 1. La frontière de décision du perceptron permet de séparer les points correspondant à la sortie 1 de ceux correspondant à la sortie 0. Pour le cas du OU, le perceptron arrive à trouver une solution linéaire, ce qui est attendu puisqu'une seule droite suffit à séparer les points de ce problème. Cela montre que le perceptron simple est capable de résoudre correctement ce type de tâche de classification linéairement séparable.

## II. Apprentissage Widrow

### A. Stratégie de Développement

Pour implémenter l'algorithme de Widrow-Hoff (ou LMS), nous avons utilisé un perceptron simple afin de classer les points dans deux catégories distinctes. La stratégie adoptée repose sur la mise à jour progressive des poids après un certain nombre d'éléments (**batch\_size**), et non à chaque itération. Également, mettre à jour les poids après chaque élément pourrait orienter le modèle dans une mauvaise direction si cet élément est mal classé. Cette approche par lots permet de lisser les fluctuations causées par des données bruitées.

Afin de maximiser l'efficacité de l'apprentissage, à chaque nouvelle **epoch**, les données sont mélangées aléatoirement. Cela évite que le perceptron n'apprenne uniquement sur des éléments d'une seule classe en début d'apprentissage, ce qui pourrait provoquer une convergence prématurée dans une mauvaise direction. Le mélange des données permet donc une exploration plus équilibrée des deux classes, réduisant ainsi le risque d'un apprentissage biaisé.

Pour la mise à jour des poids, nous utilisons la règle suivante :

$$\begin{aligned} \text{nouveau\_poids}[i] \\ &= \text{ancien\_poids}[i] - \text{learning\_rate} \times (-(\text{resultat\_désiré} \\ &\quad - \text{resultat\_obtenu}) \times \text{dérivé\_fonction\_activation}(\text{poids} \cdot \text{données}) \times \text{données}[i]) \end{aligned}$$

L'erreur globale est calculée comme la somme des carrés des erreurs pour chaque élément :

$$\text{erreur\_globale} = \sum (\text{résultat\_désiré} - \text{résultat\_obtenu})^2$$

Une erreur nulle signifie que les classes sont correctement séparées.

Pour accélérer l'apprentissage, nous appliquons la fonction `sign()` en sortie du perceptron, car la fonction d'activation utilisée, `tanh`, retourne des valeurs continues entre -1 et 1. Sans cette transformation, il faudrait plus de temps pour que le perceptron converge vers des valeurs discrètes (-1 ou 1), ce qui ralentirait considérablement l'apprentissage.

En ce qui concerne le taux d'apprentissage, nous avons opté pour une valeur de 0,1, qui s'est avérée être un bon compromis entre la rapidité et la précision. Un taux plus élevé (> 0,1) peut empêcher le modèle de converger correctement, car les mises à jour seraient trop importantes. À l'inverse, un taux plus bas (< 0,1) ralentirait inutilement le processus d'apprentissage sans pour autant améliorer la précision finale.

## B. Ensemble Test 1

### 1. Graphiques des étapes d'apprentissage

Lors de l'entraînement sur le premier ensemble de test, les itérations montrent une convergence rapide. Cela est dû au `batch_size` de 1, car les éléments sont distincts, et à chaque nouvel élément nous pouvons mettre à jour nos poids. La droite de séparation évolue au fil des itérations, jusqu'à ce qu'elle divise clairement les deux classes.

Voici les résultats obtenus pour une séparation des données sur l'ensemble "Test1" en 5 époques :

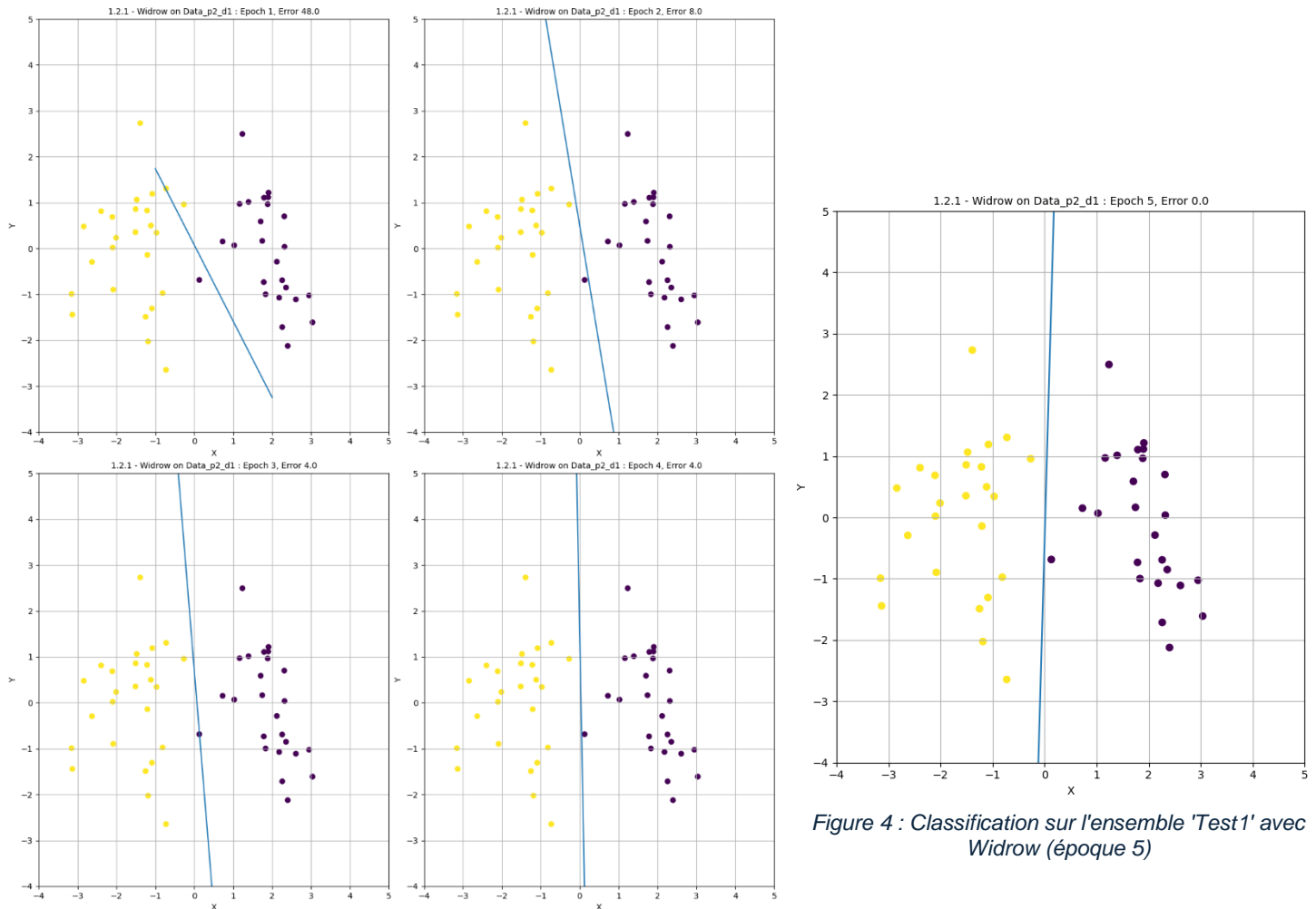


Figure 4 : Classification sur l'ensemble 'Test1' avec Widrow (époque 5)

Figure 4 : Classification sur l'ensemble 'Test1' avec Widrow (époques 1 à 4)

Comme nous pouvons le voir sur les graphiques ([Figure 3](#) et [Figure 4](#)), les deux classes sont représentées par deux couleurs : le jaune et le violet.

Au départ, à époque 1, la droite est placée aléatoirement ([Figure 3](#)) et au fil des époques, elle s'oriente de façon à couper les deux ensembles en deux, à l'époque 5 ([Figure 4](#)).



Voici en plus grand, la séparation des deux classes ([Figure 5](#)) :

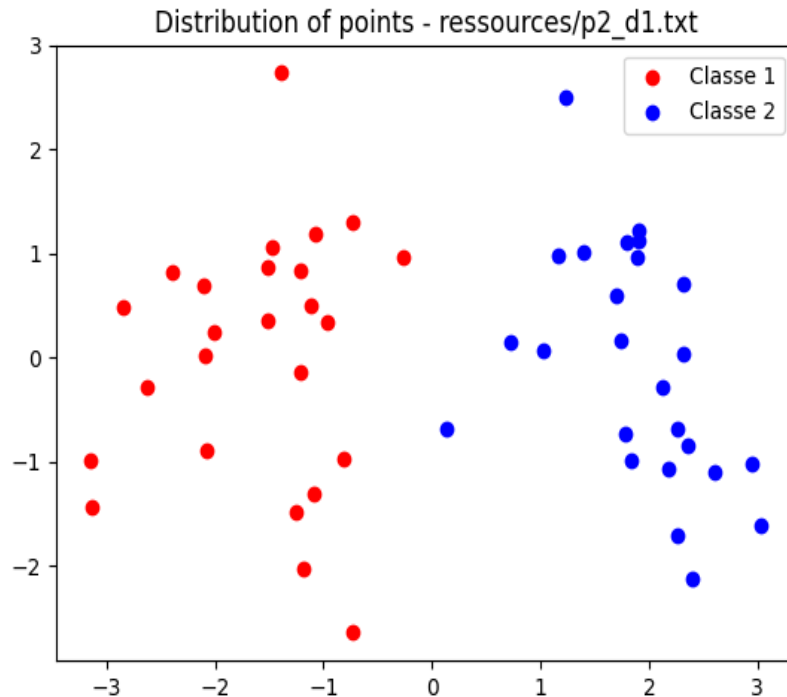


Figure 5 : Séparation des données pour l'ensemble 'Test1'

## 2. Erreur en fonction des itérations

En analysant l'évolution de l'erreur au fil des itérations, nous observons une diminution rapide de l'erreur. Cela montre que l'algorithme converge efficacement pour cet ensemble de données.

