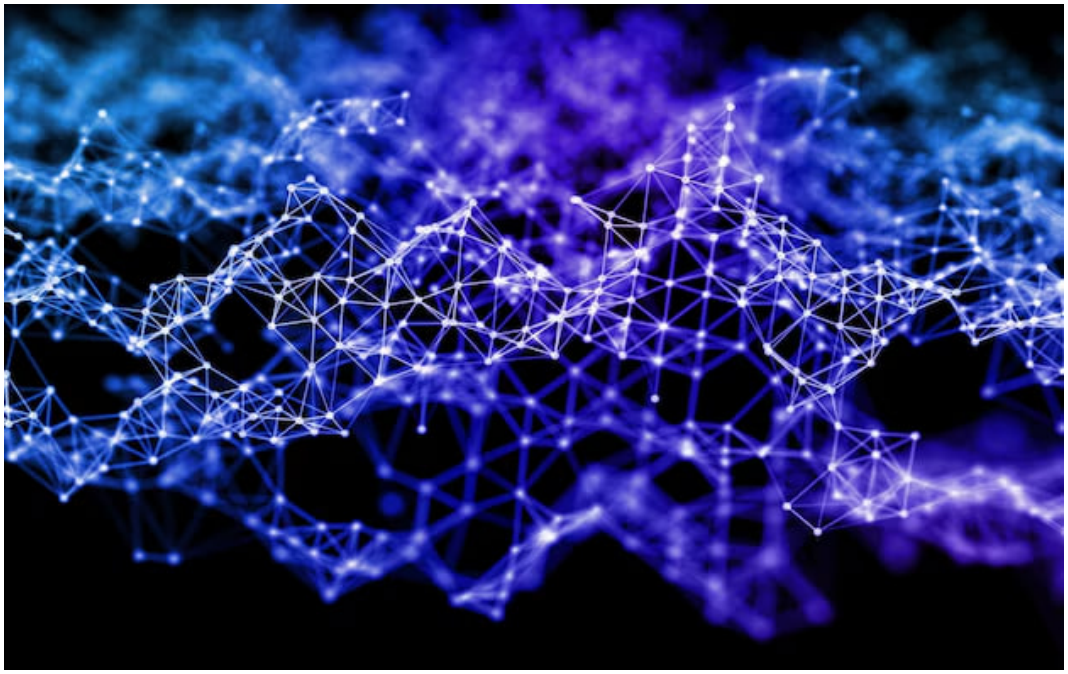
Université de Poitiers 2024-2025 MASTER 2 STDV

Projet De Machine Learning

Réalisé par :

Kodzo LIMA :
kodzo.lima@etu.univ-poitiers.fr
Taha HAMDAOUI :
taha.hamdaoui@etu.univ-poitiers.fr

Les réseaux de neurones



Chargés de l'UE

M.Philippe CARRE
M. Ayoud SEDDOUKI

Table des matières

1 Introduction

La classification et la reconnaissance d'images sont aujourd'hui au cœur de nombreuses applications en intelligence artificielle, qu'il s'agisse de la reconnaissance faciale, de la détection d'objets ou du tri automatisé. Ce projet se propose d'explorer plusieurs méthodes de discrimination d'images, en combinant des approches fondées sur les descripteurs d'image et des techniques de réseaux de neurones, allant des perceptrons simples jusqu'aux modèles d'apprentissage profond.
Nous commencerons par l'implémentation d'un perceptron simple, en intégrant un algorithme d'apprentissage de type Widrow-Hoff pour ajuster les poids en fonction des données d'entraînement. Par la suite, le projet s'étendra à un perceptron multicouche et à l'apprentissage multicouche, permettant de traiter des problèmes plus complexes grâce à une architecture plus profonde.
La deuxième partie qui consite à réaliser un système discrimination d'images réelles sera divisée en deux catégories principales : l'approche par descripteurs et l'approche "Deep" basée sur les données. Dans l'approche par descripteurs, nous calculerons des caractéristiques spécifiques pour chaque image, qui alimenteront ensuite un système de discrimination. Dans l'approche "Deep", nous utiliserons un réseau de neurones entièrement connecté et des techniques d'apprentissage profond pour extraire automatiquement des informations et discriminer les images.

2 Développement d'un perceptron

2.1 Mise en place d'un perceptron simple

2.1.1 La stratégie utilisée pour développer le perceptron simple

La fonction est structurée pour être facilement réutilisable avec trois paramètres (x, w et active), facilitant la gestion de différents types de données d'entrée, de poids et de fonctions d'activation. Le calcul de la somme pondérée intègre directement un biais (w[0]), en ajoutant ce dernier au produit scalaire des poids et des entrées, ce qui simplifie le code sans variables supplémentaires. Le choix de l'activation est basé sur le paramètre active, permettant d'utiliser soit une activation binaire de type "marche" (1 if somme $>$ 0 else 0), soit une activation continue avec np.tanh. Par ailleurs, la fonction est pensée pour gérer un cas particulier où la sortie attendue pourrait être 0 au lieu de -1, ce qui est utile lorsque la convention binaire de sortie est (0, 1) au lieu de (-1, 1).

```
def perceptron_simple(x, w, active):
    somme = w[0] + np.dot(w[1:], x)

    if active == 0:
        return 1 if somme > 0 else 0
    else:
        return np.tanh(somme)
```

2.1.2 Le fonctionnement du code dans le cas du OU

Pour tester votre perceptron avec l'exemple du OU logique vu en cours (en utilisant $\phi(x) = sign(x)$), nous avons créé une fonction *test_perceptron()* dans laquelle le vecteur $w = np.array([-0.5, 1, 1])$ représente les poids du perceptron, avec un biais de -0.5, et des poids de 1 pour chaque entrée. Ces valeurs sont choisies pour que le perceptron apprenne la fonction OU logique ensuite inputs contient les quatre combinaisons possibles de valeurs binaires en entrée et enfin une boucle de test qui parcourt chaque entrée x et utilise *perceptron_simple(x, w, 0)*, avec *active* = 0, pour appliquer une fonction de seuil (activation sign(x)).

```
def test_perceptron_ou_logique():
    w = np.array([-0.5, 1, 1])
    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    print("Test du perceptron pour l'OU logique :")
    for i, x in enumerate(inputs):
        result = perceptron_simple(x, w, 0)
        print(f"Entrée : {x}, Sortie obtenue : {result}")

    return w
w = test_perceptron_ou_logique()
```

Résultats du perceptron pour le OU logique :

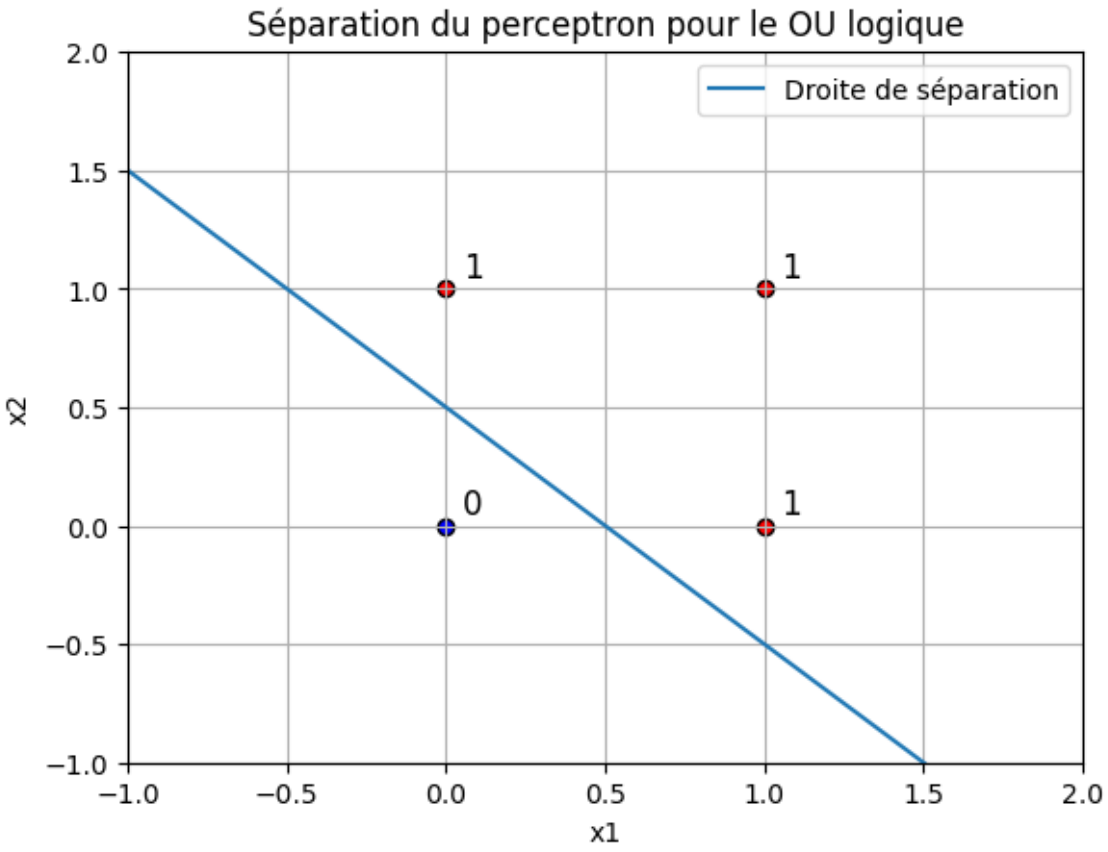
```
Test du perceptron pour l'OU logique :
Entrée : [0 0], Sortie obtenue : 0
Entrée : [0 1], Sortie obtenue : 1
Entrée : [1 0], Sortie obtenue : 1
Entrée : [1 1], Sortie obtenue : 1
```

Conclusion : Nous obtenons bien les résultats attendus pour chaque entrée ce qui jutifie que ce code marche bien dans le cas OU.

2.1.3 Affichage de la figure individu et frontière

Ce code ci-dessous définit les poids du perceptron pour implémenter la logique OU et trace la frontière de décision calculée par l'équation de la droite de séparation. Les points sont colorés en bleu pour la sortie 0 et en rouge pour la sortie 1, montrant ainsi la capacité du perceptron à distinguer les classes du OU logique.

```
def tracer_droite_perceptron(w):
    a = -w[1] / w[2]
    b = -w[0] / w[2]

    t = np.array([-3, 3])
    z = a * t + b

    plt.plot(t, z, label="Droite de séparation")

    inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    outputs = [0, 1, 1, 1]  # Sorties de la fonction OR logique
    for (x, y), output in zip(inputs, outputs):
        plt.scatter(x, y, color='red' if output == 1 else 'blue',
           edgecolors='k')
        plt.text(x + 0.05, y + 0.05, f'{output}', fontsize=12, color='
           black')

    plt.xlim(-1, 2)
    plt.ylim(-1, 2)

    plt.title("Séparation du perceptron pour le OU logique")
    plt.xlabel("x1")
    plt.ylabel("x2")
    plt.grid(True)
    plt.legend()
    plt.show()

tracer_droite_perceptron(w)
```



Commentaire :

*** Poids du perceptron :** $w = np.array([-0.5, 1, 1])$ définit le biais et les poids de l'entrée. Ces valeurs permettent d'implémenter la logique OU.

*** Ligne de séparation :** La droite de séparation est calculée à partir de l'équation $x_2 = -(w[0] + w[1] * x_1)/w[2]$, qui représente la frontière de décision pour ce perceptron.

*** Couleurs et étiquettes :** Les points avec une sortie 0 sont affichés en bleu, ceux avec une sortie 1 en rouge.

2.2 Apprentissage Widrow : Ensemble Test 1 / Test 2

2.2.1 La stratégie utilisée pour développer l'apprentissage Widrow.

Pour mettre en place en place l'apprentissage widrow ,nous avons commencer par initialiser aléatoirement Les poids, y compris un biais puis ajustés au fil des époques en utilisant une descente de gradient. À chaque époque, les données sont traitées par lots (définis par la taille *batch_size*) : pour chaque lot, le modèle prédit la sortie, calcule l'erreur par rapport à la sortie attendue, puis met à jour les poids afin de minimiser l'erreur quadratique moyenne. Le code suit l'évolution de cette erreur pour chaque époque, permettant de visualiser la convergence des poids vers une solution optimale. Ci-dessous le code utilisé

```
    def apprentissage_widrow(x, yd, epoch, batch_size, learning_rate
        =0.01):
    n_features, n_samples = x.shape
    w = np.random.randn(n_features + 1)
    erreurs = []

    x = np.vstack((np.ones((1, n_samples)), x))

    for ep in range(epoch):
        erreur_epoch = 0

        for i in range(0, n_samples, batch_size):
            x_batch = x[:, i:i + batch_size]
            yd_batch = yd[i:i + batch_size]

            y_pred = np.dot(w, x_batch)

            erreur = yd_batch - y_pred

            w += learning_rate * np.dot(erreur, x_batch.T)

            erreur_epoch += np.sum(erreur ** 2)

        erreurs.append(erreur_epoch / n_samples)

    return w, erreurs
```

*** Pour s'assurer que déjà le code fonctionne bien nous avons stimulé un exemple pour tester.**

```
x = np.array([[0.5, 1.0, -1.5], [1.0, -1.0, 0.0]])
yd = np.array([1, -1, 1])
epoch = 100
batch_size = 1

w, erreur = apprentissage_widrow(x, yd, epoch, batch_size)

print("Poids après apprentissage :", w)
print("Erreur au fil des itérations :", erreur)
```

*** Nous obtenons les résultats suivants pour les variables poids(w) et erreur. Ce qui nous rassure au premier à bord que le code marche**

```
Poids après apprentissage : [ 0.28132897 -0.5558805   0.68212251]

Erreur au fil des itérations : [6.884737538397487, 6.455039097936531, 6.053449442451647,
5.678073535263127, 5.327146514209319, 4.99902464509015, 4.692176908473807,
```

```
4.405177175305952, 4.136696929906693, 3.8854985018636596, 3.6504287710458674,
3.430413312487121,3.2244509502333596, 3.0316086914279636, 2.851017013934564,
2.681865482679236, 2.523398671643164,2.3749123700624435, 2.2357500529024383,
2.105299597077981,1.9829902261955663,1.8682896678061927,1.760701508284539,
1.6597627314974914, 1.5650414283985086, 1.476134665589021, 1.3926665017289268,
1.3142861414598115, 1.2406662172309106, 1.1715011900930055, 1.1065058611530183,
1.0454139859653737, 0.9879769846783901, 0.9339627412579709, 0.8831544855793191,
0.8353497526128884, 0.7903594133355902, 0.748006772374572, 0.7081267277407041,
0.6705649883341, 0.6351773452063166, 0.6018289928449158, 0.5703938970073678,
0.5407542058741734, 0.5127997015169273, 0.4864272888870131, 0.46154051972583726,
0.43804914897901176, 0.4158687214656624, 0.39492018671095247, 0.3751295399958218,
0.3564274878136012, 0.33874913604932927, 0.32203369931489817, 0.30622422998225185,
0.2912673655583004, 0.277113093139554, 0.2637145297722224, 0.25102771762510573,
0.23901143295850336, 0.22762700794292978, 0.2168381644470797, 0.20661085897553436,
0.19691313799348867, 0.18771500292860674, 0.17898828418924348, 0.17070652358397864,
0.16284486456992922, 0.15537994979685632, 0.14828982545088334, 0.14155385193587383,
0.13515262046236654, 0.12906787514360193, 0.12328244022574043, 0.11778015210503312,
0.11254579580857127, 0.10756504563746, 0.10282440969192709, 0.0983111780171186,
0.09401337412623252, 0.08991970967430556, 0.0860195420714725, 0.08230283483895441,
0.07876012052446364, 0.07538246600622082, 0.07216144002642039, 0.06908908280581758,
0.0661578776011999, 0.06336072407689824, 0.06069091337024049, 0.0581421047389906,
0.05570830368640282, 0.0533838414665777, 0.05116335587938602, 0.04904177327034781,
0.04701429165655927,0.04507636490507306, 0.04322368789508446, 0.04145218259988748,
0.039757985028858585]
```