Voici la courbe d'erreur que nous avons obtenue pour les résultats précédants ([Figure 6](#)) :

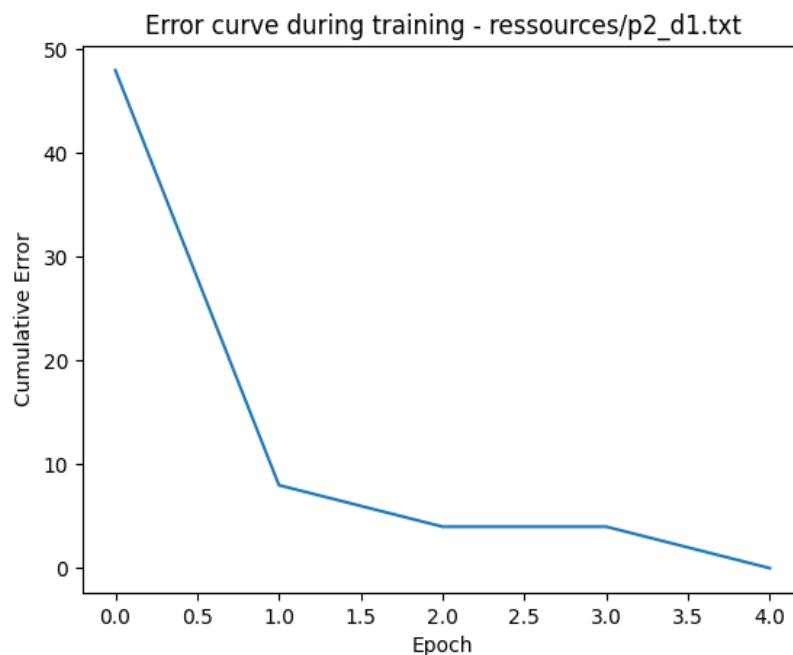


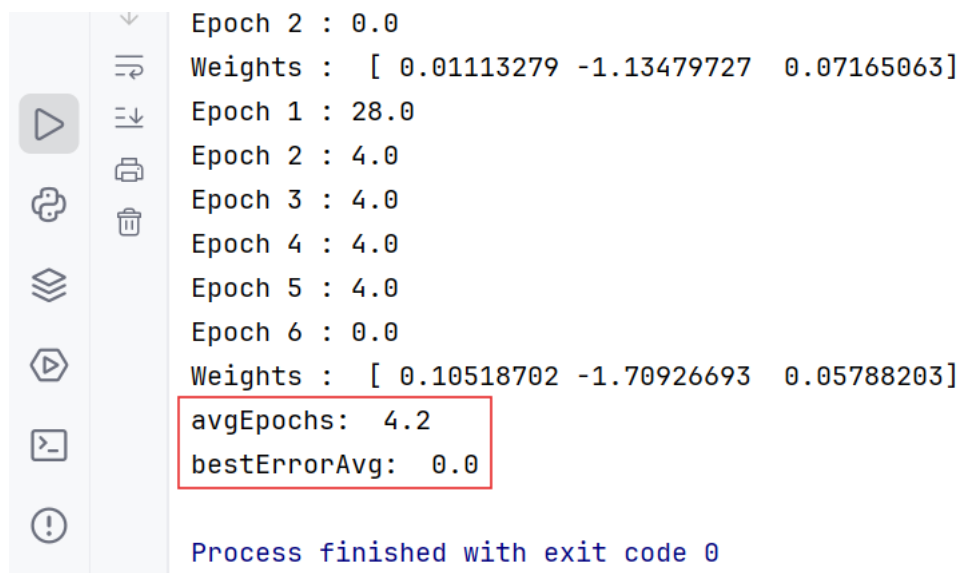
Figure 6 : Courbe d'erreur pour l'ensemble 'Test1' avec Widrow

La courbe commence avec une erreur de 48 (époque 1), ce qui s'explique par le fait que 4 points se trouvent dans la mauvaise classe (**Erreur ! Source du renvoi introuvable.**). Ensuite, plus nous avançons dans le nombre d'époques et plus l'erreur diminue pour finalement atteindre 0, ce qui indique que notre entraînement a trouvé une solution à notre problème de classification et que nos deux classes sont correctement séparées.

### 3. Tests avec initialisations différentes

Pour tester la robustesse de l'algorithme, nous avons effectué 100 essais avec différentes initialisations des poids (initialisation aléatoire). Dans chaque cas, l'algorithme parvient à converger rapidement vers une solution satisfaisante. En effet, c'est en environ 4 époques que le modèle arrive en moyenne à une séparation correcte des classes. L'algorithme semble donc peu sensible à l'initialisation des poids pour cet ensemble de données.

Voici les résultats que nous obtenons avec notre code pour 100 essais (Figure 7) :



```
Epoch 2 : 0.0
Weights : [ 0.01113279 -1.13479727  0.07165063]
Epoch 1 : 28.0
Epoch 2 : 4.0
Epoch 3 : 4.0
Epoch 4 : 4.0
Epoch 5 : 4.0
Epoch 6 : 0.0
Weights : [ 0.10518702 -1.70926693  0.05788203]
avgEpochs: 4.2
bestErrorAvg: 0.0

Process finished with exit code 0
```

Figure 7 : Résultat obtenu après 100 essais d'apprentissage avec Widrow sur l'ensemble 'Test1'

Avec 'avgEpochs' qui est le nombre d'époques en moyenne et 'bestErrorAvg' qui est le taux d'erreur à la fin de l'apprentissage.

## C. Ensemble Test 2

### 1. Graphiques des étapes d'apprentissage

Dans le deuxième ensemble de test, la droite de séparation n'arrive pas à séparer complètement les deux classes, ce qui est attendu puisque certains éléments de la classe 2 se trouvent dans la zone de la classe 1. Ces deux éléments rendent impossible une séparation parfaite par une simple droite. Le graphique montre que malgré ces contraintes, l'algorithme tente de minimiser l'erreur autant que possible.

Voici les résultats obtenus pour une séparation des données sur l'ensemble "Test2" en 15 époques :

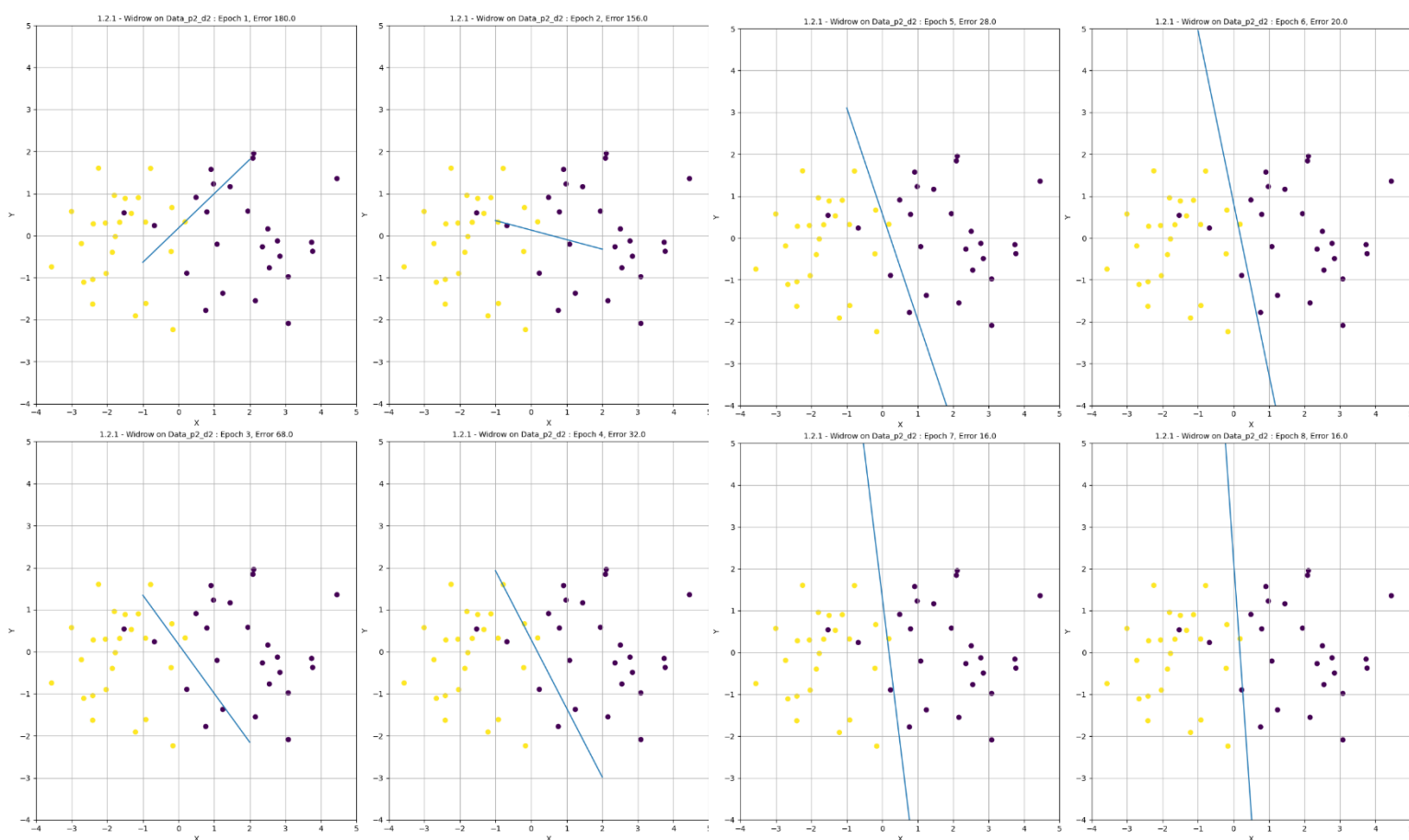


Figure 9 : Classification sur l'ensemble 'Test2' avec Widrow (époques 1 à 4)

Figure 9 : Classification sur l'ensemble 'Test2' avec Widrow (époques 5 à 8)