- **Pour l'ensemble Test 1**

  Indiquez graphiquement quelques étapes de l'apprentissage (ensemble et droite de séparation)

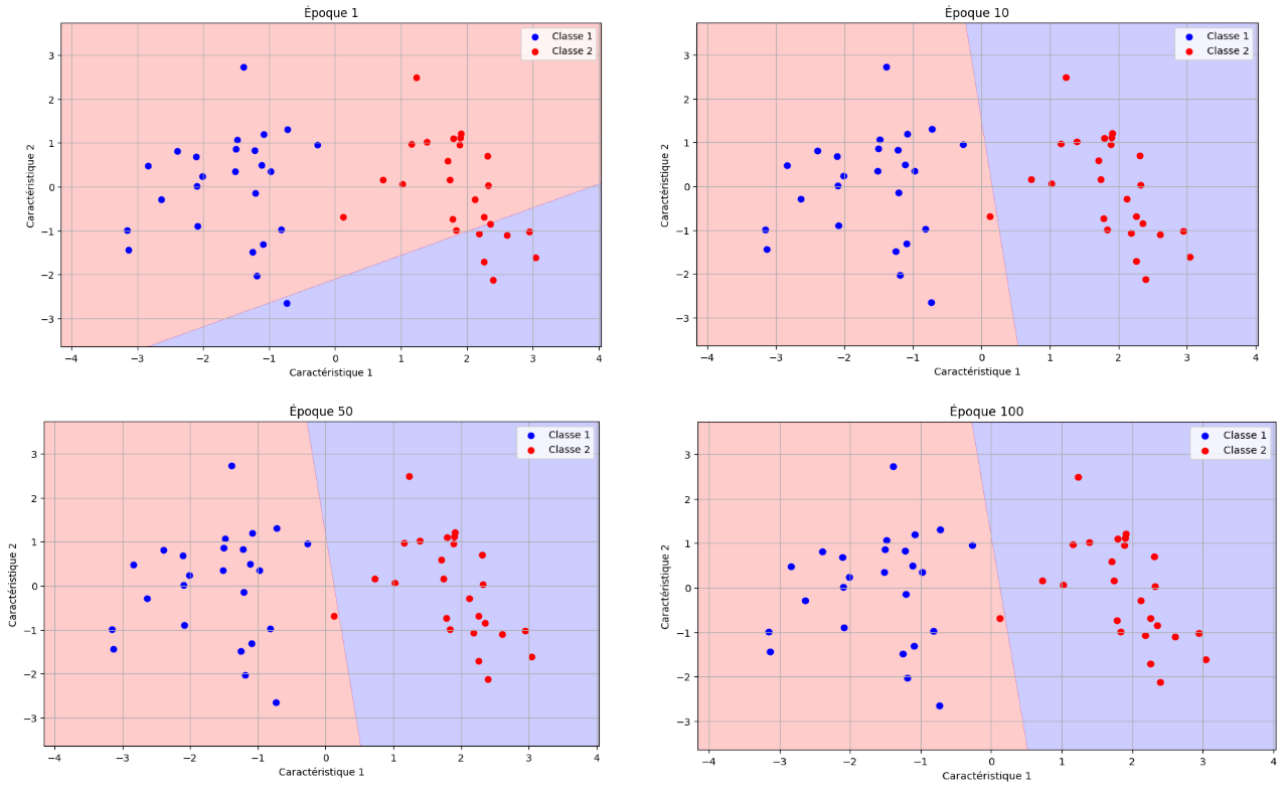
  Pour afficher graphiquement l'évolution de la frontière de décision pendant l'apprentissage, nous avons besoin de modifier le code de la fonction d'aprentissage pour enregistrer les poids w à certains intervalles d'époques et tracer la frontière de décision pour chaque étape enregistrée.Nous avons choisi de tracer des graphiques pour les époques 1 à 100 (l'état initial et la fin de l'entrainenement)dans le but de visualiser l'évolution de l'apprentissage à différentes étapes. Ici nous allons affiché que 4 étapes 1 ,10,50 et 100 puisque à partir de 10 nous obtenons le même graphique.

```
def apprentissage_widrow(x, yd, epoch, batch_size, learning_
    rate=0.01, display_epochs=None):
     n_features, n_samples = x.shape
     w = np.random.randn(n_features + 1)
     erreurs = []

     x = np.vstack((np.ones((1, n_samples)), x))

     if display_epochs is None:
         display_epochs = []
```

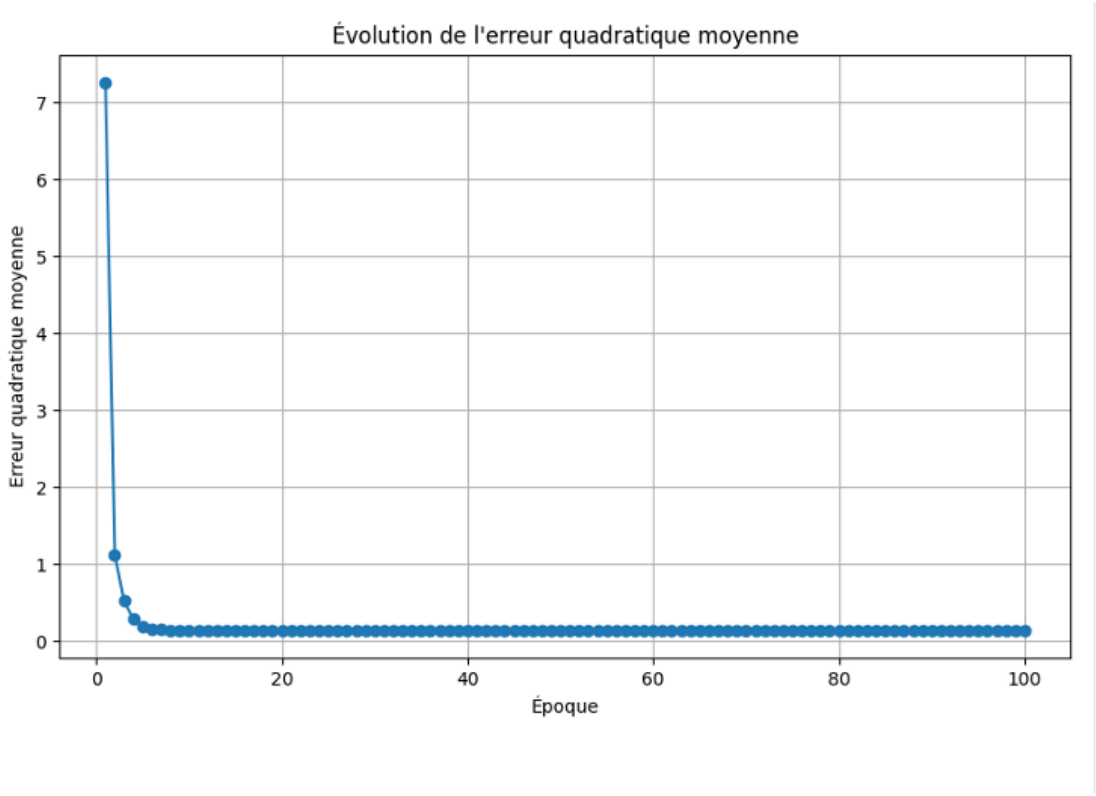
```
    for ep in range(epoch):
        erreur_epoch = 0

        for i in range(0, n_samples, batch_size):
            x_batch = x[:, i:i + batch_size]
            yd_batch = yd[i:i + batch_size]

            y_pred = np.dot(w, x_batch)

            erreur = yd_batch - y_pred

            w += learning_rate * np.dot(erreur, x_batch.T)

            erreur_epoch += np.sum(erreur ** 2)

        erreurs.append(erreur_epoch / n_samples)

        if ep + 1 in display_epochs:
            print(f" poque  {ep+1}:")
            tracer_frontiere_decision(x[1:], yd, w, f" poque
               {ep+1}")

    return w, erreurs

def tracer_frontiere_decision(x, yd, w, title="Frontière de dé
   cision"):
    x_min, x_max = x[0, :].min() - 1, x[0, :].max() + 1
    y_min, y_max = x[1, :].min() - 1, x[1, :].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200),
                         np.linspace(y_min, y_max, 200))

    grid = np.c_[xx.ravel(), yy.ravel()]
    grid = np.vstack((np.ones((1, grid.shape[0])), grid.T))

    Z = np.dot(w, grid)
    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(10, 6))
    plt.contourf(xx, yy, Z, levels=[Z.min(), 0, Z.max()],
       alpha=0.2, colors=['blue', 'red'])
    plt.scatter(x[0, :25], x[1, :25], color='blue', label='
       Classe 1')
    plt.scatter(x[0, 25:], x[1, 25:], color='red', label='
       Classe 2')
    plt.title(title)
    plt.xlabel("Caractéristique 1")
    plt.ylabel("Caractéristique 2")
    plt.legend()
    plt.grid(True)
    plt.show()
```

```
Data = np.loadtxt("p2_d1.txt")
print("Shape de Data :", Data.shape)
x = Data

yd = np.array([1]*25 + [-1]*25)

epoch = 100
batch_size = 1
learning_rate = 0.01

w, erreurs = apprentissage_widrow(x, yd, epoch, batch_size,
    learning_rate, display_epochs=[1, 10,50,100])

plt.figure(figsize=(10, 6))
plt.plot(range(1, epoch+1), erreurs, marker='o')
plt.title(" volution  de l'erreur quadratique moyenne")
plt.xlabel(" poque ")
plt.ylabel("Erreur quadratique moyenne")
plt.grid(True)
plt.show()
```

Nous obtenons le resultat suivant qui est un graphique montrant deux régions en couleur bleu et rouge séparées donc par une droite sépateur(frontières de décision) qui montrent les changements au fil des époques d'apprentissage.



– **Représentation de l'erreur en fonction des itérations**



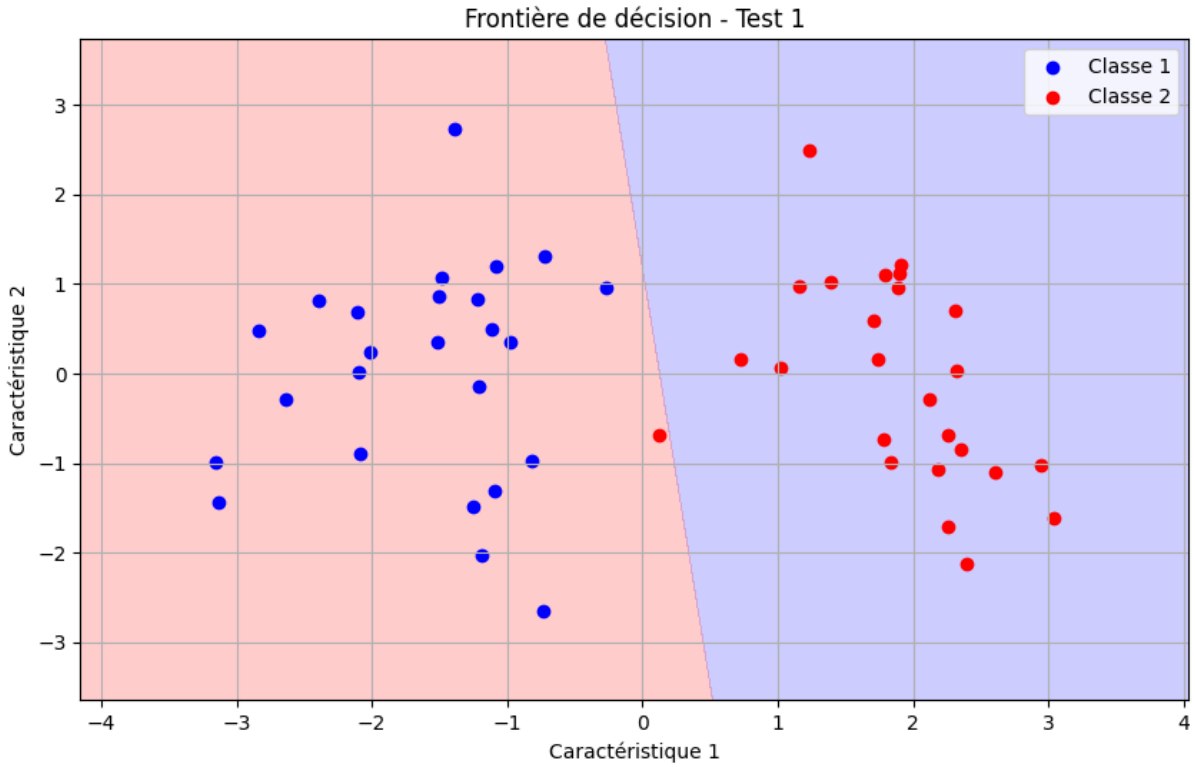
– **Quelques tests avec des initialisations différentes, commenter le résultat après convergence.**

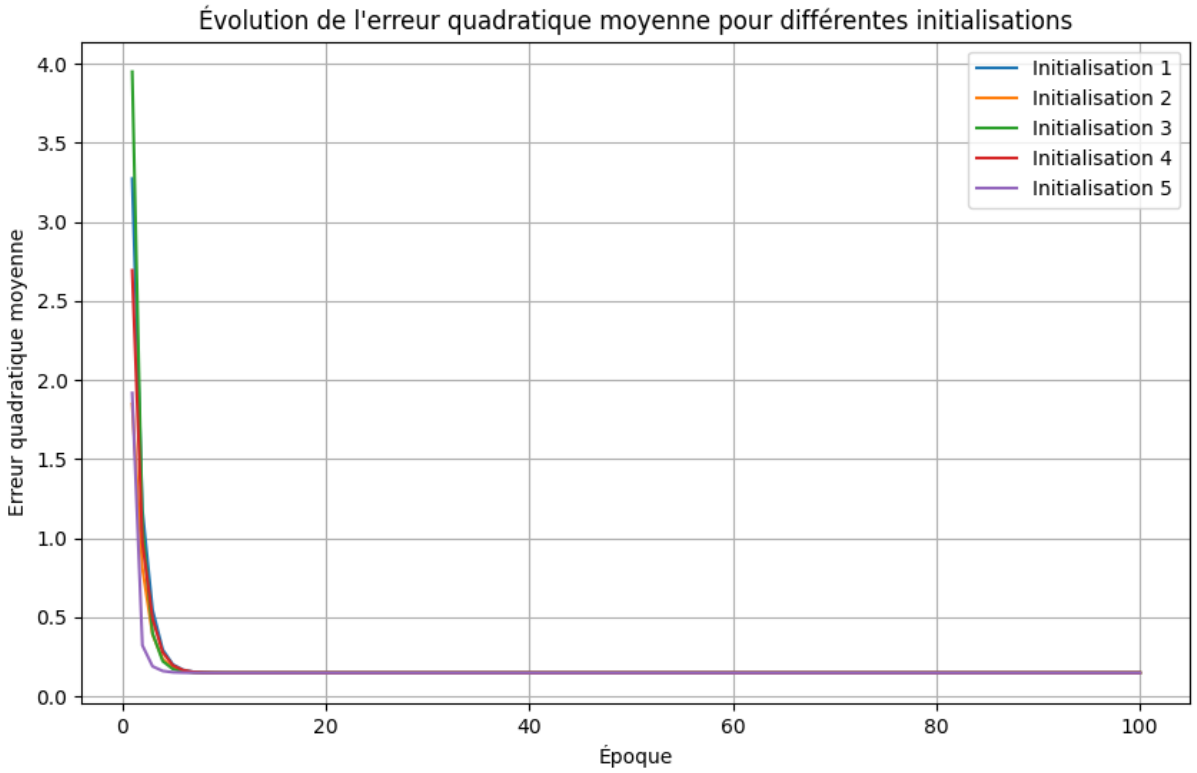
Dans cette partie nous avons procéder à une modification du code précédent pour exécuter l'apprentissage plusieurs fois avec différentes initialisations des poids, permettant ainsi d'explorer l'impact de ces initialisations sur la convergence du modèle. Il trace la frontière de décision après chaque apprentissage et affiche plusieurs courbes d'évolution de l'erreur pour chaque initialisation.

Voici les parties modifiées pour tester avec 5 initialisations différentes ensuite changer l'initialisation des poids à chaque fois, visualiser la frontière pour la dernière itération de chaque testet enfin tracer l'évolution de l'erreur quadratique moyenne pour chaque initialisation

```
results = []
for i in range(5):
    np.random.seed(i)
    w, erreurs = apprentissage_widrow(x, yd, epoch, batch_size
        , learning_rate)
    results.append((w, erreurs))

    tracer_frontiere_decision(x, yd, w, title=f"Frontière de d
        écision - Test {i+1}")
```

```
plt.figure(figsize=(10, 6))
for i, (_, erreurs) in enumerate(results):
    plt.plot(range(1, epoch + 1), erreurs, label=f"
       Initialisation {i+1}")
plt.title("  volution  de l'erreur quadratique moyenne pour
   différentes initialisations")
plt.xlabel("  poque  ")
plt.ylabel("Erreur quadratique moyenne")
plt.legend()
plt.grid(True)
plt.show()
```

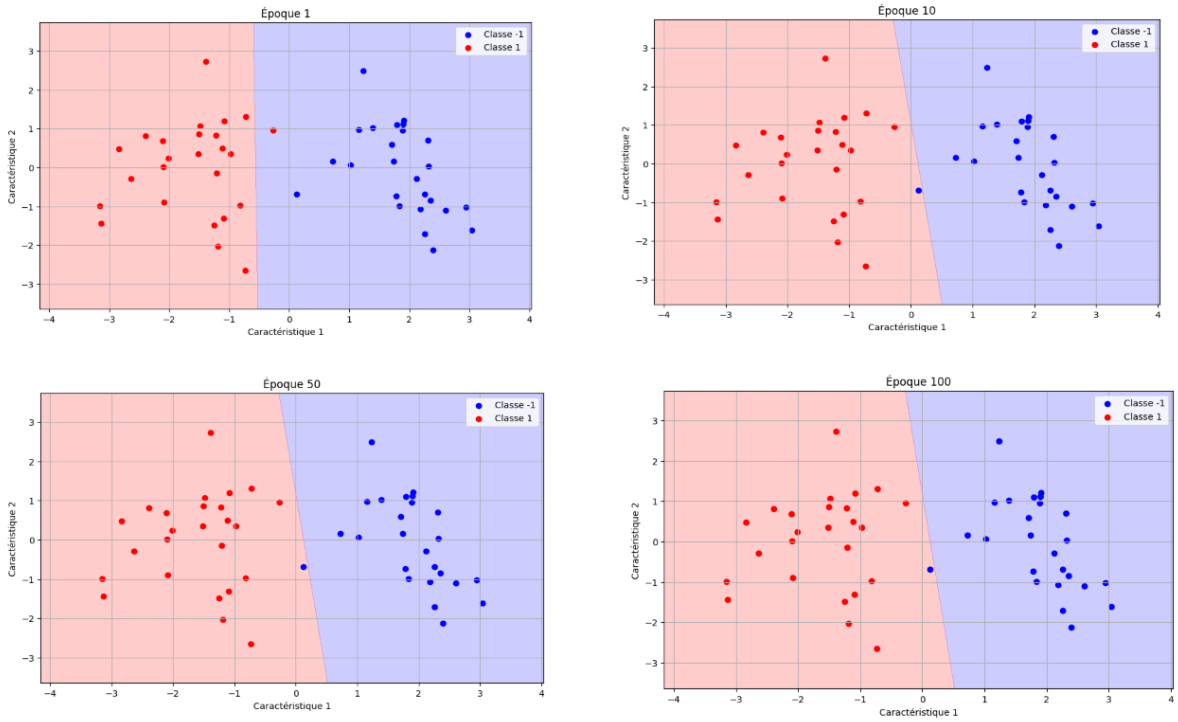




Conclusion : Nous obtenons pour les cinq (5) initialiseurs le même graphe pour la frontière de décision et pour l'évolution des erreurs quadratiques moyennes,après la convergence, on observe sur le graphique que l'erreur pour toutes les initialisations atteint une valeur très faible et stable. Cela indique que le processus d'apprentissage a bien minimisé l'erreur, obtenant un bon ajustement quel que soit le point de départ des poids initiaux. La convergence de toutes les courbes d'initialisation vers une erreur faible similaire montre que le processus d'entraînement du modèle est robuste face aux différentes valeurs de départ, conduisant à une solution finale cohérente.

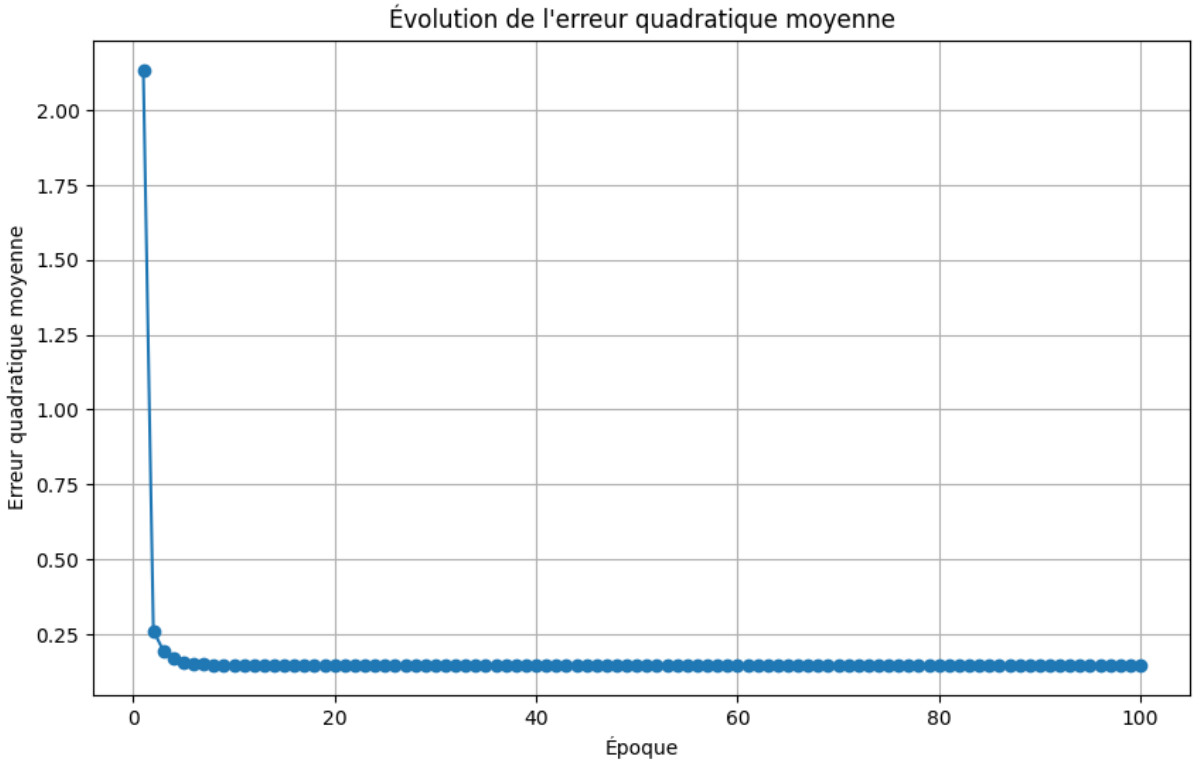
– **Pour l'ensemble Test 2**

– **Indiquez graphiquement quelques étapes de l'apprentissage (ensemble et droite de séparation)**
En appliquand les mêmes principes que pour le test 1 nous obtenenons e resultat suivant



– **Représentation de l'erreur en fonction des itérations**

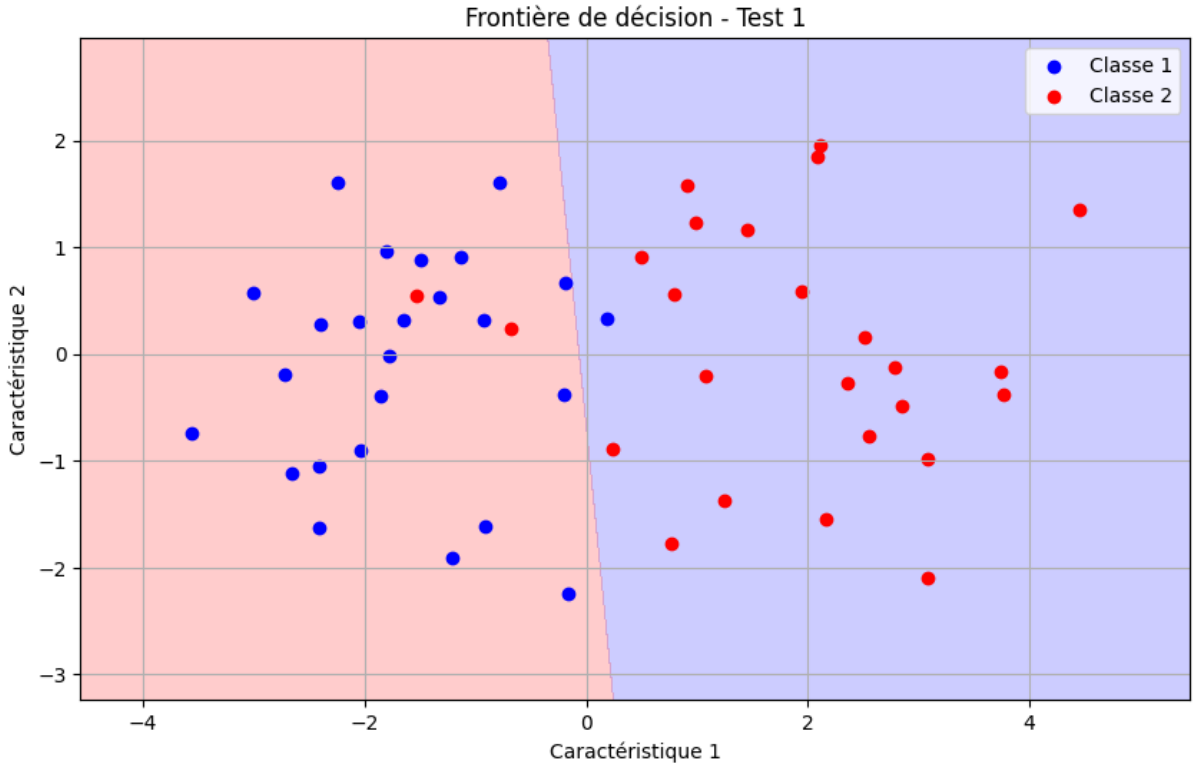
De même que dans le test 1 l'erreur est obtenu par la même manière dans le test 2

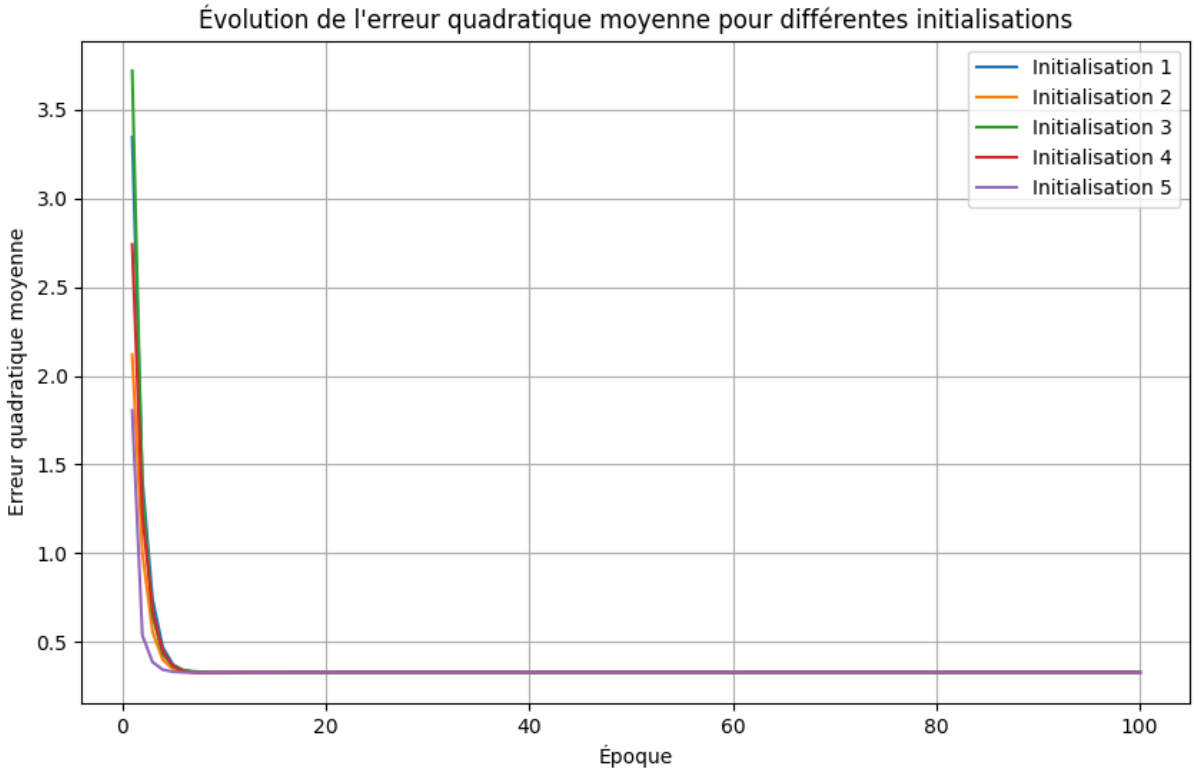


Comparaison :
Dans le premier test (test 1), l'algorithme converge vers une erreur plus faible, ce qui peut indiquer un meilleur ajustement final. En revanche, le deuxième cas (test 2) montre une convergence vers une valeur plus élevée, autour de 0,25, signifiant que l'apprentissage n'a pas réussi à atteindre un niveau aussi bas que dans le premier graphique.
En résumé, le premier graphique montre une convergence plus efficace, tandis que le second semble atteindre une limite d'erreur plus élevée

– **Faites quelques tests avec des initialisations différentes, commenter le résultat après convergence.**

Nous Obtenons pour les cinq (5) initialisations les courbes d'évolutions de l'erreur suivantes





Commentaire :
On observe que toutes les initialisations présentent une diminution rapide de l'erreur au début, indiquant que le modèle réduit rapidement l'erreur lors des premières époques. Après environ 20 époques, toutes les initialisations convergent vers une valeur d'Erreur basse et similaire, ce qui suggère que, quelle que soit l'initialisation

des paramètres, le modèle atteint une solution similaire en termes de minimisation de l'erreur. Une fois la convergence atteinte, l'Erreur reste pratiquement constante pour toutes les initialisations, indiquant une performance stable du modèle. En résumé, ce graphique montre que l'initialisation des paramètres n'a pas d'impact significatif sur la solution finale, bien que cela puisse affecter la vitesse de convergence dans les premières époques.

2.3 Mise en place d'un perceptron Multicouche

2.4 Eléments de commentaires sur la stratégie que vous avez utilisée pour développer le perceptron Multicouche.

Dans ce code, la stratégie de conception du perceptron multicouche repose sur une structuration claire et modulaire des étapes du calcul. La fonction d'activation utilisée est la sigmoïde logistique ($\phi(x) = 1/1+\exp x$), qui introduit une non-linéarité essentielle pour permettre au réseau d'apprendre des fonctions complexes. Le réseau est organisé en deux couches : la première (couche cachée) calcule les activations intermédiaires (z_1 et z_2) à l'aide des poids de la matrice w1, et la deuxième (couche de sortie) produit la sortie finale y en combinant les activations cachées avec les poids de w_2. L'approche est mathématiquement rigoureuse : chaque neurone inclut un biais représenté par la première colonne de poids dans les matrices (w1[*,0] et w2[0]), ce qui améliore la flexibilité du modèle.
L'algorithme calcule successivement les activations de chaque couche, en respectant l'ordre des opérations dans un réseau multicouche : d'abord la propagation vers l'avant dans la couche cachée, puis dans la couche de sortie. Cette stratégie modulaire facilite l'extension future du modèle, comme l'ajout de couches supplémentaires ou l'adaptation à des problèmes avec plus de neurones ou de dimensions d'entrée.