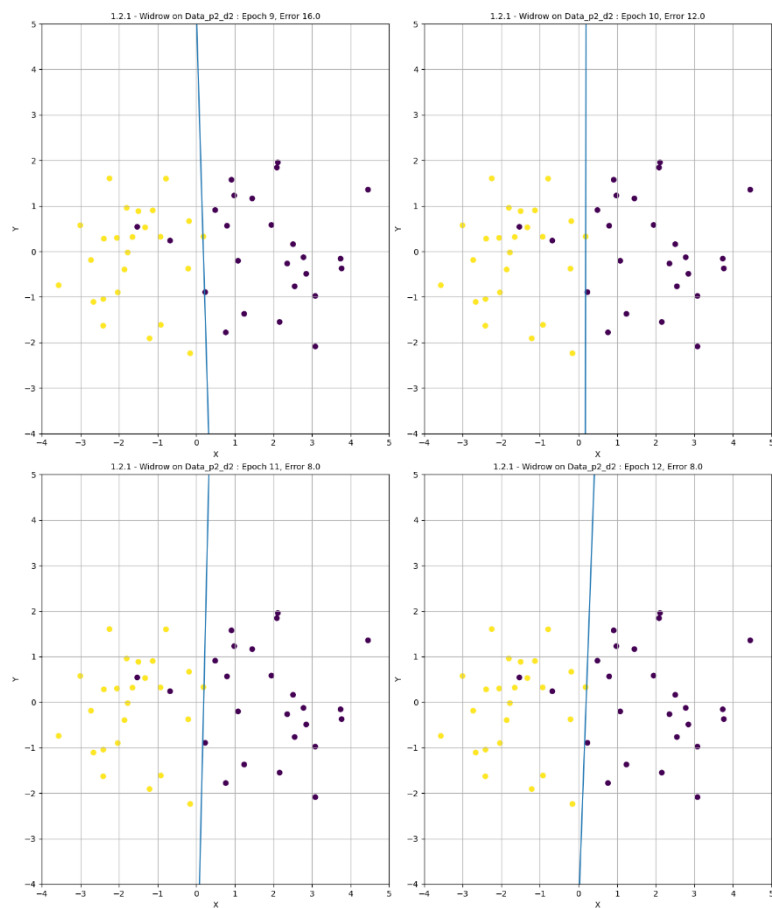


Figure 10 : Classification sur l'ensemble 'Test2' avec Widrow (époques 9 à 12)

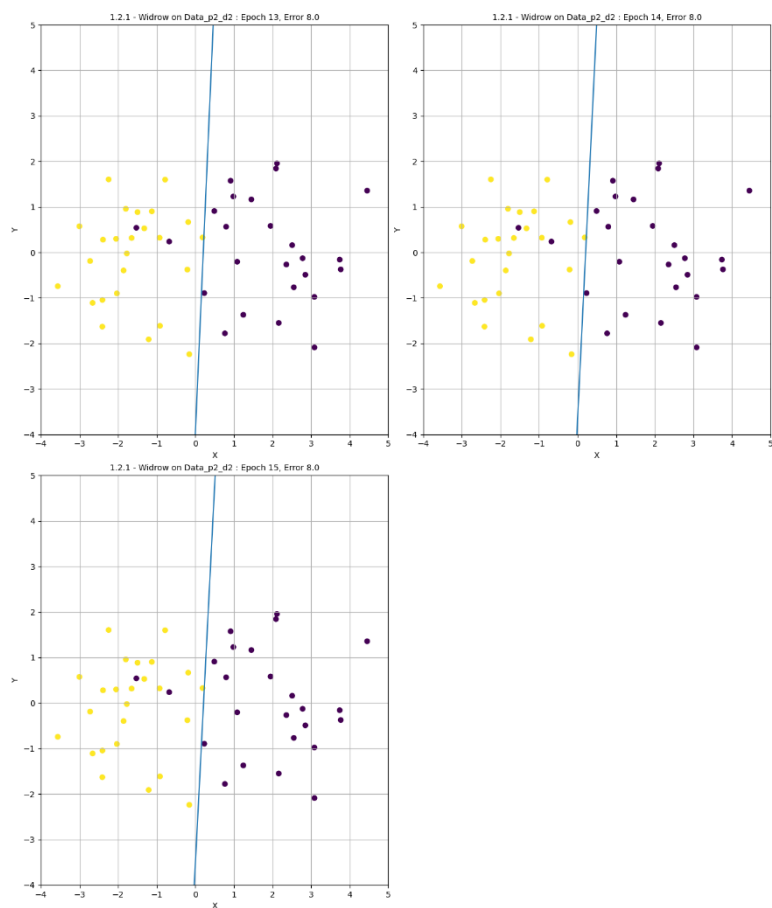


Figure 11 : Classification sur l'ensemble 'Test2' avec Widrow (époques 13 à 15)

Tout d'abord, quand l'apprentissage commence nous pouvons voir que la courbe de séparation est complètement fausse car elle coupe la première classe en deux ([Figure 8](#), époque 1). Ensuite comme pour l'apprentissage sur l'ensemble 'Test1', notre modèle progresse de plus en plus ([Figure 8](#), [Figure 9](#), [Figure 10](#), [Figure 11](#), époques 2 à 14). Et une fois arrivé au bout de l'apprentissage, nous avons bien une droite qui sépare les deux classes en deux avec bien sur, les deux éléments perturbateurs qui se retrouve de l'autre côté de la ligne ([Figure 11](#), époque 15).

Voici la répartition de l'ensemble 'Test2' (Figure 12) :

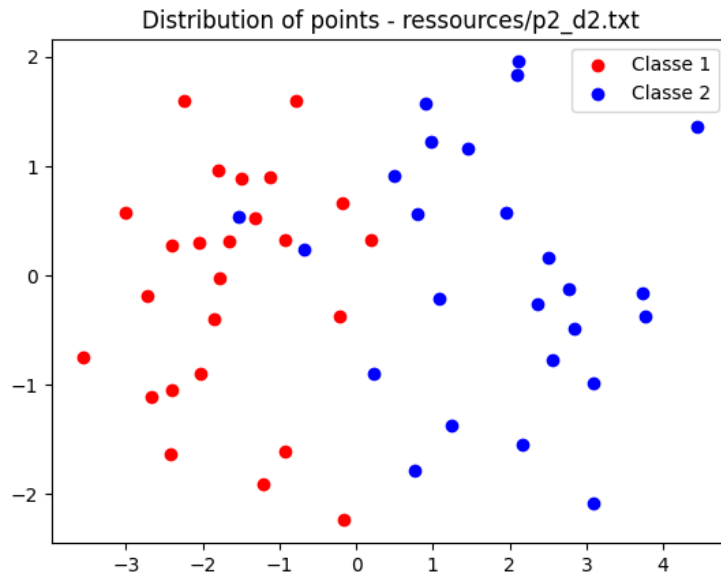


Figure 12 : Séparation des données pour l'ensemble 'Test2'

## 2. Erreur en fonction des itérations

Contrairement au premier ensemble de test, l'erreur ne parvient jamais à atteindre zéro. La présence d'éléments mal classés empêche une convergence totale. L'erreur se stabilise autour de 8, ce qui correspond aux deux points qui ne peuvent être correctement classés.

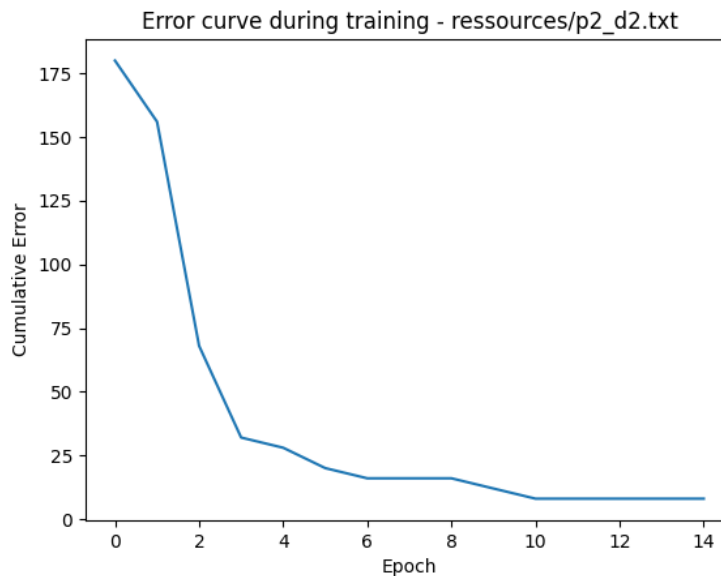
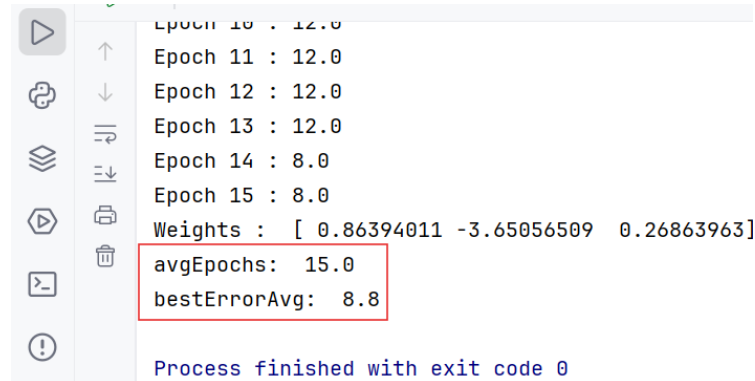


Figure 13 : Courbe d'erreur pour l'ensemble 'Test2' avec Widrow

Cette courbe (Figure 13) nous permet de voir qu'au début, comme vu précédemment, la répartition des classes n'est pas bonne, avec une erreur de 180. Ensuite, le modèle progresse rapidement pour tomber à une erreur de 20 à la 6<sup>ème</sup> époque. Puis la courbe s'affine jusqu'à la 15<sup>ème</sup> époque pour obtenir le résultat attendu (une erreur de 8).

### 3. Tests avec initialisations différentes

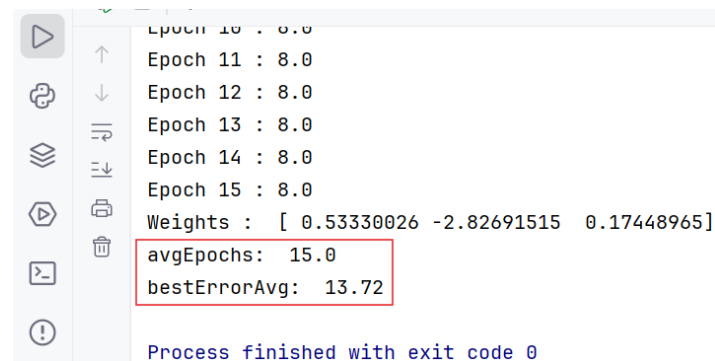
Lorsque nous testons différentes initialisations pour le deuxième ensemble (initialisation aléatoire), l'algorithme met plus de temps à converger. La présence d'éléments perturbateurs et d'un `batch_size` plus élevé rend l'apprentissage plus sensible à la position initiale de la droite de séparation. Ainsi, l'initialisation de la droite de séparation peut retarder la convergence vers la solution optimale. Cela montre que pour des ensembles plus complexes, l'algorithme de Widrow-Hoff peut nécessiter un ajustement plus fin des paramètres pour assurer une convergence rapide. Dans notre cas il faut en moyenne 15 époques pour que le modèle trouve la meilleure solution (avec une `batch_size` de 50, Figure 14). Et nous pouvons voir qu'avec un `batch_size` égal à 1, l'apprentissage est beaucoup moins bon (Figure 15), ce qui s'explique par le fait qu'un élément perturbateur peut venir changer les poids dans la mauvaise direction, ce qui retarde donc l'apprentissage.



```
Epoch 10 : 12.0
Epoch 11 : 12.0
Epoch 12 : 12.0
Epoch 13 : 12.0
Epoch 14 : 8.0
Epoch 15 : 8.0
Weights : [ 0.86394011 -3.65056509 0.26863963]
avgEpochs: 15.0
bestErrorAvg: 8.8

Process finished with exit code 0
```

Figure 14 : Résultat obtenu après 100 essais d'apprentissage avec Widrow sur l'ensemble 'Test2' (époques max 15 et `batch_size` à 50)



```
Epoch 10 : 8.0
Epoch 11 : 8.0
Epoch 12 : 8.0
Epoch 13 : 8.0
Epoch 14 : 8.0
Epoch 15 : 8.0
Weights : [ 0.53330026 -2.82691515 0.17448965]
avgEpochs: 15.0
bestErrorAvg: 13.72

Process finished with exit code 0
```

Figure 15 : Résultat obtenu après 100 essais d'apprentissage avec Widrow sur l'ensemble 'Test2' (époques max 15 et `batch_size` à 1)

Avec 'avgEpochs' qui est le nombre d'époques en moyenne et 'bestErrorAvg' qui est le taux d'erreur à la fin de l'apprentissage.

### III. Perceptron Multicouche

#### A. Stratégie de Développement

Dans cette partie, nous avons implémenté un perceptron multicouche simple, composé d'une couche d'entrée, d'une couche cachée, et d'une couche de sortie. L'architecture est définie comme suit :

- **Couche d'entrée** : Cette couche reçoit deux entrées,  $x_1$  et  $x_2$ , représentant les caractéristiques du problème à traiter. Nous y ajoutons un biais, noté 1, pour améliorer la convergence du modèle.
- **Couche cachée** : La couche cachée contient deux neurones. Les poids synaptiques associés à cette couche sont représentés par une matrice  $w_1$  de taille  $3 \times 2$ , où les trois lignes correspondent aux poids appliqués aux entrées  $[1, x_1, x_2]$ , et les deux colonnes représentent les deux neurones de la couche cachée. Nous appliquons une **fonction d'activation sigmoïde** à la sortie de chaque neurone de la couche cachée pour introduire de la non-linéarité dans le modèle, ce qui permet de mieux capter des relations complexes entre les variables.
- **Couche de sortie** : La couche de sortie se compose d'un unique neurone. Les poids synaptiques associés sont représentés par un vecteur  $w_2$  de trois éléments (pour le biais et les deux sorties de la couche cachée). Ce neurone de sortie applique également une fonction d'activation sigmoïde à sa somme pondérée pour générer une prédiction.

L'algorithme d'apprentissage suivi est le suivant :

1. Calcul de la somme pondérée des entrées de la couche cachée.
2. Application de la fonction d'activation sigmoïde pour chaque neurone de la couche cachée.
3. Utilisation de ces sorties cachées pour calculer la somme pondérée des entrées du neurone de sortie.
4. Application de la fonction sigmoïde pour produire la sortie finale.

La fonction d'activation sigmoïde est définie par l'équation suivante :

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

Cette fonction contraint les valeurs des neurones entre 0 et 1, facilitant ainsi l'apprentissage.

## B. Résultats

Nous avons testé ce perceptron multicouche avec des entrées  $x_1 = 1$  et  $x_2 = 1$ , et les poids synaptiques sont initialisés comme suit :

- **Poids de la couche cachée ( $w_1$ ) :**

$$w_1 = \begin{pmatrix} -0,5 & 0,5 \\ 2,0 & 0,5 \\ -1,0 & 1,0 \end{pmatrix}$$

- **Poids de la couche de sortie ( $w_2$ ) :**

$$w_2 = (2.0 \quad -1.0 \quad 1.0)$$

Les sorties des neurones de la couche cachée sont :

- **Neurone 1 :**

$$Sortie = \sigma(-0.5 \times 1 + 2.0 \times 1 + (-1.0) \times 1) = \sigma(0.5) \approx 0.622$$

- **Neurone 2 :**

$$Sortie = \sigma(0.5 \times 1 + 0.5 \times 1 + 1.0 \times 1) = \sigma(2.0) \approx 0.881$$

La sortie du neurone de la couche de sortie est ensuite calculée comme suit :

$$Sortie\ finale = \sigma(2.0 \times 1 - 1.0 \times 0.622 + 1.0 \times 0.881) = \sigma(2.259) \approx 0.905$$

Ainsi, la sortie finale du perceptron multicouche pour les entrées  $x_1 = 1$  et  $x_2 = 1$  est environ 0.905.

L'analyse des résultats montre que les sorties de la couche cachée, activées par la fonction sigmoïde, permettent de capturer des relations non-linéaires entre les entrées, et le perceptron multicouche réussit à produire une sortie significative après l'application des poids à la couche de sortie.



## IV. Apprentissage Multicouche

### A. Stratégie de Développement

- **Choix du Taux d'Apprentissage**

Un taux d'apprentissage de **0.5** a été choisi. Ce choix a été fait de manière empirique après plusieurs essais pour garantir une convergence rapide tout en évitant les oscillations. Un taux trop élevé pourrait entraîner un apprentissage instable, et un taux trop faible ralentirait l'apprentissage.

- **Critère d'Arrêt**

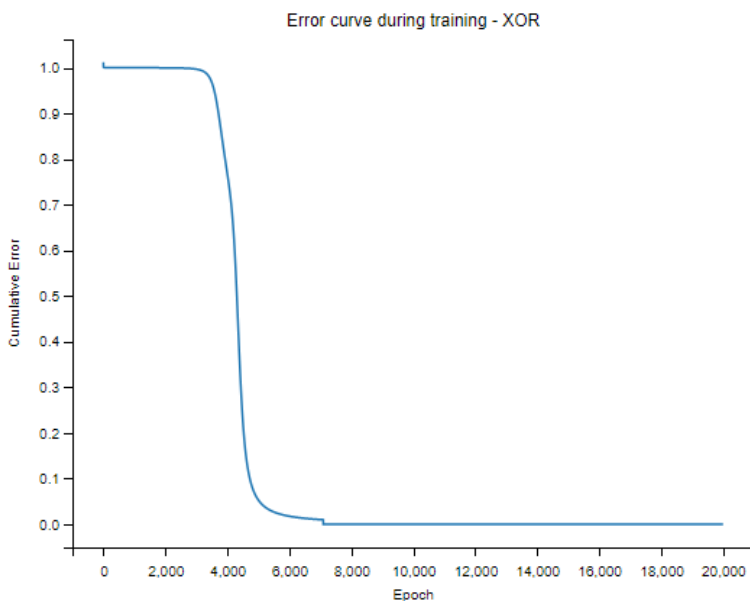
Le critère d'arrêt est basé sur l'erreur cumulée par époque. Nous avons fixé un seuil de **0.01** en dessous duquel l'apprentissage s'arrête. Ce seuil garantit une précision suffisante pour la classification XOR tout en minimisant la durée de l'entraînement.

- **Optimisation**

Nous avons utilisé une méthode par lots (batch size de **4**) afin de mettre à jour les poids après avoir accumulé les gradients pour toutes les données d'entrée. Cela permet une optimisation plus stable par rapport à une approche purement stochastique. De plus, la dérivée de la fonction sigmoïde a été utilisée pour calculer les gradients et ajuster les poids.

### B. Erreur en fonction des itérations

L'évolution de l'erreur en fonction des itérations est représentée graphiquement ci-dessous, permettant de visualiser la convergence de l'apprentissage.



- **La Convergence**

La courbe montre une décroissance continue de l'erreur jusqu'à atteindre le critère d'arrêt fixé à 0.01. Cela témoigne d'une bonne convergence de l'apprentissage, prouvant que le réseau parvient à minimiser l'erreur pour la tâche XOR. Le réseau parvient à généraliser correctement après plusieurs milliers d'époques.

➤ Test sur le XOR

Les résultats des tests du réseau sur les différentes entrées de la fonction XOR sont les suivants :

$$x = [0, 0] / y = 0$$

$$x = [0, 1] / y = 1$$

$$x = [1, 0] / y = 1$$

$$x = [1, 1] / y = 0$$

Le réseau est capable de correctement classer les quatre combinaisons d'entrées du problème XOR, comme attendu.