2.5 Le résultat numérique et par calcul pour le test demandé

*** PAR CALCUL**

Données initiales

- **Entrée :** $x = [1, 1]^T$
- **Poids de la couche cachée** ($W1$) **:**

$$W1 = \begin{bmatrix} -0.5 & 2 & -1 \\ 0.5 & 1 & 0.5 \end{bmatrix}$$

  - Neurone 1 : biais $= -0.5$, poids $= 2$ pour x_1, -1 pour x_2,
  - Neurone 2 : biais $= 0.5$, poids $= 1$ pour x_1, 0.5 pour x_2.

- **Poids de la couche de sortie** ($W2$) **:**

$$W2 = [2, -1, 1]$$

– biais = 2, poids de $z_1 = -1$, poids de $z_2 = 1$.

- **Fonction d'activation :**

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

Étape 1 : Calcul des activations des deux neurones de la couche cachée

Neurone 1 :

$$\begin{aligned} z_1 &= \text{biais} + x_1 \cdot \text{poids}_1 + x_2 \cdot \text{poids}_2 \\ z_1 &= -0.5 + 1 \cdot 2 + 1 \cdot (-1) \\ z_1 &= -0.5 + 2 - 1 = 0.5 \end{aligned}$$

Application de la sigmoïde :

$$h_1 = \varphi(z_1) = \frac{1}{1 + e^{-0.5}} \approx 0.622$$

Neurone 2 :

$$\begin{aligned} z_2 &= \text{biais} + x_1 \cdot \text{poids}_1 + x_2 \cdot \text{poids}_2 \\ z_2 &= 0.5 + 1 \cdot 1 + 1 \cdot 0.5 \\ z_2 &= 0.5 + 1 + 0.5 = 2 \end{aligned}$$

Application de la sigmoïde :

$$h_2 = \varphi(z_2) = \frac{1}{1 + e^{-2}} \approx 0.881$$

Étape 2 : Calcul de la sortie finale

Les sorties des neurones de la couche cachée sont :

$$h = [h_1, h_2] = [0.622, 0.881]$$

La couche de sortie prend en compte un biais de 2, un poids de -1 pour h_1, et un poids de 1 pour h_2.

Produit scalaire avec les poids :

$$\begin{aligned} y_{\text{entrée}} &= \text{biais} + h_1 \cdot \text{poids}_1 + h_2 \cdot \text{poids}_2 \\ y_{\text{entrée}} &= 2 + 0.622 \cdot (-1) + 0.881 \cdot 1 \\ y_{\text{entrée}} &= 2 - 0.622 + 0.881 \\ y_{\text{entrée}} &\approx 2 + 0.259 = 2.259 \end{aligned}$$

Application de la sigmoïde :

$$y = \varphi(y_{\text{entrée}}) = \frac{1}{1 + e^{-2.259}} \approx 0.905$$

Résultat final

La sortie finale est :

$$y \approx 0.905$$

*** NUMERIQUE**

```
                    def phi(x):
    return 1 / (1 + np.exp(-x))

def multiperceptron(x, w1, w2):
    z1 = phi(x[0] * w1[0, 1] + x[1] * w1[0, 2] + w1[0, 0])
    z2 = phi(x[0] * w1[1, 1] + x[1] * w1[1, 2] + w1[1, 0])
    y = phi(z1 * w2[1] + z2 * w2[2] + w2[0])
    return y

w1 = np.array([[-0.5, 2, -1], [0.5, 1, 0.5]])
w2 = np.array([2, -1, 1])
x = np.array([1, 1])
y = multiperceptron(x, w1, w2)

print("Sortie du perceptron multicouche :", y)
```

Le résultat obtenu numérique du perceptron multicouche est le suivant : 0.9053673095402572

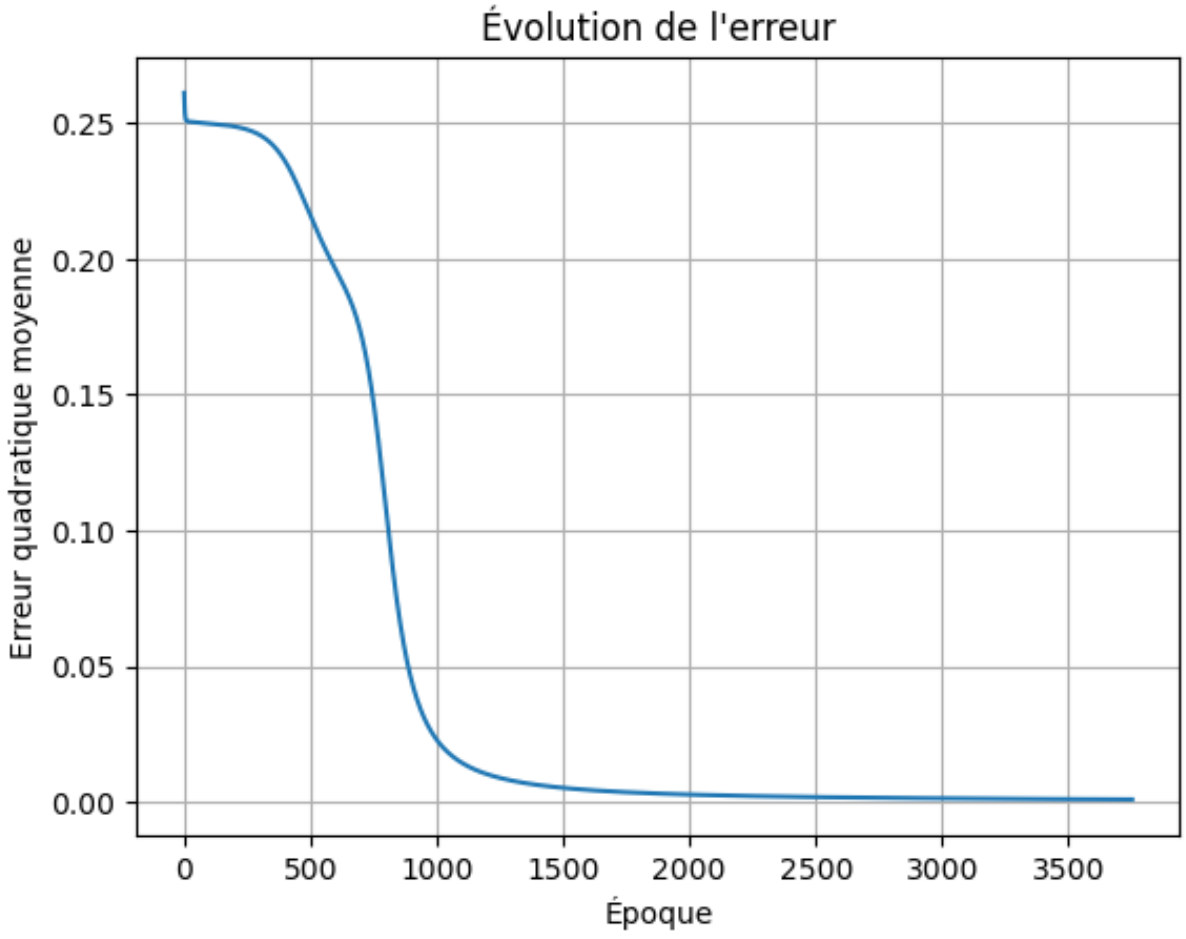
3 Apprentissage Multicouches

3.1 Représentation de l'erreur en fonction des itérations

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    return x * (1 - x)

def multiperceptron_widrow(x, yd, Epoch, Batch_size, learning_rate
   =0.5, error_threshold=1e-3):
    if x.shape[0] != 2:
        raise ValueError("La matrice d'entrée x doit avoir 2 lignes.")
    if yd.shape[0] != 1 or yd.shape[1] != x.shape[1]:
        raise ValueError("Le vecteur yd doit avoir une ligne et le mê
           me nombre de colonnes que x.")

    np.random.seed(42)
    w1 = np.random.uniform(-1, 1, (3, 2))
    w2 = np.random.uniform(-1, 1, (3, 1))
    N = x.shape[1]
```

```
    errors = []
    for epoch in range(Epoch):

        total_error = 0
        for i in range(0, N, Batch_size):
            x_batch = x[:, i:i + Batch_size]
            yd_batch = yd[:, i:i + Batch_size]

            x_bias = np.vstack((np.ones((1, x_batch.shape[1])), x_
               batch))
            z1 = sigmoid(np.dot(w1.T, x_bias))
            z1_bias = np.vstack((np.ones((1, z1.shape[1])), z1))
            y = sigmoid(np.dot(w2.T, z1_bias))

            error = yd_batch - y
            total_error += np.sum(error ** 2)

            delta2 = error * sigmoid_derivative(y)
            grad_w2 = np.dot(z1_bias, delta2.T)
            w2 += learning_rate * grad_w2

            delta1 = np.dot(w2[1:], delta2) * sigmoid_derivative(z1)
            grad_w1 = np.dot(x_bias, delta1.T)
            w1 += learning_rate * grad_w1

        mse = total_error / N

        errors.append(mse)

        if mse < error_threshold:
            print(f"Arrêt anticipé    l'époque {epoch+1} : Erreur ({
               mse:.6f}) < Seuil ({error_threshold})")
            break

    return w1, w2, errors

def plot_error(errors):
    plt.plot(errors)
    plt.xlabel(" poque ")
    plt.ylabel("Erreur quadratique moyenne")
    plt.title("  volution   de l'erreur")
    plt.grid(True)
    plt.show()

if __name__ == "__main__":
    x = np.array([[0, 0, 1, 1], [0, 1, 0, 1]])
    yd = np.array([[0, 1, 1, 0]])  # Sorties désirées


    Epoch = 10000
```

```
Batch_size = 2
learning_rate = 0.5
error_threshold = 1e-3

w1, w2, errors = multiperceptron_widrow(x, yd, Epoch, Batch_size,
    learning_rate, error_threshold)

plot_error(errors)
```



Arrêt anticipé à l'époque 3753 : Erreur (0.001000) < Seuil (0.001)

Commentaire :

Â La Phase initiale (0 à 500 époques),l'erreur commence relativement élevée (0,25). Cela correspond à une situation où les poids synaptiques sont initialisés aléatoirement, donc les prédictions sont encore loin des valeurs attendues. La descente de gradient ajuste progressivement les poids, ce qui entraîne une diminution rapide de l'erreur. La Phase intermédiaire (500 à 1000 époques), l'erreur diminue de façon exponentielle et atteint des valeurs proches de zéro. Cela indique que le modèle apprend efficacement et que les prédictions se rapprochent des sorties désirées pour l'ensemble d'apprentissage. Â partir de la Phase finale (1000 époques et plus) l'erreur devient très faible (proche de 0). Le modèle atteint presque la convergence, et les ajus-

tements supplémentaires des poids sont minimes.La stagnation de l'erreur à ce niveau suggère que le modèle a appris à résoudre le problème du cas XOR avec succès.

3.2 Tester la structure après apprentissage et montrer que c'est bien un XOR

```
def multiperceptron(x, w1, w2):
    x_bias = np.vstack((np.ones((1, x.shape[1])), x))
    z1 = sigmoid(np.dot(w1.T, x_bias))
    z1_bias = np.vstack((np.ones((1, z1.shape[1])), z1))
    y = sigmoid(np.dot(w2.T, z1_bias))
    return y

x = np.array([[0, 1, 0, 1], [0, 0, 1, 1]])
yd = np.array([[0, 1, 1, 0]])

Epoch = 10000
Batch_size = 4
learning_rate = 0.5
w1, w2, errors = multiperceptron_widrow(x, yd, Epoch, Batch_size,
   learning_rate)

y_pred = multiperceptron(x, w1, w2)
print("Prédictions du réseau :", y_pred)
```

Commentaire :

Les prédictions doivent être très proches des valeurs suivantes si le réseau est bien formé

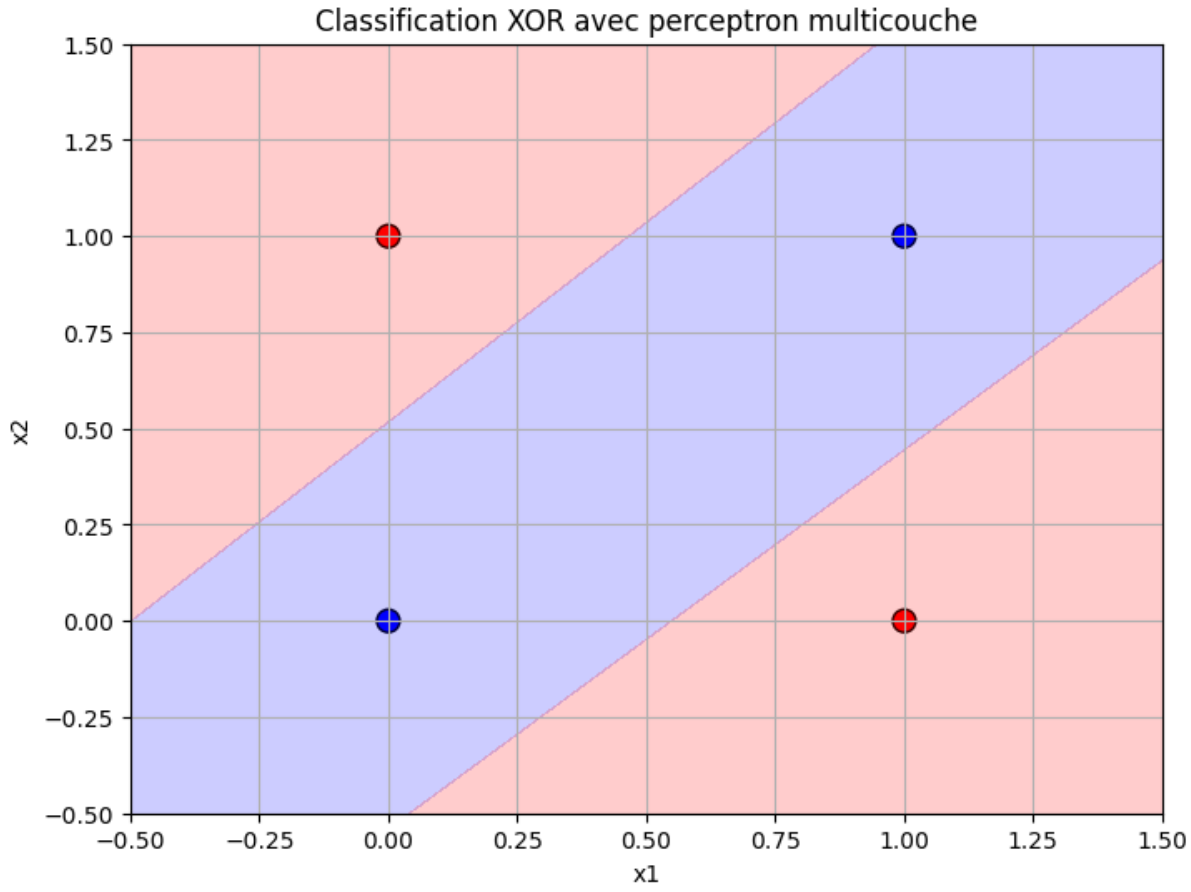
x_1	x_2	y_{attendu}
0	0	0
0	1	1
1	0	1
1	1	0

Les résultats obtenus par le code précédents sont les valeurs des Prédictions du réseau : [[0.01710283 0.98401929 0.98042081 0.01524354]].
Ces valeurs sont approximativement cohérentes avec les valeurs espérées. Ce qui signifie que c'est bien un XOR.

3.3 Représentation des trois droites séparatrices et l'ensemble d'apprentissage

En utilisation le code suivant nous allons pouvoir représenter les trois droites séparatrices et l'ensemble d'apprentissage ensuie Expliquer le fonctionnement.