## C. Droites séparatrices

- **Graphique des Droites Séparatrices**

Les trois droites séparatrices obtenues à partir des poids des neurones cachés et de sortie sont illustrées dans la figure ci-dessous :

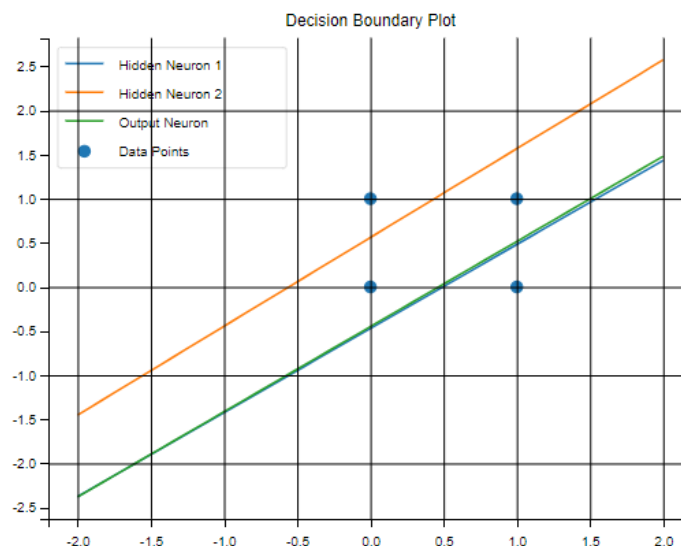


Figure 17 : Graphique des limites de décision

- **Rôle dans la Classification XOR**

- **Droite Neurone Caché 1** (orange) : Cette droite correspond au premier neurone caché. Son rôle est de séparer les données dans l'espace de manière à faciliter la tâche du neurone de sortie.
- **Droite Neurone Caché 2** (vert) : Cette droite correspond au second neurone caché. Elle est complémentaire à la première droite, permettant une séparation non linéaire des données.

- **Droite du Neurone de Sortie** (bleu) : Cette droite représente la séparation finale effectuée par le neurone de sortie pour classer les points dans l'une des deux catégories XOR (0 ou 1).

Les droites des neurones cachés découpent l'espace des entrées de manière à rendre linéairement séparable un problème initialement non linéairement séparable. Le neurone de sortie effectue ensuite une classification binaire à partir de ces transformations.

## V. Classification par Full-Connected

### A. Stratégie de Développement

Pour cette partie, une approche Full-Connected a été utilisé pour effectuer la classification des images selon les dix catégories définies (*Jungle [J]*, *Plage [P]*, *Monuments [M]*, *Bus [B]*, *Dinosaures [D]*, *Éléphants [E]*, *Fleurs [F]*, *Chevaux [C]*, *Montagne [M]*, *Plats [P]*) [dans cet ordre pour les confusion matrix]. Le modèle construit est un réseau de neurones séquentiel comprenant :

- **Couche d'entrée** : Les descripteurs extraits ou concaténés des images.
- **Architecture du réseau** :
  - Une première couche dense avec 512 neurones et une fonction d'activation *ReLU*, suivie d'un *dropout* (0,5).
  - Une seconde couche dense avec 256 neurones (*ReLU*) suivie d'un *dropout* (0,5).
  - Une troisième couche dense avec 128 neurones (*ReLU*).
  - Une quatrième couche dense avec 64 neurones (*ReLU*).
  - Une couche de sortie avec 10 neurones et une fonction d'activation *softmax*.

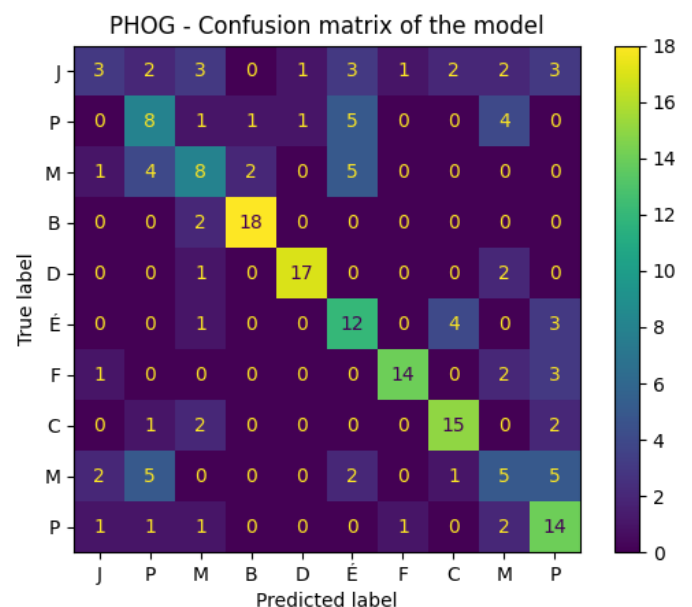
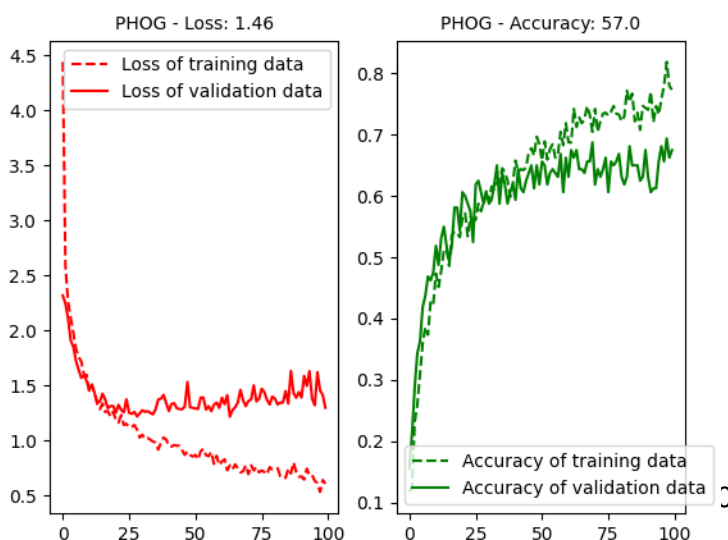
Le réseau est entraîné avec une perte *categorical\_crossentropy*, un optimiseur *Adam*, et la métrique d'évaluation est la précision (*accuracy*). Les données sont divisées en 80 % pour l'entraînement et 20 % pour les tests, avec une validation interne de 20 % sur l'ensemble d'entraînement. Le nombre d'époques a été fixé à 100 avec une taille de batch de 32.

### B. Comparaison des caractéristiques

Comparer la qualité de la discrimination en fonction des différentes caractéristiques d'entrée utilisées.

Six types de descripteurs ont été testés pour évaluer leur impact sur la performance du réseau Full-Connected :

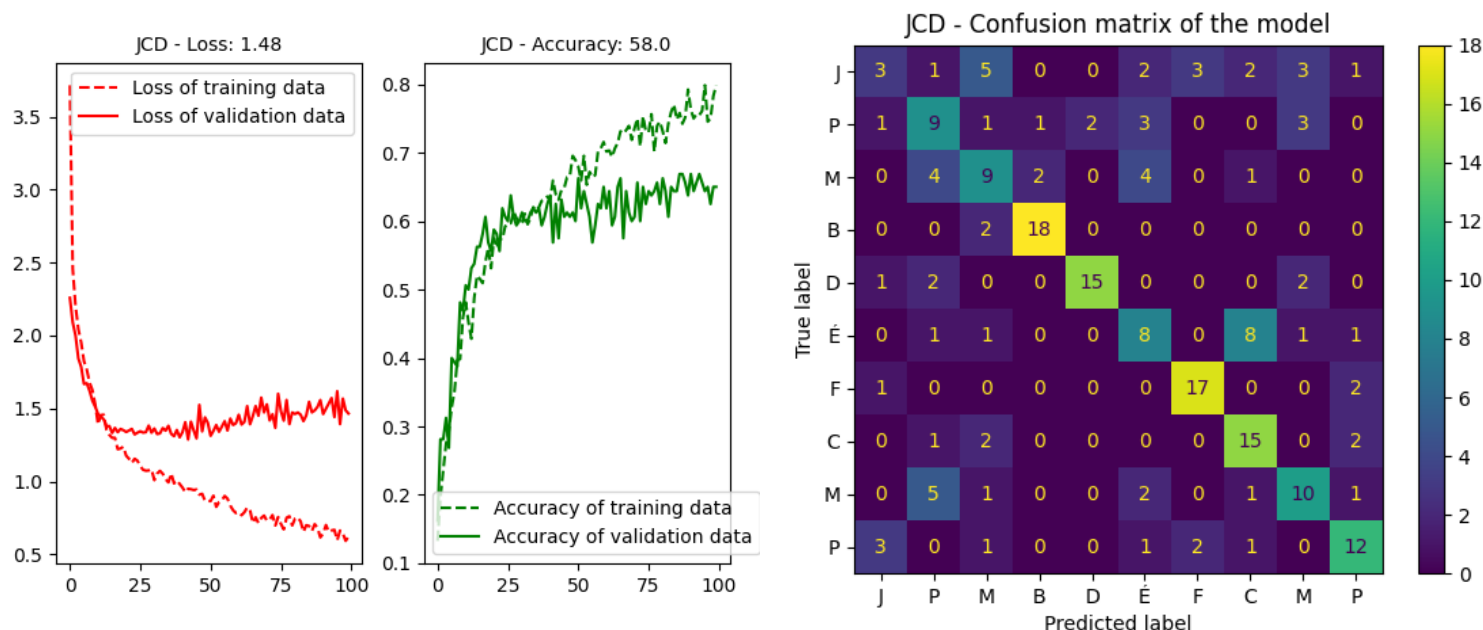
#### 1. PHOG (*Pyramid Histogram of Oriented Gradients*)



Loss	Accuracy (Full Connected)	Accuracy (KppV)
1.4564	56.99 %	51 %

Le descripteur PHOG capture les caractéristiques structurales et les gradients orientés des images. Bien qu'il permette une classification relativement correcte (57 % de précision), ses performances restent limitées pour des catégories nécessitant davantage de discrimination basée sur les couleurs ou les textures (ex. : *Fleurs* ou *Plage*). Il est plus adapté pour des classes avec des motifs bien définis (ex. : *Bus* ou *Dinosaures*), mais la perte élevée montre qu'il ne capte pas toute la variabilité des données.

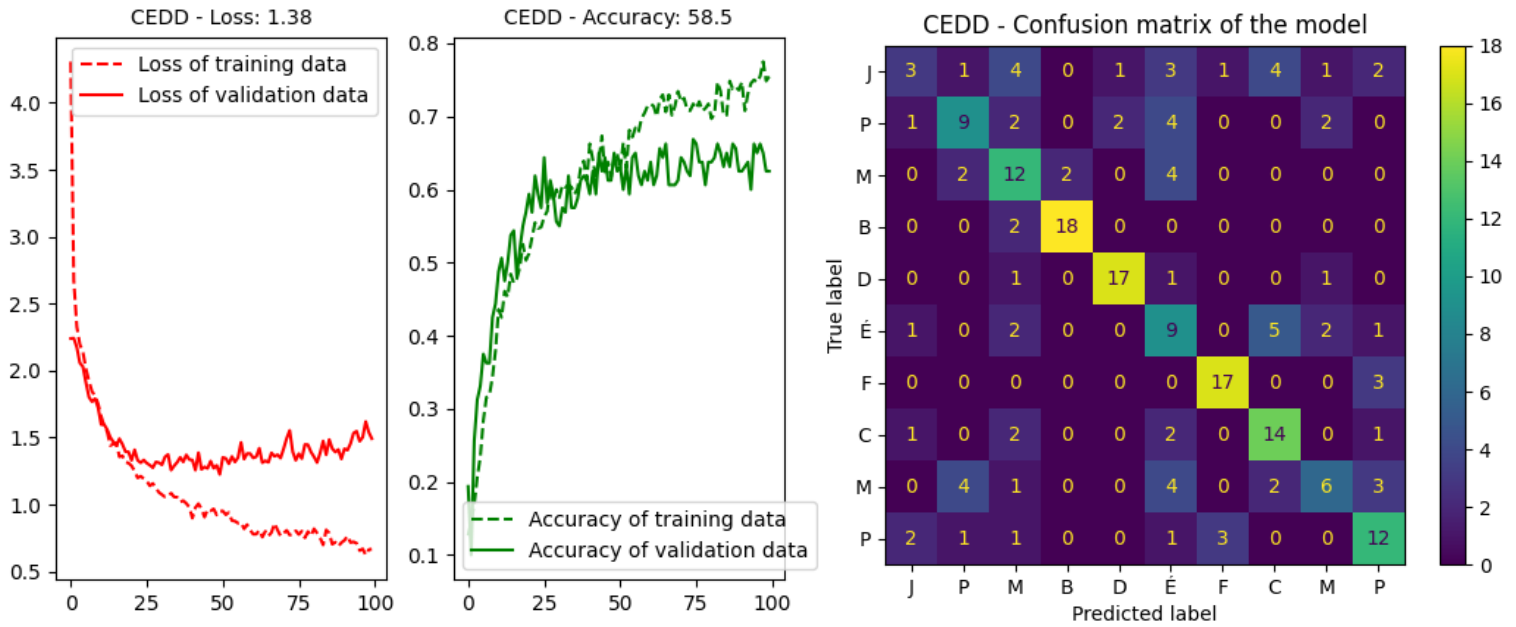
## 2. JCD (Joint Composite Descriptor)



Loss	Accuracy (Full Connected)	Accuracy (KppV)
1.4758	57.99 %	51 %

Le descripteur JCD combine des informations de couleur et de texture, ce qui le rend pertinent pour des classes visuellement complexes. Malgré cela, sa précision est légèrement inférieure à celle d'autres descripteurs comme *CEDD* ou *Fuzzy Color Histogram* (dans la suite). Les résultats montrent qu'il souffre d'un manque de robustesse face à des classes présentant des similitudes (ex. : *Montagne* et *Plage*).

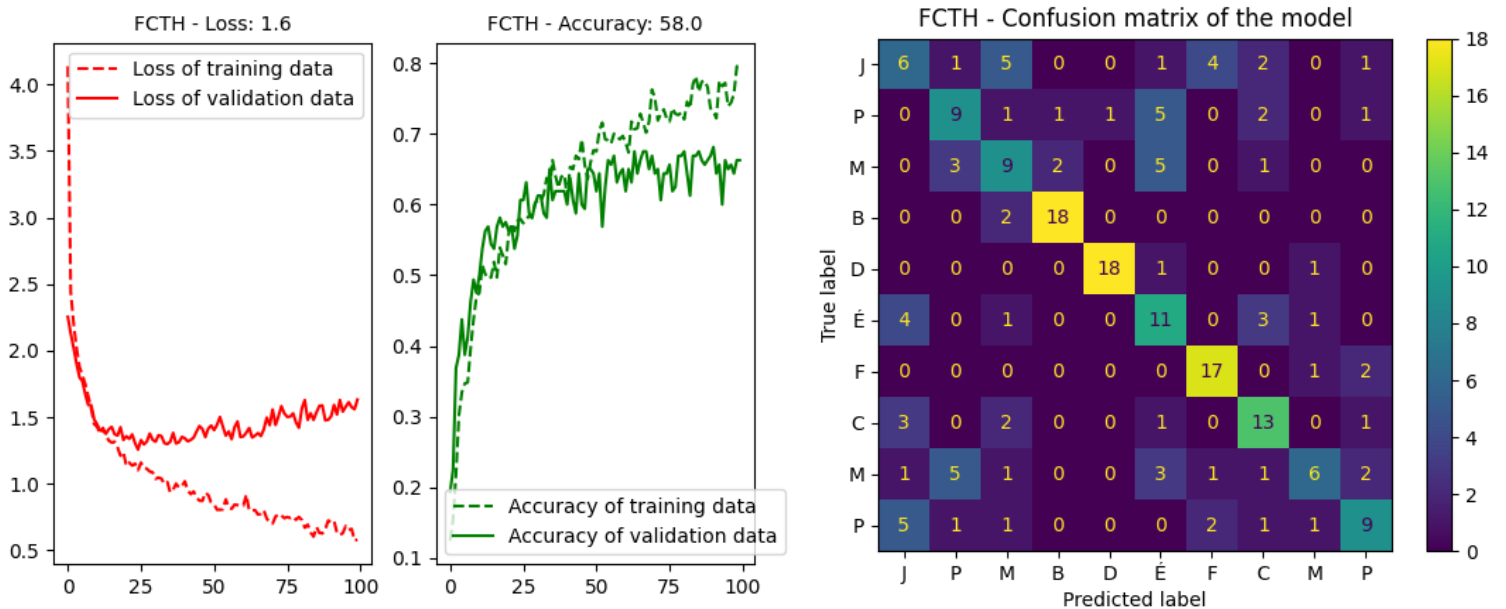
### 3. CEDD (Color and Edge Directivity Descriptor)



Loss	Accuracy (Full Connected)	Accuracy (KppV)
1.3766	58.49 %	51 %

Le descripteur CEDD intègre des informations sur la couleur et les bords. Avec une précision de 58.49 %, il surpasse légèrement PHOG et JCD. Cela reflète sa capacité à mieux discriminer des classes visuellement distinctes (ex. : *Dinosaures*, *Fleurs*). Cependant, sa perte relativement élevée montre une certaine difficulté à traiter des classes aux motifs plus complexes ou hétérogènes.

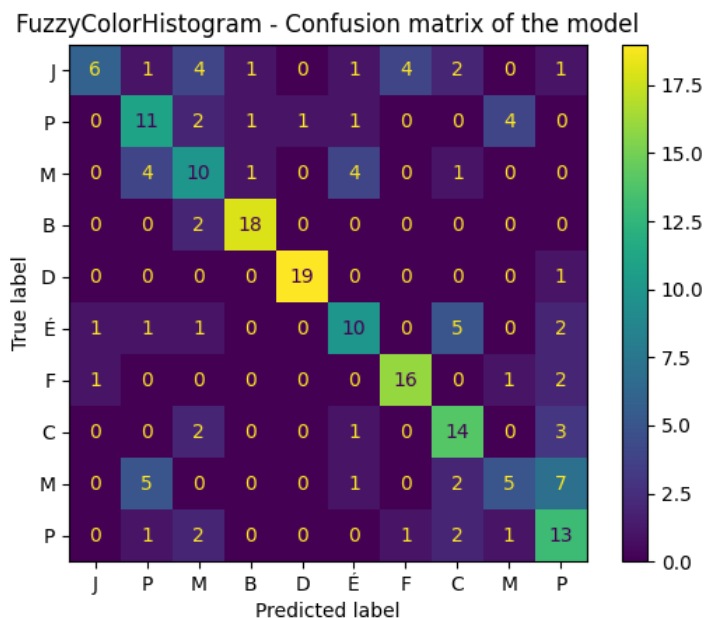
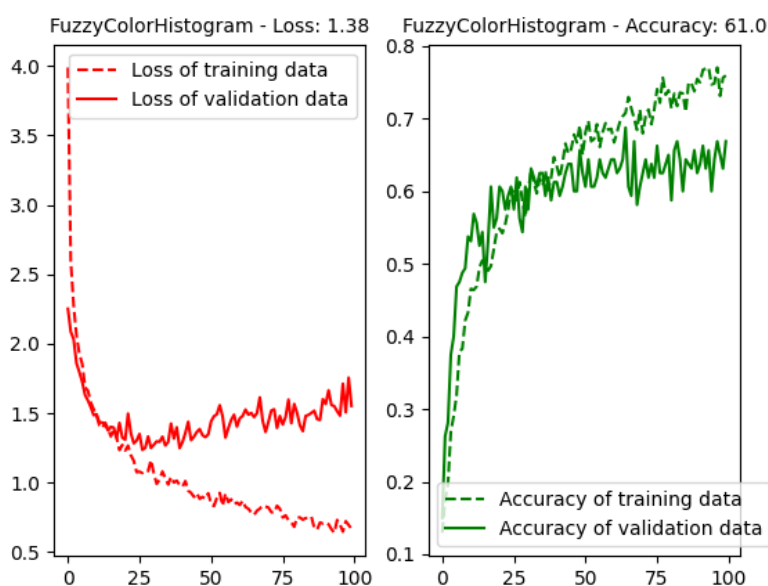
### 4. FCTH (Fuzzy Color and Texture Histogram)



Loss	Accuracy (Full Connected)	Accuracy (KppV)
1.5989	57.99 %	51 %

Le descripteur FCTH met l'accent sur la combinaison de textures et de couleurs en utilisant une approche floue. Cependant, ses performances sont légèrement inférieures à celles de *CEDD* et *Fuzzy Color Histogram*. Une perte plus élevée (1.5989) indique qu'il a du mal à généraliser, notamment pour des classes aux caractéristiques partagées (ex. : *Montagne* et *Plats*). Cela peut être dû à une moins bonne sensibilité aux subtilités des variations de couleur.