```
def plot_xor_boundary(x, yd, w1, w2):
    plt.figure(figsize=(8, 6))
    colors = ['red' if label == 1 else 'blue' for label in yd.flatten
        ()]
    plt.scatter(x[0, :], x[1, :], c=colors, edgecolor='k', s=100)

    xx, yy = np.meshgrid(np.linspace(-0.5, 1.5, 100), np.linspace
        (-0.5, 1.5, 100))
    grid = np.c_[xx.ravel(), yy.ravel()].T

    z = multiperceptron(grid, w1, w2)
    zz = z.reshape(xx.shape)

    plt.contourf(xx, yy, zz, levels=[0, 0.5, 1], alpha=0.2, colors=['
        blue', 'red'])
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.title('Classification XOR avec perceptron multicouche')
    plt.grid(True)
    plt.show()

plot_xor_boundary(x, yd, w1, w2)
```



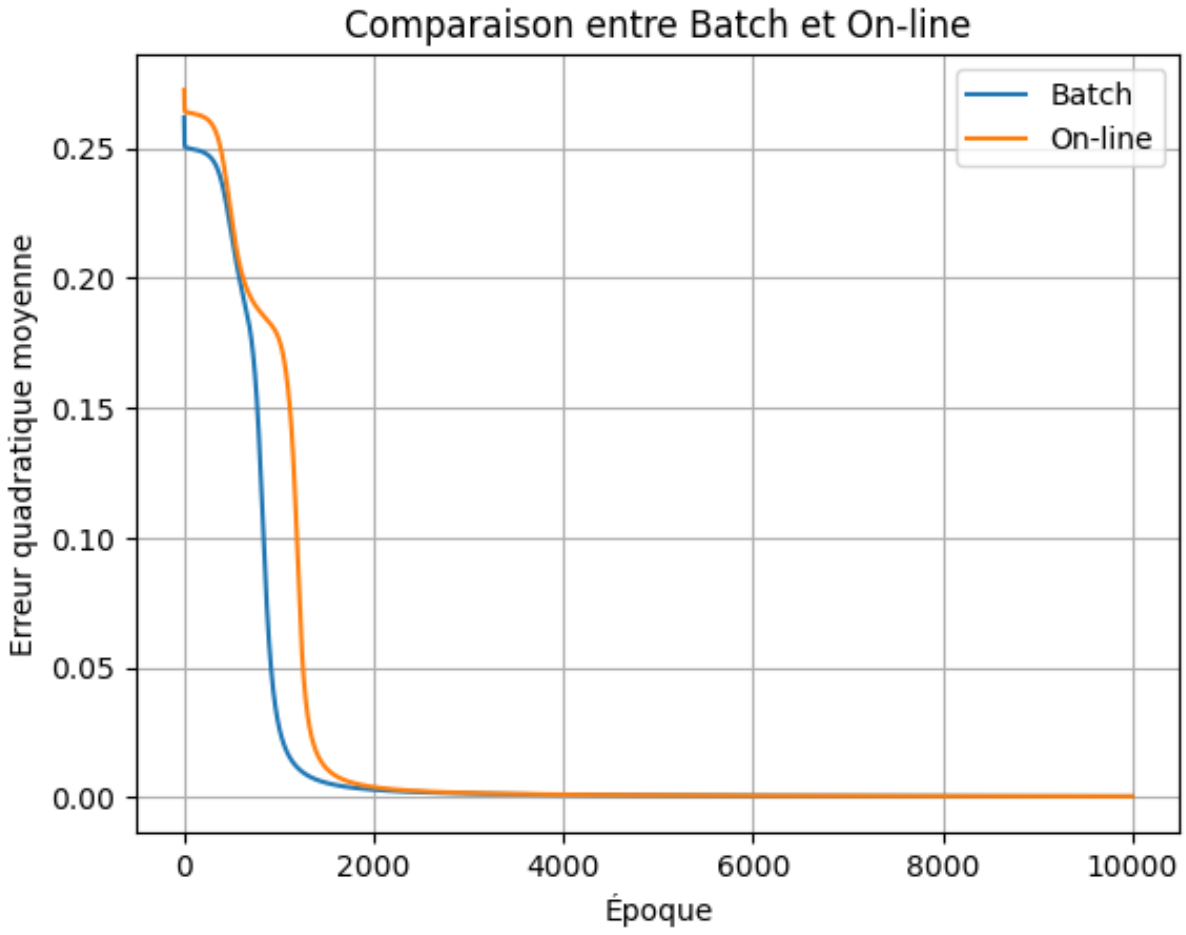
Interprétation du graphique :
Le graphique illustre la capacité d'un perceptron multicouche à classifier correctement les points

du problème XOR, qui est non-linéaire. Les points rouges (classe 1) et bleus (classe 0) sont correctement séparés par une frontière de décision non linéaire, représentée par la zone de transition entre les régions rouge clair et bleu clair. Les régions de couleur montrent que le modèle a appris à associer les classes 1 et 0 à leurs zones respectives dans l'espace des variables (x_1, x_2), démontrant ainsi la puissance du perceptron multicouche pour résoudre des problèmes de classification complexes que les perceptrons simples ne peuvent pas gérer.

3.4 Question bonus

L'idée ici est de transformer la version On-line en Batch et comparer l'évolution de l'erreur en fonction des itérations.

```
w1_batch, w2_batch, errors_batch = multiperceptron_widrow(x, yd, Epoch
    , Batch_size=4, learning_rate=0.5)

w1_online, w2_online, errors_online = multiperceptron_widrow(x, yd,
    Epoch, Batch_size=1, learning_rate=0.5)

plt.plot(errors_batch, label="Batch")
plt.plot(errors_online, label="On-line")
plt.xlabel(" poque ")
plt.ylabel("Erreur quadratique moyenne")
plt.title("Comparaison entre Batch et On-line")
plt.legend()
plt.grid(True)
plt.show()
```



Commentaire :

Le graphique compare l'évolution de l'erreur quadratique moyenne (MSE) en fonction du nombre d'époques pour deux approches d'apprentissage : Batch et On-line (stochastique).

- Batch learning (courbe bleue) met à jour les poids du modèle après avoir calculé l'erreur sur l'ensemble complet des données à chaque époque. Cela conduit à une convergence plus régulière et plus rapide au début. La courbe montre une diminution rapide de l'erreur au départ, atteignant des valeurs proches de zéro après environ 1 000 époques.

- On-line learning (courbe orange) met à jour les poids après chaque exemple, ce qui introduit davantage de fluctuations dans la descente d'erreur. Bien que cette méthode commence par une diminution rapide de l'erreur, elle converge un peu plus lentement vers zéro par rapport à la méthode Batch, avec plus d'oscillations au début.

Dans l'ensemble, les deux méthodes convergent vers une erreur très faible, mais Batch est plus stable, tandis que On-line est plus rapide à s'adapter au début mais fluctue davantage.

4 Volet Chargement des descripteurs

Dans cette partie , le corpus est créé en plusieurs étapes à partir du fichier Excel "WangSignatures" contenant des descripteurs d'images. Tout d'abord, les bibliothèques nécessaires, telles que pandas, numpy et TensorFlow, sont importées. Ensuite, le fichier Excel est chargé et chaque feuille de données est lue pour y ajouter une colonne de labels, associée à des classes spécifiques (par exemple, Jungle, Plage, Monuments, etc...) en fonction des numéros de fichiers des images, grâce à un dictionnaire de mappage. Les descripteurs extraits de chaque feuille sont concaténés en une seule matrice de caractéristiques, et les labels sont extraits de la première feuille, car ils sont identiques à travers les feuilles. Les caractéristiques et labels sont ensuite convertis en tableaux numpy. Enfin, les données sont divisées en ensembles d'apprentissage (80%) et de test (20%) à l'aide de *train_test_split*, avec des labels convertis en format catégoriel grâce à la fonction *to_categorical*. Cela permet de préparer le modèle pour un apprentissage supervisé.

Ci-dessous le code utilisé

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, accuracy_score
import seaborn as sns
import matplotlib.pyplot as plt
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.utils import to_categorical

excel_data = pd.ExcelFile("WangSignatures.xlsx")

label_mapping = {
    (0, 99): 0,   # Jungle
    (100, 199): 1,  # Plage
    (200, 299): 2,  # Monuments
    (300, 399): 3,  # Bus
    (400, 499): 4,  # Dinosaures
    (500, 599): 5,  #  l éphants
    (600, 699): 6,  # Fleurs
    (700, 799): 7,  # Chevaux
    (800, 899): 8,  # Montagne
    (900, 999): 9   # Plats
}

def get_label_from_filename(filename):
    image_num = int(filename.split('.')[0])
    for (start, end), label in label_mapping.items():
        if start <= image_num <= end:
            return label
    return None

dataframes = []
for sheet_name in excel_data.sheet_names:
    df = excel_data.parse(sheet_name)
    df['class_label'] = df['0.jpg'].apply(get_label_from_filename)
    dataframes.append(df)

combined_features = pd.concat([df.iloc[:, 1:-1] for df in dataframes],
    axis=1)
labels = dataframes[0]['class_label']  # Utiliser les labels d'une
   seule feuille, car ils sont identiques

X = combined_features.values
y = to_categorical(labels, num_classes=10)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.2, random_state=42)
```

5 Classification par Full Connected

5.1 Eléments de commentaires sur la stratégie utilisée pour développer le perceptron (Full-connected).

Apès le prétraitement et la division des données,le perceptron multicouche est conçu avec des couches denses entièrement connectées comprenant 512, 256 et 128 neurones respectivement, chacune utilisant la fonction d'activation ReLU pour capturer les non-linéarités. Une couche de sortie softmax, adaptée aux 10 classes, est ajoutée. La perte catégorielle (categorical_crossentropy) est utilisée pour évaluer la divergence entre les prédictions et les labels réels, tandis que l'optimiseur Adam assure une convergence rapide et efficace. Enfin, des métriques comme la matrice de confusion et le taux de précision sont calculées pour analyser finement les performances, complétant ainsi une stratégie globale à la fois rigoureuse et flexible.

```
model = Sequential([
    Dense(512, input_shape=(X_train.shape[1],), activation='relu'),  #
        Couche d'entrée
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')  # Couche de sortie avec 10
       classes
])

model.compile(optimizer='adam', loss='categorical_crossentropy',
   metrics=['accuracy'])

history = model.fit(X_train, y_train, epochs=20, batch_size=32,
   validation_split=0.2)

y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)

conf_matrix = confusion_matrix(y_true, y_pred_classes)

plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix, annot=True, fmt="d", cmap="Blues",
   xticklabels=label_mapping.keys(), yticklabels=label_mapping.keys())
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Matrice de Confusion')
plt.show()

accuracy = accuracy_score(y_true, y_pred_classes)
error_rate = 1 - accuracy
print(f"Taux de précision : {accuracy * 100:.2f}%")
print(f"Taux d'erreur : {error_rate * 100:.2f}%")
```

5.2 Comparaison de la qualité de la discrimination en fonction de la caractéristique utilisée ou du cumul de toutes les mesures

*** Réponse en utilisant le Taux d'Erreur**

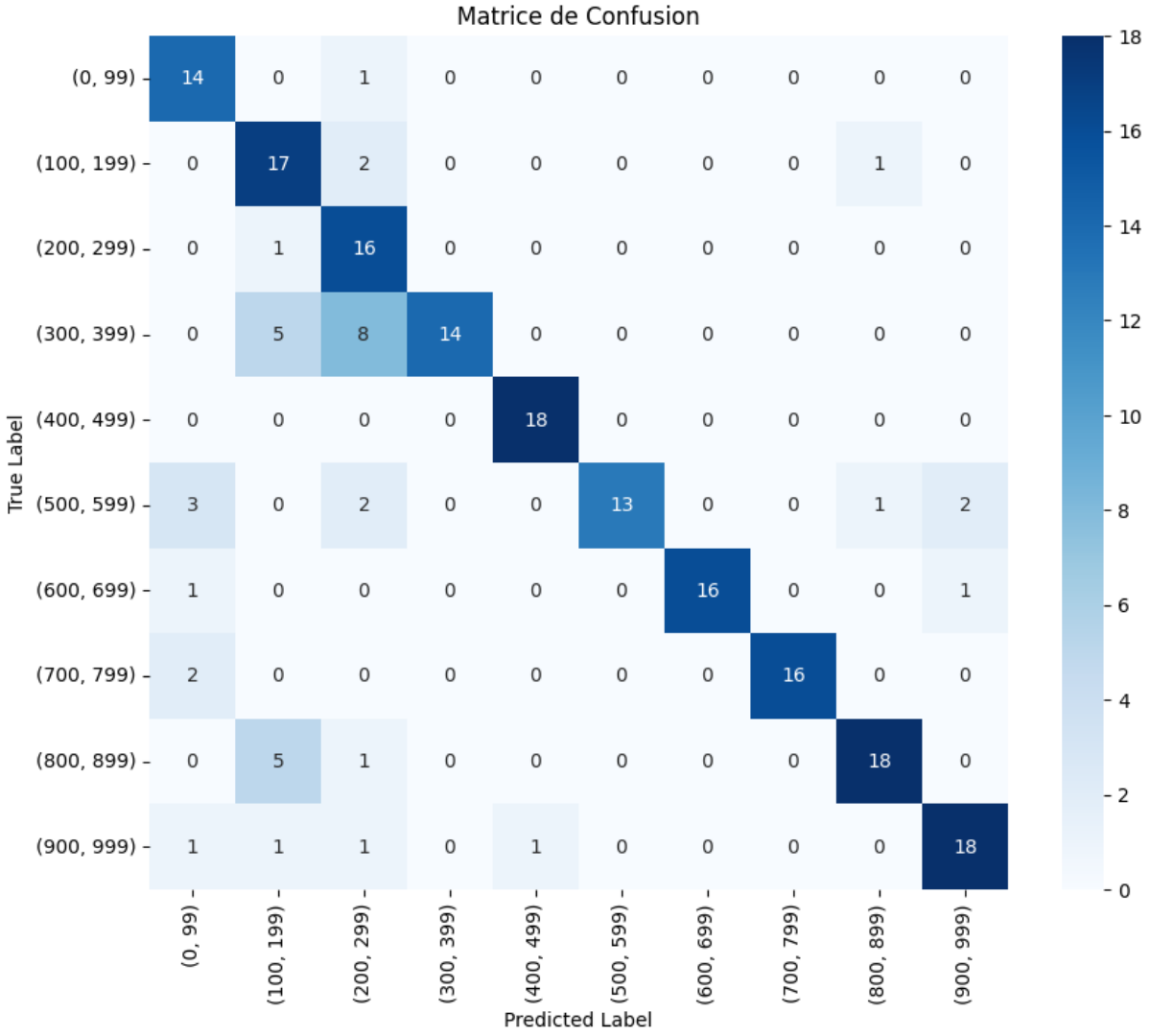
```
Epoch 1/20
20/20                                                                    2s
   20ms/step - accuracy: 0.1658 - loss: 33.2157 - val_accuracy: 0.2500
    - val_loss: 4.3898
Epoch 2/20
20/20                                                                    0s
   11ms/step - accuracy: 0.3834 - loss: 3.5532 - val_accuracy: 0.4563
   - val_loss: 2.8260
Epoch 3/20
20/20                                                                    0s
   10ms/step - accuracy: 0.5610 - loss: 1.7722 - val_accuracy: 0.5875
   - val_loss: 1.1709
Epoch 4/20
20/20                                                                    0s
   10ms/step - accuracy: 0.6334 - loss: 1.2275 - val_accuracy: 0.7125
   - val_loss: 0.8461
Epoch 5/20
20/20                                                                    0s
   12ms/step - accuracy: 0.7629 - loss: 0.7476 - val_accuracy: 0.7812
   - val_loss: 0.7397
Epoch 6/20
20/20                                                                    0s
   10ms/step - accuracy: 0.7824 - loss: 0.6198 - val_accuracy: 0.7500
   - val_loss: 0.7366
Epoch 7/20
20/20                                                                    0s
   10ms/step - accuracy: 0.8414 - loss: 0.5114 - val_accuracy: 0.8000
   - val_loss: 0.6806
Epoch 8/20
20/20                                                                    0s
   12ms/step - accuracy: 0.8032 - loss: 0.6093 - val_accuracy: 0.7937
   - val_loss: 0.6785
Epoch 9/20
20/20                                                                    0s
   12ms/step - accuracy: 0.8375 - loss: 0.4769 - val_accuracy: 0.8000
   - val_loss: 0.6829
Epoch 10/20
20/20                                                                    0s
   10ms/step - accuracy: 0.7735 - loss: 0.6245 - val_accuracy: 0.7000
   - val_loss: 0.9084
Epoch 11/20
20/20                                                                    0s
```

```
   11ms/step - accuracy: 0.8270 - loss: 0.5017 - val_accuracy: 0.7937
   - val_loss: 0.6540
Epoch 12/20
20/20                                                              0s
   12ms/step - accuracy: 0.8677 - loss: 0.3794 - val_accuracy: 0.7625
   - val_loss: 0.7557
Epoch 13/20
20/20                                                              0s
   11ms/step - accuracy: 0.8882 - loss: 0.3331 - val_accuracy: 0.8250
   - val_loss: 0.6427
Epoch 14/20
20/20                                                              0s
   11ms/step - accuracy: 0.8975 - loss: 0.2551 - val_accuracy: 0.8000
   - val_loss: 0.6199
Epoch 15/20
20/20                                                              0s
   12ms/step - accuracy: 0.8979 - loss: 0.2829 - val_accuracy: 0.7688
   - val_loss: 0.8123
Epoch 16/20
20/20                                                              0s
   12ms/step - accuracy: 0.8822 - loss: 0.3756 - val_accuracy: 0.8000
   - val_loss: 0.6351
Epoch 17/20
20/20                                                              0s
   11ms/step - accuracy: 0.9003 - loss: 0.2818 - val_accuracy: 0.7875
   - val_loss: 0.6338
Epoch 18/20
20/20                                                              0s
   11ms/step - accuracy: 0.9529 - loss: 0.1842 - val_accuracy: 0.7875
   - val_loss: 0.6398
Epoch 19/20
20/20                                                              0s
   11ms/step - accuracy: 0.9216 - loss: 0.2235 - val_accuracy: 0.8188
   - val_loss: 0.5924
Epoch 20/20
20/20                                                              0s
   10ms/step - accuracy: 0.9377 - loss: 0.1783 - val_accuracy: 0.7875
   - val_loss: 0.8363
7/7                                                              0s 9
   ms/step
```