## 5. Fuzzy Color Histogram



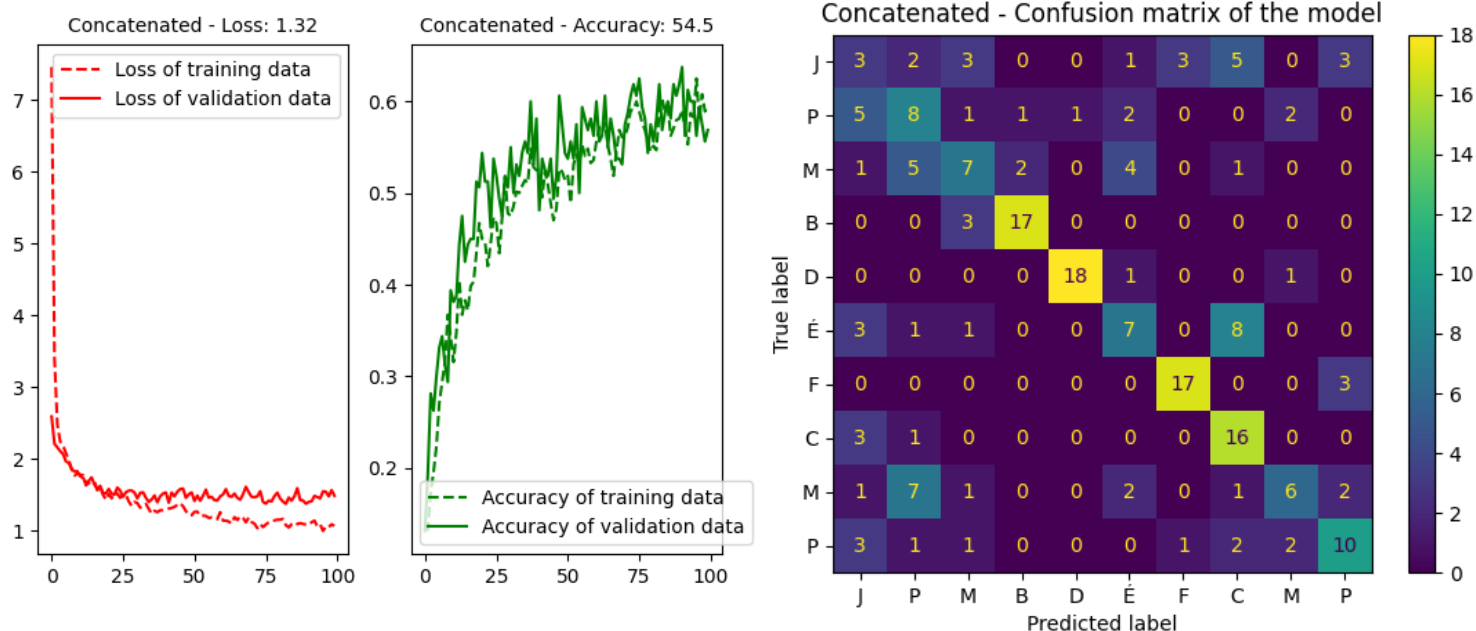
Loss	Accuracy (Full Connected)	Accuracy (KppV)
1.3775	61.00 %	51 %

Ce descripteur est le meilleur parmi tous les candidats, avec une précision de 61 % et une perte relativement faible. Sa force réside dans sa capacité à discriminer efficacement les images en se basant sur des informations de couleur plus granulaires et sur une approche floue qui capture les nuances. Les performances sont particulièrement bonnes pour des classes comme :

- Dinosaur, où les variations de couleur jouent un rôle.
- Bus, qui se distingue par des tons clairs et homogènes.

Cependant, des confusions subsistent pour des classes comme *Montagne* ou *Jungle*, où les caractéristiques de couleur peuvent être moins discriminantes.

## 6. Descripteurs concaténés : Combinaison de tous les descripteurs.



Loss	Accuracy (Full Connected)	Accuracy (KppV)
1.3154	54.50 %	51 %

Les descripteurs concaténés intègrent toutes les caractéristiques extraites (PHOG, JCD, CEDD, FCTH, Fuzzy Color Histogram). Bien que cette approche soit théoriquement plus riche, elle a produit une performance inférieure, avec une précision de seulement 54.50 %. Cela peut être due à une surcharge d'informations non pertinentes ou redondantes qui perturbent l'apprentissage du réseau.

### → Synthèse des performances :

DESCRIPTEUR	FORCES	FAIBLESSES
<b>PHOG</b>	Capture des motifs et des formes structurées.	Faible discrimination pour des classes basées sur les couleurs.
<b>JCD</b>	Bonne combinaison de couleur et texture.	Moins efficace pour des classes complexes.
<b>CEDD</b>	Bonne performance pour des classes distinctes.	Difficulté avec des motifs hétérogènes.
<b>FCTH</b>	Approche floue pour texture et couleur.	Sensibilité réduite aux nuances.
<b>FUZZY COLOR HISTOGRAM</b>	Excellente discrimination basée sur la couleur.	Limité pour des classes avec peu de variations.
<b>CONCATENES</b>	Plus grande richesse d'informations.	Surcharge et perte d'efficacité.



## C. Paramétrage optimal

Le paramétrage optimal pour cette classification a été obtenu en combinant :

- **Descripteur** : *Fuzzy Color Histogram*, qui a donné les meilleurs résultats en termes de précision.
- **Architecture du réseau** : Une architecture Full-Connected bien équilibrée, intégrant des couches denses et des mécanismes de régularisation (comme le *dropout*).
- **Optimisation** : Utilisation de l'optimiseur *Adam* avec un apprentissage sur 100 époques.

Ce paramétrage a permis d'atteindre une précision supérieure à 60 %, avec des performances constantes et une bonne généralisation sur l'ensemble de test.

## D. Évaluation par classe d'images

L'analyse de la matrice de confusion montre des performances variées selon les catégories :

- **Catégories bien discriminées** : Les classes comme *Dinosaures* et *Bus* ont été bien reconnues grâce à leurs caractéristiques distinctives.
- **Classes confondues** : Les catégories comme *Plats* et *Montagne* montrent une confusion notable, probablement en raison de similitudes visuelles dans leurs descripteurs.
- **Variabilité intra-classe** : Certaines classes, comme *Fleurs*, ont montré une variabilité importante, ce qui complique leur classification précise.

## E. Comparaison avec K-Plus-Proches-Voisins (KppV)

Un KppV avec  $k = 5$  a été utilisé comme algorithme de base pour comparer les performances. Les résultats montrent une précision constante de 51 % pour tous les descripteurs.

- **Observation principale** : Le modèle Full-Connected surpasse largement KppV, indiquant sa capacité à capturer des relations complexes entre les caractéristiques.
- **Limites de KppV** : L'algorithme repose uniquement sur des distances dans l'espace des caractéristiques, ce qui le rend moins performant face à des données complexes.

## VI. Classification par Deep Learning

### A. Architecture et stratégie d'apprentissage :

#### 1. Modèle Simple :

- **Architecture** : Ce modèle comprend une couche convolutive avec 32 filtres de taille 3×3, une couche de pooling 2×2 pour la réduction dimensionnelle, et une couche dense entièrement connectée pour la classification en sortie (10 neurones avec activation *softmax*).
- **Objectif** : Il sert de point de référence pour comparer les performances des modèles plus complexes. Sa simplicité permet une convergence rapide, mais il est limité dans la capture des caractéristiques complexes des images.

#### 2. Modèle Profond (Deep Model) :

- **Architecture** :
  - Deux couches convolutives successives, la première avec 8 filtres 5×5, et la seconde avec 16 filtres 3×3, chacune suivie d'une couche de pooling 2×2.
  - Une couche de Flatten pour transformer les caractéristiques en un vecteur, puis une couche dense en sortie.
- **Stratégie** : En augmentant la profondeur, le modèle peut détecter des motifs plus abstraits. La première couche identifie des caractéristiques basiques (bords, textures), tandis que les couches suivantes capturent des relations plus complexes entre ces caractéristiques.

#### 3. Modèle de Transfert Learning :

- **Base pré-entraînée** : Le modèle utilise **Xception**, un réseau convolutif avancé pré-entraîné sur ImageNet, connu pour ses performances élevées dans la reconnaissance d'images.
- **Architecture personnalisée** :
  - Les couches convolutives de la base sont gelées pour préserver les caractéristiques générales.
  - Une couche de pooling global (*GlobalAveragePooling2D*) est ajoutée pour compresser les cartes de caractéristiques. Ensuite, une couche dense de 512 neurones (ReLU) et une couche de sortie (softmax) sont utilisées pour la classification spécifique.
- **Objectif** : Exploiter les connaissances préexistantes pour des tâches spécifiques tout en réduisant le temps et les ressources nécessaires à l'entraînement.

#### 4. Modèle avec Data Augmentation :

- **Prétraitement** : Ce modèle inclut une couche d'augmentation des données (rotation aléatoire et symétrie horizontale) pour générer des variations artificielles des images.
- **Architecture** : Similaire au modèle profond, mais avec des filtres plus nombreux (32 et 64). Cela permet au modèle d'exploiter des données diversifiées pour une meilleure généralisation.
- **Stratégie** : L'objectif est de contrer le manque de diversité des données d'entraînement et d'améliorer les performances sur des images inconnues.

## B. Comparaison des caractéristiques du réseau

Pour analyser l'impact des paramètres, les tests sont réalisés sur des modèles avec des variations dans la profondeur, la largeur et les hyperparamètres.