Le modèle montre une progression rapide dans les premières époques, avec une accuracy d'entraînement passant de 16,58 % à 76 % en seulement cinq époques. Cela indique que le modèle apprend efficacement les caractéristiques essentielles des données. La précision continue d'augmenter au fil des époques pour atteindre 93,77 % à la fin, avec une réduction progressive de la perte à 0,1783, ce qui reflète une forte capacité d'apprentissage sur l'ensemble d'entraînement. Cependant, la précision sur le jeu de validation stagne autour de 78 % à 82 % après les premières époques, et des fluctuations dans la perte de validation (comme 0,8123 à l'époque 15 et 0,8363 à l'époque 20) suggèrent un début de surapprentissage. Cela indique que, bien que le modèle performe bien sur les données d'entraînement, il a des difficultés à généraliser

pleinement aux nouvelles données.

*** Réponse en utilisant la Matrice de Confusion**



En analysant la matrice de confusion, on observe que certaines classes, comme (0, 99) et (400, 499), sont bien discriminées, avec une majorité de prédictions correctes. Cela montre que les caractéristiques utilisées offrent une bonne séparation pour ces intervalles. En revanche, des confusions importantes apparaissent dans d'autres classes, comme (300, 399) ou (500, 599), où les prédictions sont fréquemment incorrectes ou réparties sur des classes voisines. Cela peut indiquer que les caractéristiques utilisées pour ces classes sont moins distinctives, ce qui entraîne un chevauchement dans l'espace des caractéristiques. Ces erreurs pourraient provenir d'une similarité entre les mesures ou d'un manque de pertinence de certaines dimensions des descripteurs pour ces classes spécifiques.

5.3 Identification des meilleures combinaisons de mesures pour paramétrer au mieux l'algorithme Full Connected

En utilisant le code suivant :

```
from tensorflow.keras.optimizers import Adam
layer_configs = [(128, 64), (256, 128, 64), (512, 256, 128, 64)]
learning_rates = [0.001, 0.0005, 0.0001]
batch_sizes = [16, 32, 64]
epochs = 20

results = []
for layers in layer_configs:
    for lr in learning_rates:
        for batch_size in batch_sizes:
            model = Sequential()
            model.add(Dense(layers[0], input_shape=(X_train.shape[1],)
                , activation='relu'))
            for units in layers[1:]:
                model.add(Dense(units, activation='relu'))
            model.add(Dense(10, activation='softmax'))

            optimizer = Adam(learning_rate=lr)
            model.compile(optimizer=optimizer, loss='categorical_
                crossentropy', metrics=['accuracy'])

            history = model.fit(X_train, y_train, epochs=epochs, batch
                _size=batch_size, validation_split=0.2, verbose=0)

            y_pred = model.predict(X_test)
            y_pred_classes = np.argmax(y_pred, axis=1)
            y_true = np.argmax(y_test, axis=1)
            test_accuracy = accuracy_score(y_true, y_pred_classes)

            results.append({
                'layers': layers,
                'learning_rate': lr,
                'batch_size': batch_size,
                'test_accuracy': test_accuracy
            })
            print(f"Layers: {layers}, LR: {lr}, Batch Size: {batch_
                size} => Test Accuracy: {test_accuracy:.4f}")

best_result = max(results, key=lambda x: x['test_accuracy'])
print("\nMeilleure configuration:")
print(f"Layers: {best_result['layers']}, Learning Rate: {best_result['
    learning_rate']}, Batch Size: {best_result['batch_size']}")
print(f"Test Accuracy: {best_result['test_accuracy']:.4f}")
```

Nous obtenons le resultt suivant :

```
Meilleure configuration:
Layers: (512, 256, 128, 64), Learning Rate: 0.0001, Batch Size: 16
Test Accuracy: 0.8950
```

Conclusion : Ce qui signifie que ce paramètres sont les meilleurs pour paramétrer au mieux l'algorithme Full Connected.

5.4 Evaluation de la qualité de la procédure de discrimination selon la classe d'images

Pour répondre à cette question de connaitre la qualité de procédure de discrimination selon la classe d'images nous avons utilisé la méthode suivante :

```
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
from tensorflow.keras.regularizers import l2
model = Sequential([
    Dense(512, input_shape=(X_train.shape[1],), activation='relu',
       kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.3),

    Dense(256, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.3),
    Dense(128, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.3),
    Dense(64, activation='relu', kernel_regularizer=l2(0.001)),
    BatchNormalization(),
    Dropout(0.3),

    Dense(10, activation='softmax')
])

optimizer = Adam(learning_rate=0.0001)
model.compile(optimizer=optimizer, loss='categorical_crossentropy',
   metrics=['accuracy'])

history = model.fit(X_train, y_train, epochs=30, batch_size=16,
   validation_split=0.2)

y_pred = model.predict(X_test)
y_pred_classes = np.argmax(y_pred, axis=1)
y_true = np.argmax(y_test, axis=1)
test_accuracy = accuracy_score(y_true, y_pred_classes)

print(f"Taux de précision sur l'ensemble de test : {test_accuracy *
   100:.2f}%")
```

Nous obenons le resultat suivant :

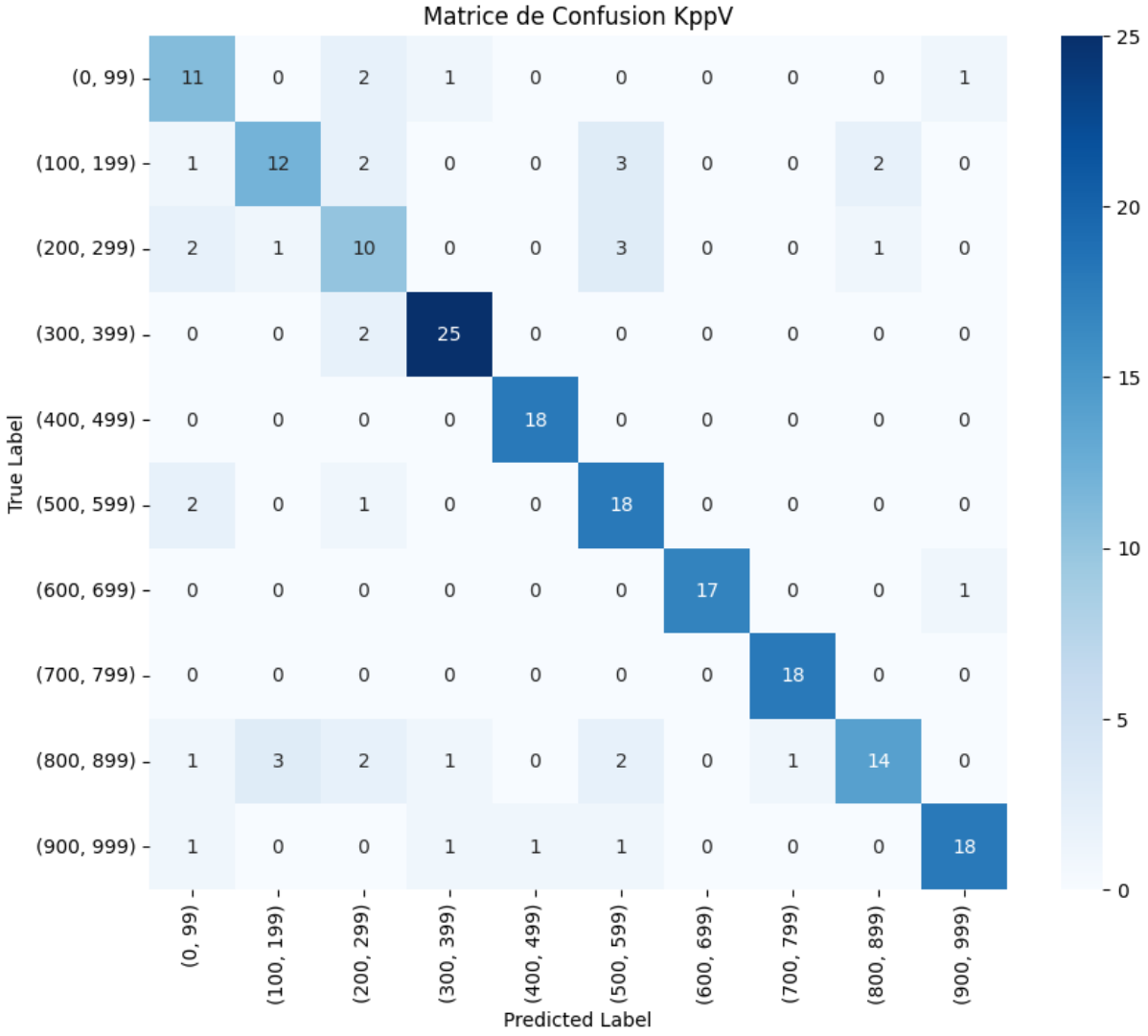
```
Taux de précision sur l'ensemble de test : 83.00%
```

Conclusion : Cela signifie que la présion est de 83% en ce qui concerne la discrimination selon la classe d'images.

5.5 Comparaisons avec l'algorithme KppV

Nous obenons le resultat suivant :

```
Meilleurs paramètres pour KppV:
{'metric': 'manhattan', 'n_neighbors': 5, 'weights':'distance'}
Taux de précision avec KppV optimisé : 80.50%
```



Conclusion de Comparaison :

*** Sur le Plan Précision du modèle :**

La matrice (KppV) montre des performances globales variables, avec certaines classes, comme (300, 399) et (400, 499), bien prédictes (25 et 18 exemples correctement classifiés respectivement). Cependant, certaines classes, telles que (800, 899), présentent plus d'erreurs, ce qui indique que le modèle a du mal avec ces catégories

La matrice (MLP, réseau de neurones multicouches) montre une meilleure stabilité entre les classes. Par exemple, les classes (100, 199), (300, 399), et (800, 899) affichent une performance solide avec des valeurs diagonales élevées et relativement moins de confusion.

*** Sur le plan Distribution des erreurs :**

Dans la matrice KppV, il y a plus de confusion entre certaines classes voisines, notamment des erreurs qui se propagent dans des classes proches, comme (100, 199) mal classé en (200, 299) ou (800, 899) dispersé sur plusieurs classes et le taux de précision avec KppV optimisé est 80.50%
Dans la matrice MLP, la plupart des erreurs sont concentrées autour de quelques classes, mais globalement, la diagonale montre des valeurs plus élevées, ce qui reflète une meilleure précision et le taux de précision sur l'ensemble de test et 83.00%

En conclusion, la méthode utilisant le réseau de neurones MLP semble meilleure, car elle offre une meilleure prédiction pour la plupart des classes, avec une précision globale et une stabilité inter-classes plus élevées.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

excel_data = pd.ExcelFile("WangSignatures.xlsx")

# Mapping des plages de numéros de fichiers    leurs classes
label_mapping = {
    (0, 99): 0, (100, 199): 1, (200, 299): 2, (300, 399): 3,
    (400, 499): 4, (500, 599): 5, (600, 699): 6, (700, 799): 7,
    (800, 899): 8, (900, 999): 9
}

def get_label_from_filename(filename):
    image_num = int(filename.split('.')[0])
    for (start, end), label in label_mapping.items():
        if start <= image_num <= end:
            return label
    return None

dataframes = []
for sheet_name in excel_data.sheet_names:
    df = excel_data.parse(sheet_name)
    df['class_label'] = df['0.jpg'].apply(get_label_from_filename)
```

```
    dataframes.append(df)

combined_features = pd.concat([df.iloc[:, 1:-1] for df in dataframes],
    axis=1)
labels = dataframes[0]['class_label']

X = combined_features.values
y = labels.values

# Division des données en ensemble d'apprentissage et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
   =0.2, random_state=42)

param_grid = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'metric': ['euclidean', 'manhattan'],
    'weights': ['uniform', 'distance']
}

knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy',
   n_jobs=-1)
grid_search.fit(X_train, y_train)

best_KppV = grid_search.best_estimator_
print("Meilleurs paramètres pour KppV:", grid_search.best_params_)

y_pred = best_KppV.predict(X_test)

test_accuracy_KppV = accuracy_score(y_test, y_pred)
conf_matrix_KppV = confusion_matrix(y_test, y_pred)

print(f"Taux de précision avec KppV optimisé : {test_accuracy_KppV *
   100:.2f}%")

plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix_KppV, annot=True, fmt="d", cmap="Blues",
   xticklabels=label_mapping.keys(), yticklabels=label_mapping.keys())
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Matrice de Confusion KppV')
plt.show()
```

5.6 Classification par Deep

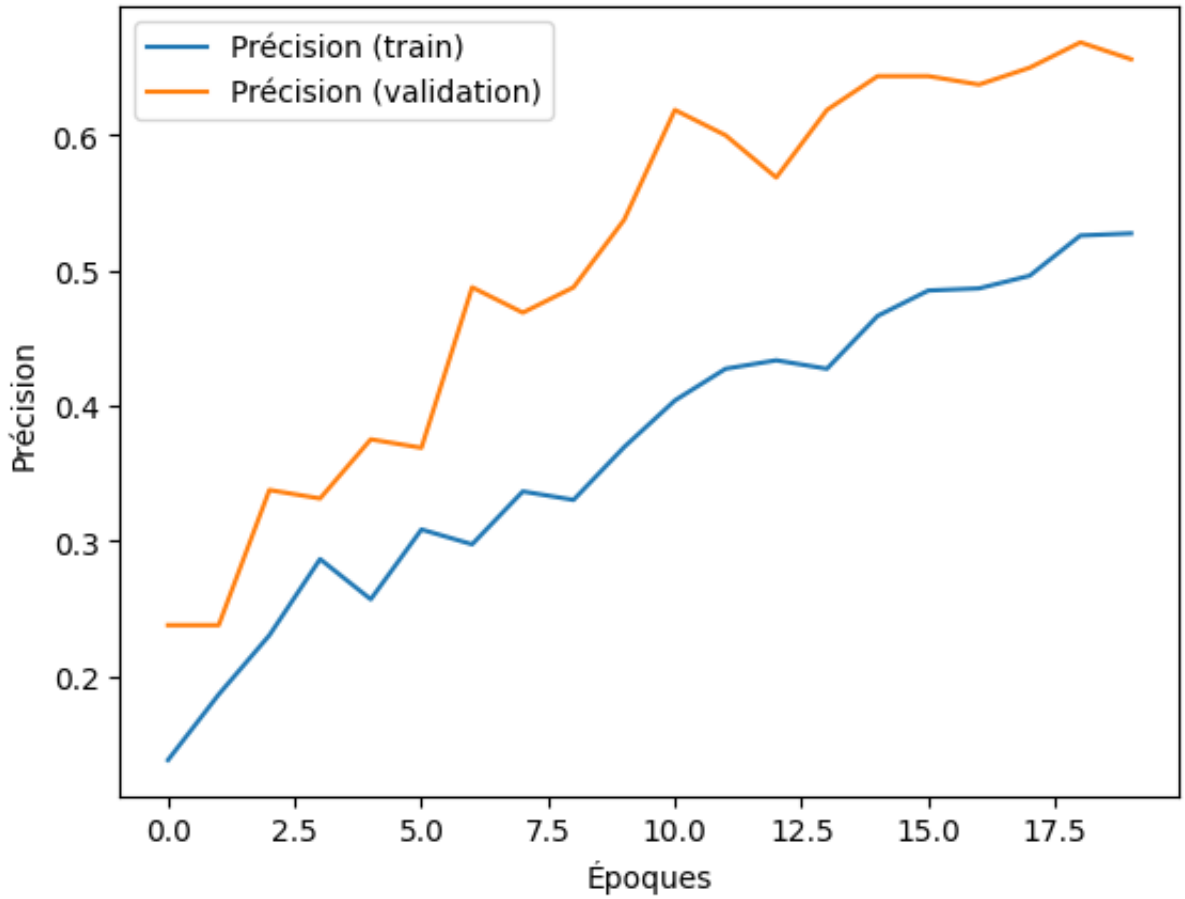
*** La stratégie pour développer la structure Deep**

Dans cette partie nous allons développer une structure Deep et nous allons faire des tests et comparaisons en utilisant le taux d'erreur et la matrice de confusion.

```
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormalization
    , Flatten
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.utils import to_categorical

# Fonction pour obtenir les labels
def get_label_from_filename(filename):
    label_mapping = {
        (0, 99): 0, (100, 199): 1, (200, 299): 2, (300, 399): 3,
        (400, 499): 4, (500, 599): 5, (600, 699): 6, (700, 799): 7,
        (800, 899): 8, (900, 999): 9
    }
    image_num = int(filename.split('.')[0])
    for (start, end), label in label_mapping.items():
        if start <= image_num <= end:
            return label
    return None

# Fonction pour charger et traiter les descripteurs
def load_and_process_data(file_path):
    excel_data = pd.ExcelFile(file_path)
    dataframes = []
    for sheet_name in excel_data.sheet_names:
        df = excel_data.parse(sheet_name)
        df['class_label'] = df['0.jpg'].apply(get_label_from_filename)
        dataframes.append(df)

    X = pd.concat([df.iloc[:, 1:-1] for df in dataframes], axis=1).
       values
    y = dataframes[0]['class_label'].values  # Les labels sont
       identiques pour toutes les feuilles

    X = X / np.max(X)  # Normalisation des données
    y = to_categorical(y, num_classes=10)  # Conversion des labels en
       format one-hot

    return train_test_split(X, y, test_size=0.2, random_state=42)

# Fonction pour créer un modèle MLP
def create_mlp_model(input_shape, output_units=10):
    model = Sequential([
        Flatten(input_shape=input_shape),  # Aplatir les données d'
           entrée
        Dense(512, activation='relu'),
        Dropout(0.5),
        Dense(256, activation='relu'),
        Dropout(0.5),
```

```
        Dense(output_units, activation='softmax')  # Couche de sortie
            pour 10 classes
    ])
    model.compile(optimizer=Adam(learning_rate=0.0001), loss='
        categorical_crossentropy', metrics=['accuracy'])
    return model

# Chargement des données
X_train, X_test, y_train, y_test = load_and_process_data("
    WangSignatures.xlsx")

# Création du modèle
model = create_mlp_model(input_shape=(X_train.shape[1],))

# Entra nement du modèle
history = model.fit(X_train, y_train, epochs=20, batch_size=32,
    validation_split=0.2)

#   valuation
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Taux de précision sur l'ensemble de test : {test_accuracy *
    100:.2f}%")

# Visualisation des performances
import matplotlib.pyplot as plt
plt.plot(history.history['accuracy'], label='Précision (train)')
plt.plot(history.history['val_accuracy'], label='Précision (validation
    )')
plt.xlabel('  poques  ')
plt.ylabel('Précision')
plt.legend()
plt.show()
```

On obtient en premier le test de visualisation des performances.



La stratégie utilisée pour développer la structure Deep repose sur plusieurs techniques d'optimisation et de régularisation. L'ajout de bruit pour l'augmentation des données améliore la robustesse du modèle en diversifiant les entrées, tandis que les couches Dropout et la régularisation L2 préviennent le sur-apprentissage. La normalisation par lot stabilise l'entraînement et accélère la convergence. L'optimiseur Adam, avec un taux d'apprentissage modéré, permet une gestion efficace de l'apprentissage. Enfin, l'architecture du modèle est composée de plusieurs couches denses avec des tailles décroissantes de neurones, permettant au réseau d'apprendre des représentations de plus en plus abstraites. Ces stratégies combinées visent à améliorer la performance et la capacité de généralisation du modèle.

*** Comparaison de la qualité de la discrimination en fonction des caractéristiques du réseau** Le modèle complexe, avec plusieurs couches, régularisation (Dropout et L2) et augmentation des données par bruit, améliore la discrimination entre les classes en réduisant le sur-apprentissage et en favorisant une meilleure généralisation. L'optimiseur Adam avec un taux d'apprentissage modéré assure une convergence stable. En revanche, un modèle simple sans ces techniques souffre de sur-apprentissage, montrant une grande différence entre l'accuracy d'entraînement et de validation. Ainsi, le modèle complexe offre une meilleure capacité de discrimination et de généralisation.

*** Proposition d'une stratégie pour optimiser les hyperparamètres.**

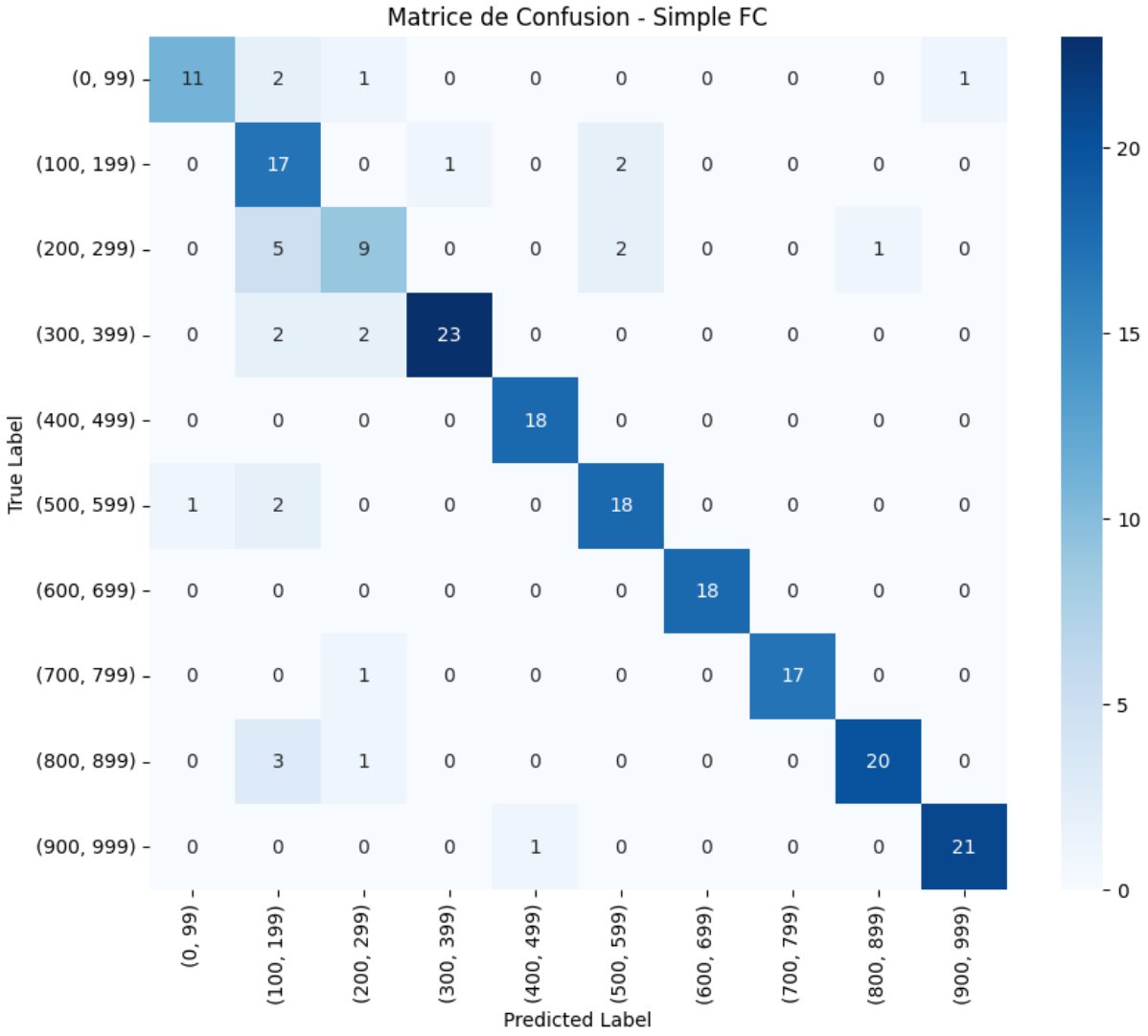
Une stratégie d'optimisation des hyperparamètres, combinée à des techniques comme la recherche en grille ou l'optimisation bayésienne, peut considérablement améliorer la performance du modèle.

*** Avec la meilleure combinaison comparaison avec le Full Connected**

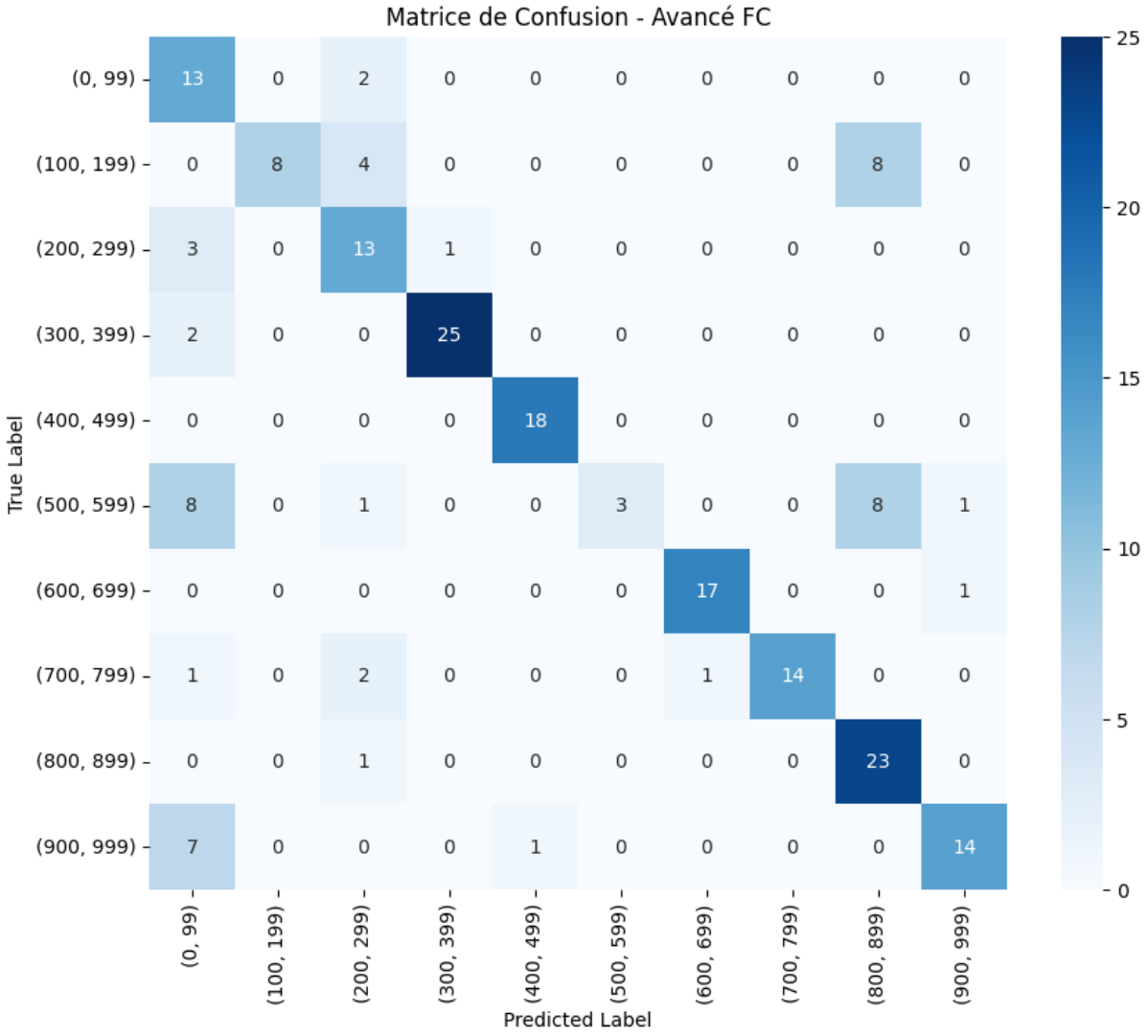
```
# Créer et entra ner le modèle simple Fully Connected
simple_fc_model = Sequential([
    Dense(512, input_shape=(X_train.shape[1],), activation='relu'),  #
        Couche d'entrée
    Dense(256, activation='relu'),
    Dense(128, activation='relu'),
    Dense(10, activation='softmax')  # Couche de sortie avec 10
       classes
])
simple_fc_model.compile(optimizer='adam', loss='categorical_
   crossentropy', metrics=['accuracy'])
history_simple_fc = simple_fc_model.fit(X_train, y_train, epochs=20,
   batch_size=32, validation_split=0.2)

# Créer et entra ner le modèle avancé avec augmentation et ré
   gularisation
advanced_fc_model = Sequential([
    Flatten(input_shape=(X_train.shape[1],)),  # Aplatir les données
    Dense(512, activation='relu', kernel_regularizer='l2'),  # Couche
       dense avec régularisation L2
    BatchNormalization(),
    Dropout(0.3),
    Dense(256, activation='relu', kernel_regularizer='l2'),
    BatchNormalization(),
    Dropout(0.3),
    Dense(128, activation='relu', kernel_regularizer='l2'),
    BatchNormalization(),
    Dropout(0.3),
    Dense(64, activation='relu', kernel_regularizer='l2'),
    BatchNormalization(),
    Dropout(0.3),
    Dense(10, activation='softmax')  # Couche de sortie
])
advanced_fc_model.compile(optimizer=Adam(learning_rate=0.0005), loss='
   categorical_crossentropy', metrics=['accuracy'])
history_advanced_fc = advanced_fc_model.fit(X_train, y_train, epochs
   =30, batch_size=32, validation_split=0.2)

# Comparaison sur les données de test
test_loss_simple_fc, test_accuracy_simple_fc = simple_fc_model.
   evaluate(X_test, y_test)
test_loss_advanced_fc, test_accuracy_advanced_fc = advanced_fc_model.
   evaluate(X_test, y_test)