#### 1. Profondeur (Nombre de couches) :

- Le modèle simple (1 couche convolutive) montre des performances correctes, mais limitées pour des images complexes.
- En ajoutant une seconde couche convolutive (modèle profond), les résultats s'améliorent significativement car le réseau peut détecter des relations plus subtiles entre les caractéristiques.

#### 2. Largeur (Nombre de filtres) :

- Des filtres plus nombreux (16, 32, 64) permettent une meilleure détection de motifs variés. Cependant, cela augmente le risque de surajustement si le modèle est trop complexe pour le volume de données disponible.

#### 3. Taux d'apprentissage :

- Un taux fixe est utilisé avec l'optimiseur Adam. Bien qu'efficace, il serait pertinent de tester des stratégies de taux dynamiques (scheduler ou warm-up) pour observer leur impact sur la convergence.

## C. Comparaison avec Full-Connected

### - Performances :

- Les réseaux convolutifs (CNN) surpassent largement les architectures *Full-Connected*, qui traitent les pixels individuellement, sans considérer les relations spatiales entre eux.
- Les CNN exploitent les propriétés locales des images (bords, textures, motifs), cruciales pour la classification visuelle.

### - Limites du Full-Connected :

- Nécessite un plus grand nombre de paramètres et est sujet à un surajustement rapide. Sa complexité rend la convergence lente sans amélioration notable des résultats.

## D. Data Augmentation

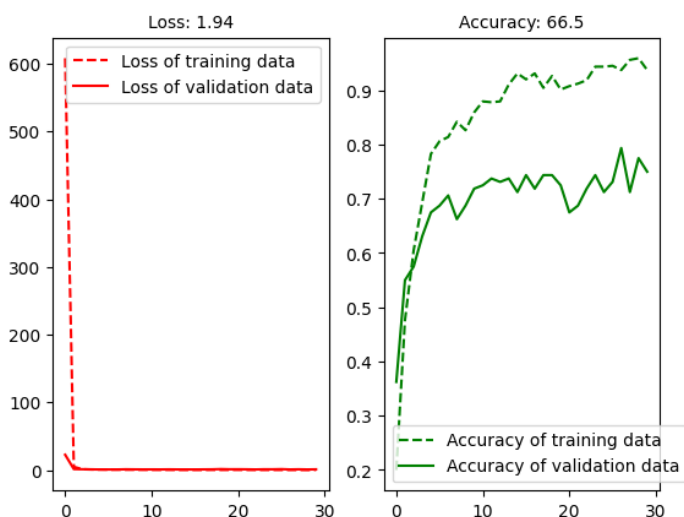
### - Mise en œuvre :

- Deux techniques d'augmentation sont utilisées :

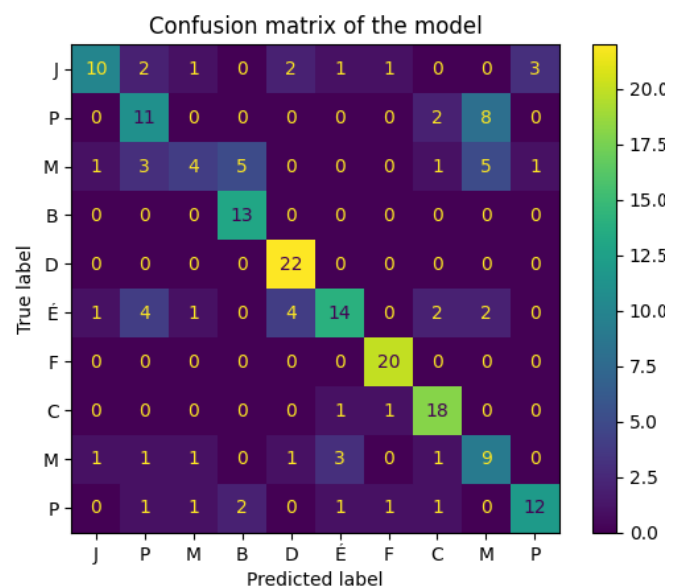
1. **Symétrie horizontale aléatoire (RandomFlip)** : Simule une inversion de l'image pour augmenter la diversité.
2. **Rotation aléatoire (RandomRotation)** : Ajoute une variation dans l'orientation des objets pour améliorer la robustesse.

### - Impact :

- L'augmentation des données permet de compenser la taille limitée de la base d'entraînement en fournissant des exemples supplémentaires.
- Cela améliore la généralisation et réduit les erreurs sur les données de test.



28



## E. Stratégies pour limiter l'overfitting

Pour réduire l'overfitting, plusieurs approches sont envisageables :

### 1. Dropout :

- Non utilisé ici, mais il pourrait être intégré (par ex., après chaque couche dense) pour désactiver aléatoirement des neurones pendant l'entraînement, réduisant ainsi la dépendance à certaines connexions.

### 2. Régularisation L2 :

- Une pénalisation des poids excessifs pourrait être ajoutée aux couches denses pour limiter la complexité du modèle.

### 3. Data Augmentation :

- Comme mentionné précédemment, elle contribue également à réduire l'overfitting en exposant le modèle à une plus grande variété d'exemples.

### 4. Split validation :

- Une validation croisée sur 20% des données a été utilisée pour surveiller la performance sur un ensemble indépendant.

### 5. Optimisation du gradient :

- Le choix de l'optimiseur Adam est efficace, mais l'ajout de techniques comme *gradient clipping* pourrait stabiliser davantage l'entraînement.

## F. Transfert Learning

### - Avantages :

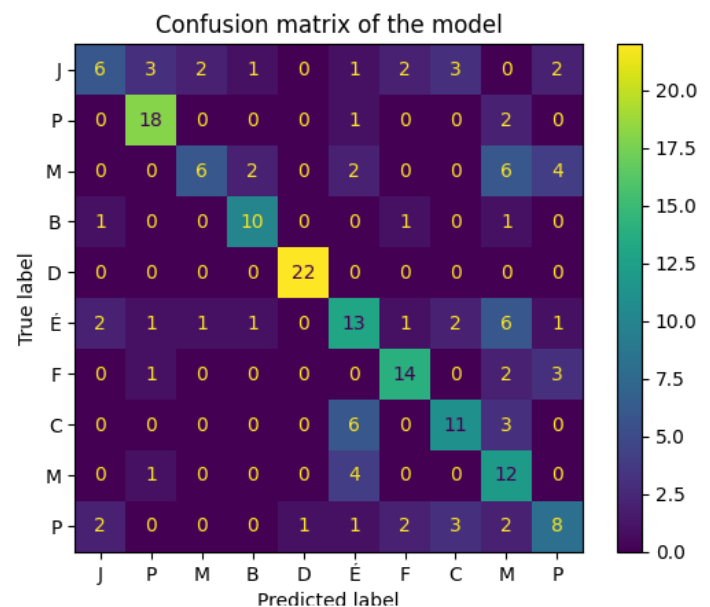
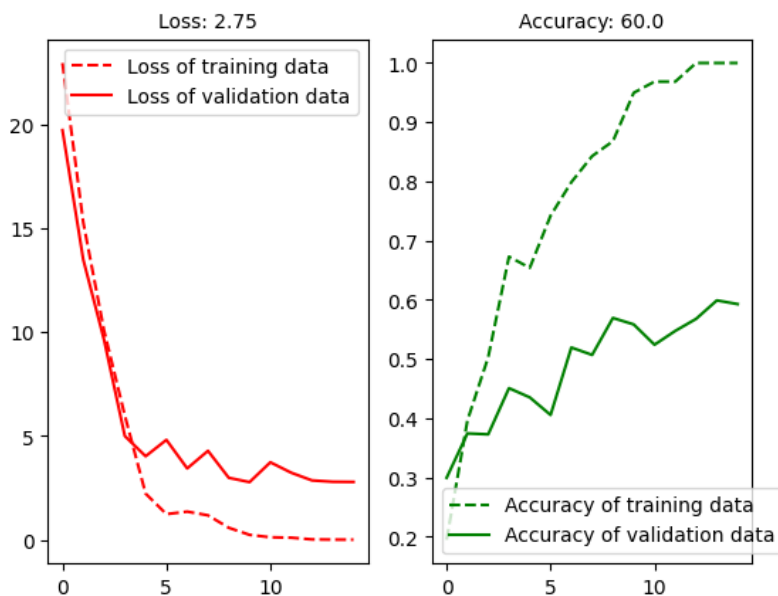
- Le modèle Xception pré-entraîné sur ImageNet permet une excellente base pour la classification. Les couches profondes capturent des caractéristiques complexes, tandis que les couches finales s'adaptent à la tâche spécifique.

### - Performance :

- Ce modèle atteint des performances supérieures, notamment grâce à ses représentations bien structurées. Cependant, il nécessite des ressources GPU importantes.

### - Améliorations possibles :

- Ajuster les couches gelées : En débloquant les dernières couches convolutives, le modèle pourrait apprendre des caractéristiques encore plus spécifiques.
- Augmenter les epochs avec des techniques comme *fine-tuning*.



## **Conclusion :**

Ce projet nous aura permis d'explorer en profondeur les principaux algorithmes d'apprentissage supervisé, depuis des modèles simples jusqu'à des architectures avancées. Nous aurons ainsi pu mettre en évidence leurs forces, leurs limites et leurs performances dans des contextes variés, tout en analysant l'impact des paramètres et des stratégies d'apprentissage.

Nous aurons démontré la capacité du Perceptron à résoudre des problèmes linéairement séparables et la nécessité d'approches plus complexes, comme le réseau multicouche, pour des cas non linéaires comme le XOR. Les expérimentations avec l'apprentissage selon Widrow auront souligné l'importance des initialisations et des critères de convergence, tandis que les travaux sur la classification Full-Connected et Deep Learning auront mis en lumière l'intérêt de techniques avancées comme la Data Augmentation ou le transfert de connaissances pour améliorer les performances.

Au-delà des résultats obtenus, ce projet aura également permis de renforcer notre compréhension des concepts fondamentaux de l'apprentissage automatique et d'approfondir nos compétences dans le développement, l'analyse et l'évaluation des modèles. Ces acquis nous serviront de base pour aborder des problématiques plus complexes à l'avenir, en tirant parti des enseignements et des expériences réalisés ici.