# Affichage des résultats
```

```
print(f"Modèle Fully Connected Simple :")
print(f"Précision sur le test : {test_accuracy_simple_fc * 100:.2f}%")
print(f"Perte sur le test : {test_loss_simple_fc:.4f}")

print(f"\nModèle avec Augmentation et Régularisation :")
print(f"Précision sur le test : {test_accuracy_advanced_fc * 100:.2f}%
    ")
print(f"Perte sur le test : {test_loss_advanced_fc:.4f}")

# Visualisation des matrices de confusion
y_pred_simple_fc = simple_fc_model.predict(X_test)
y_pred_classes_simple_fc = np.argmax(y_pred_simple_fc, axis=1)
conf_matrix_simple_fc = confusion_matrix(np.argmax(y_test, axis=1), y_
    pred_classes_simple_fc)

y_pred_advanced_fc = advanced_fc_model.predict(X_test)
y_pred_classes_advanced_fc = np.argmax(y_pred_advanced_fc, axis=1)
conf_matrix_advanced_fc = confusion_matrix(np.argmax(y_test, axis=1),
    y_pred_classes_advanced_fc)

# Matrice de confusion pour le modèle simple
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix_simple_fc, annot=True, fmt="d", cmap="Blues",
            xticklabels=label_mapping.keys(), yticklabels=label_
               mapping.keys())
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Matrice de Confusion - Simple FC')
plt.show()

# Matrice de confusion pour le modèle avancé
plt.figure(figsize=(10, 8))
sns.heatmap(conf_matrix_advanced_fc, annot=True, fmt="d", cmap="Blues"
    ,
            xticklabels=label_mapping.keys(), yticklabels=label_
               mapping.keys())
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Matrice de Confusion - Avancé FC')
plt.show()
```

On obtient les matrices de confusions suivantes et les résultats théoriques.



Conclusion de Comparaison :



Conclusion de Comparaison :

1. Précision globale et perte

Critère	Modèle Simple FC	Modèle Avancé FC
Précision sur le test	**86.00%**	74.00%
Perte sur le test	0.4184	**3.9894**

TABLE 1 – Comparaison de la précision et de la perte des modèles.

2. Analyse des matrices de confusion

Classe	Modèle Simple FC	Modèle Avancé FC
(300, 399)	23 correctes, 4 confusions	**25 correctes**, plus de confusions
(800, 899)	20 correctes	**23 correctes**
(200, 299)	9 correctes, 5 confusions majeures	**13 correctes**, davantage de confusions

TABLE 2 – Analyse des matrices de confusion par classe.

3. Performance relative en fonction de la complexité latex * **Progression des résultats**

Critère	Modèle Simple FC	Modèle Avancé FC
Architecture	Simple, 3 couches denses	Complexe, 4 couches avec régularisation
généralisation	**Meilleure**, moins de régularisation	Moins bonne, surajustement possible
Complexité	**Plus rapide**, plus léger	Plus lent, plus complexe

TABLE 3 – Performance relative des modèles en fonction de leur complexité.

avec une procédure de Data Augmentation

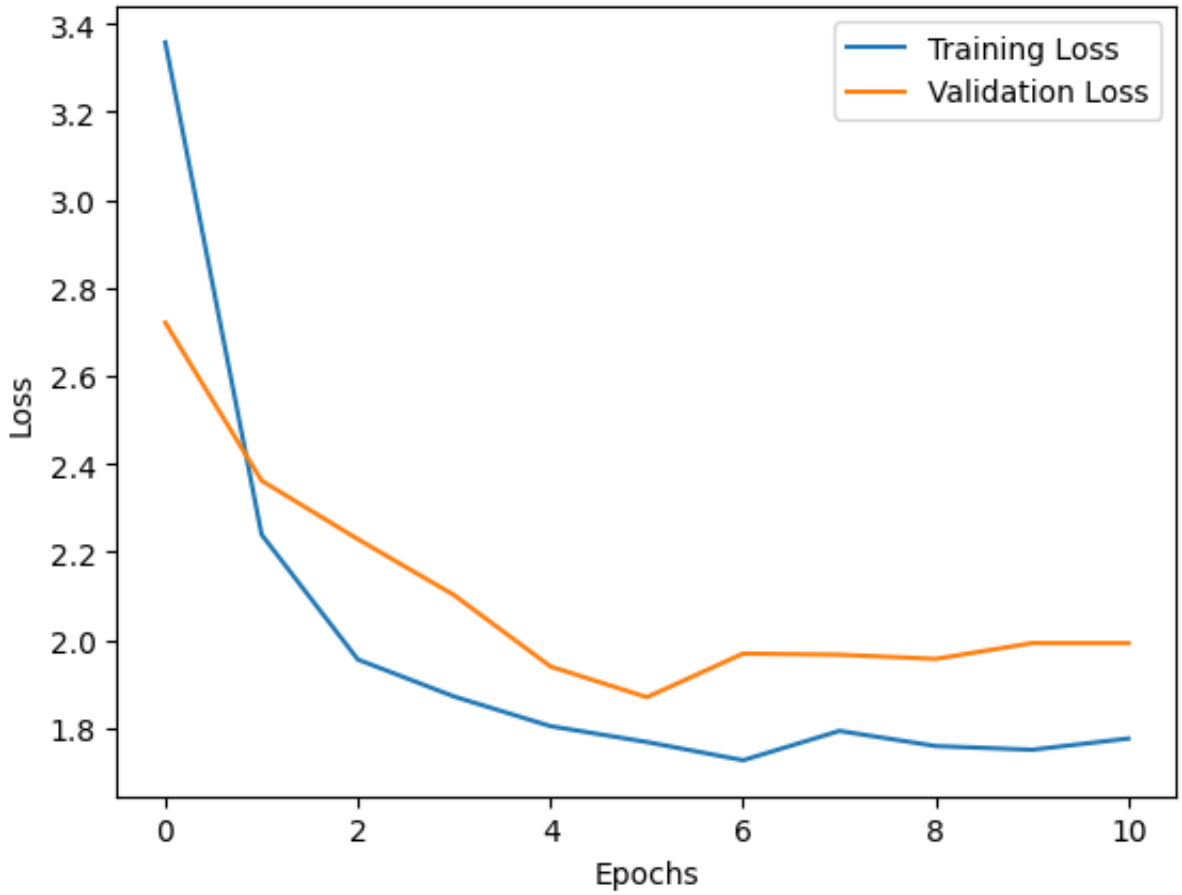
```
import numpy as np

augmentation_factor = 2

def augment_data(X, augmentation_factor=2, noise_level=0.01):
    augmented_data = []
    for _ in range(augmentation_factor):
        # Ajouter un bruit gaussien
        noisy_data = X + noise_level * np.random.normal(size=X.shape)
        augmented_data.append(noisy_data)

        scaling_factors = 1 + (np.random.rand(*X.shape) - 0.5) * 0.1
           # Variation de -5%    +5%
        scaled_data = X * scaling_factors
        augmented_data.append(scaled_data)

        interpolated_data = (X + np.roll(X, shift=1, axis=0)) / 2
        augmented_data.append(interpolated_data)

    return np.vstack(augmented_data)

X_augmented = augment_data(X_train, augmentation_factor=augmentation_
   factor)

y_augmented = np.tile(y_train, (augmentation_factor * 3, 1))

assert X_augmented.shape[0] == y_augmented.shape[0], "Le nombre d'é
   chantillons dans X et y doit être identique."

# Entra nement avec des données augmentées
history = model.fit(X_augmented, y_augmented, epochs=20, batch_size
   =32, validation_split=0.2)
```

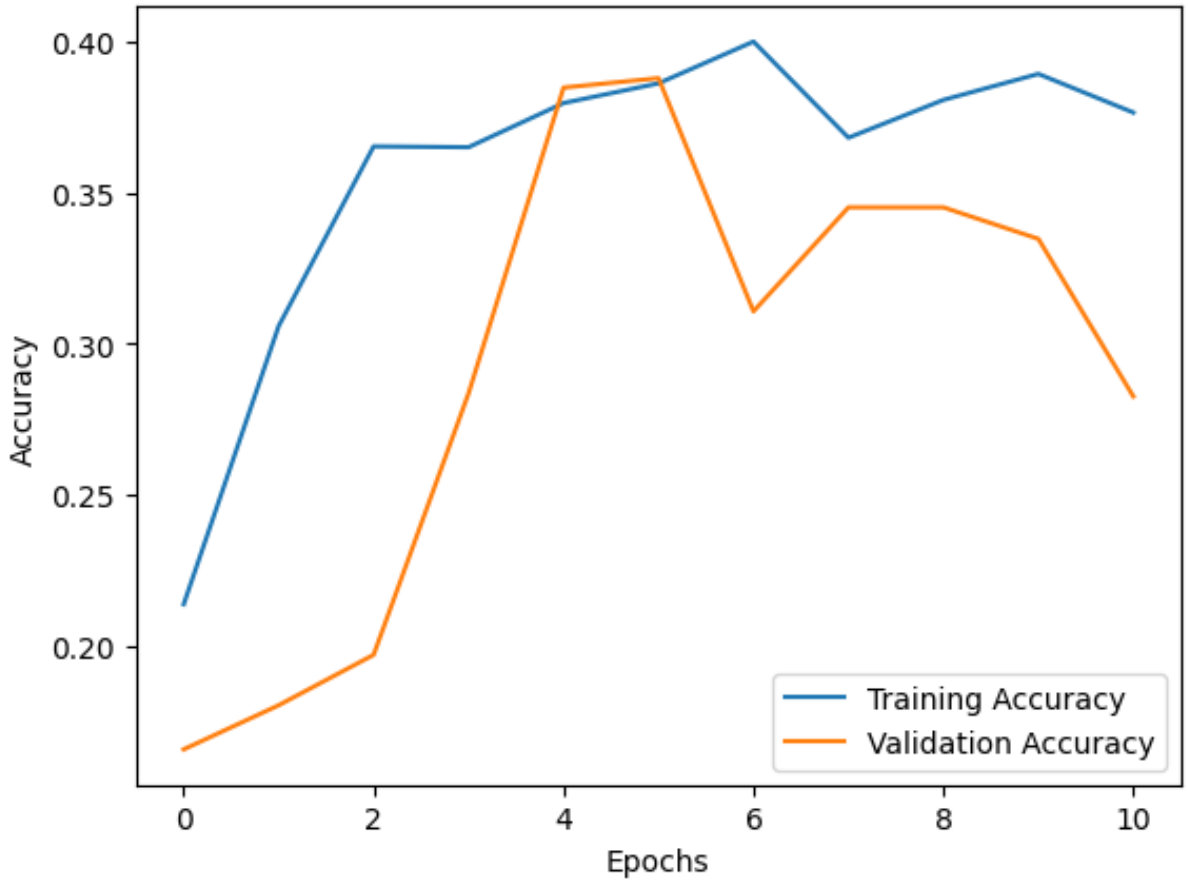
*** Evaluer les performances avec les stratégies limitant l'overfitting si besoin ou la "'disparition"' du gradient**

```
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Dropout, BatchNormalization
from keras.callbacks import EarlyStopping
from keras.optimizers import Adam
```

```
from keras import regularizers

# Fonction pour tracer les courbes d'apprentissage
def plot_history(history):
    # Tracer la perte d'entra nement et de validation
    plt.plot(history.history['loss'], label='Training Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

    # Tracer la précision d'entra nement et de validation
    plt.plot(history.history['accuracy'], label='Training Accuracy')
    plt.plot(history.history['val_accuracy'], label='Validation
       Accuracy')
    plt.xlabel('Epochs')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.show()

# Définir le modèle
model = Sequential()
model.add(Dense(128, input_dim=X_train.shape[1], activation='relu',
   kernel_initializer='he_normal', kernel_regularizer=regularizers.l2
   (0.01)))
model.add(Dropout(0.5))
model.add(BatchNormalization())
model.add(Dense(64, activation='relu', kernel_initializer='he_normal')
   )
model.add(Dropout(0.5))
model.add(Dense(32, activation='relu', kernel_initializer='he_normal')
   )
model.add(Dense(10, activation='softmax'))  # Nombre de classes

# Compiler le modèle
model.compile(optimizer=Adam(), loss='categorical_crossentropy',
   metrics=['accuracy'])

# Callback d'arrêt précoce
early_stopping = EarlyStopping(monitor='val_loss', patience=5, restore
   _best_weights=True)

# Entra ner le modèle avec des données augmentées
history = model.fit(X_augmented, y_augmented, epochs=50, batch_size
   =32, validation_split=0.2, callbacks=[early_stopping])

#  valuer  les performances avec des courbes d'apprentissage
plot_history(history)
```

On obtient les résultats suivants :





*** Evaluer les performances par transfert Learning**

```
import os
import shutil

# Chemin du dossier contenant les images
base_dir = 'C:/Users/tahas/Desktop/ML/Wang'

# Chemin o   les sous-dossiers des classes seront créés
output_dir = 'C:/Users/tahas/Desktop/ML/WangOrganized'

# Correspondance des plages d'images aux classes
label_mapping = {
    (0, 99): '0',
    (100, 199): '1',
    (200, 299): '2',
    (300, 399): '3',
    (400, 499): '4',
    (500, 599): '5',
    (600, 699): '6',
    (700, 799): '7',
    (800, 899): '8',
    (900, 999): '9'
}
```

```
# Créer les sous-dossiers pour chaque classe
for _, label in label_mapping.items():
    os.makedirs(os.path.join(output_dir, label), exist_ok=True)

# Parcourir les fichiers et les déplacer dans les bons sous-dossiers
for filename in os.listdir(base_dir):
    if filename.endswith('.jpg'):
        image_num = int(filename.split('.')[0])  # Obtenir le numéro
           de l'image
        for (start, end), label in label_mapping.items():
            if start <= image_num <= end:
                src_path = os.path.join(base_dir, filename)
                dst_path = os.path.join(output_dir, label, filename)
                shutil.move(src_path, dst_path)  # Déplacer le fichier
                break

print("Organisation terminée. Les fichiers sont maintenant dans des
   sous-dossiers.")

image_dir = 'C:/Users/tahas/Desktop/ML/WangOrganized'

train_generator = datagen.flow_from_directory(
    image_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='training'
)

val_generator = datagen.flow_from_directory(
    image_dir,
    target_size=(224, 224),
    batch_size=32,
    class_mode='categorical',
    subset='validation'
)

print("Classes détectées : ", train_generator.class_indices)

from tensorflow.keras.applications import VGG16
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Flatten, Dropout
from tensorflow.keras.optimizers import Adam

# Charger le modèle VGG16 sans sa dernière couche dense
base_model = VGG16(weights='imagenet', include_top=False, input_shape
   =(224, 224, 3))

# Geler les couches du modèle pré-entra né
for layer in base_model.layers:
```

```
    layer.trainable = False

# Ajouter des couches spécifiques pour votre classification
model = Sequential([
    base_model,
    Flatten(),
    Dense(256, activation='relu'),
    Dropout(0.5),
    Dense(10, activation='softmax')  # 10 classes
])

# Compiler le modèle
model.compile(optimizer=Adam(learning_rate=0.0001),
              loss='categorical_crossentropy',
              metrics=['accuracy'])

history = model.fit(
    train_generator,
    epochs=2,
    validation_data=val_generator
)
```

Les images initialement stockées dans un seul dossier ont été réorganisées en sous-dossiers correspondant aux classes (`0` à `9`) dans un nouveau répertoire (`WangOrganized`) en fonction de leur numéro de fichier. À l'aide de `ImageDataGenerator`, les images ont été redimensionnées à 224×224 pixels, puis divisées en ensembles d'entraînement (800 images) et de validation (200 images). Un modèle basé sur **VGG16** pré-entraîné sur *ImageNet* a été utilisé, avec ses couches gelées pour préserver les poids existants, et des couches supplémentaires ont été ajoutées : une couche de `Flatten`, une dense de 256 neurones avec `ReLU`, une `Dropout` (0.5), et une couche de sortie `softmax` adaptée aux 10 classes. Compilé avec l'optimiseur `Adam` (taux d'apprentissage = 0.0001) et la fonction de perte `categorical_crossentropy`, le modèle a été entraîné pendant 2 époques, atteignant une précision d'entraînement de 83,41 % et une précision de validation de 87 %, avec une diminution significative de la perte, indiquant une convergence rapide et prometteuse.