Neural-Network - Algorithme du Perceptron

☰ TOC

Objectifs du TP

- Comprendre le fonctionnement du perceptron simple
- Implémenter l'algorithme du perceptron
- Analyser les limites du perceptron sur des problèmes non-linéairement séparables
- Appliquer le perceptron à des données réelles

Livrables Attendus

Code

- Implémentation complète de la classe `PerceptronSimple`
- Implémentation de la classe `PerceptronMultiClasse`
- Scripts de test et de visualisation

Rapport

1. **Introduction** : Contexte et objectifs
2. **Méthodes** : Description des algorithmes implémentés
3. **Résultats** :
   - Tests sur fonctions logiques
   - Analyse de convergence
   - Évaluation sur données réelles
4. **Discussion** :
   - Limites du perceptron
   - Cas d'usage appropriés
5. **Conclusion** : Synthèse des apprentissages

Visualisations

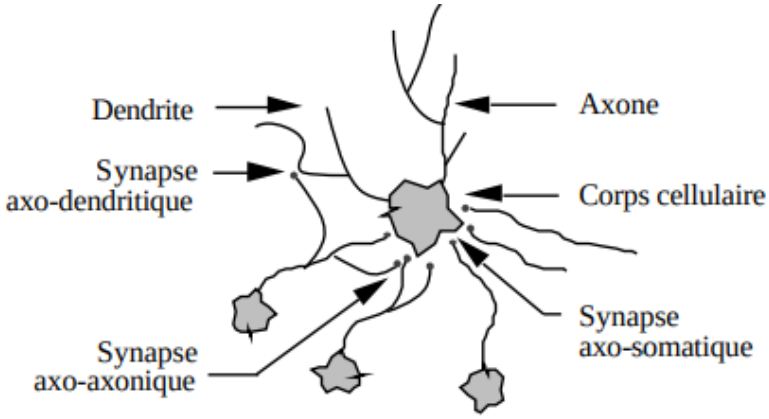
- Graphiques de convergence
- Visualisation des droites de séparation
- Matrices de confusion
- Comparaisons de performances

Partie 1 : Le Perceptron Simple (un seul neurone)

1.1 Introduction théorique

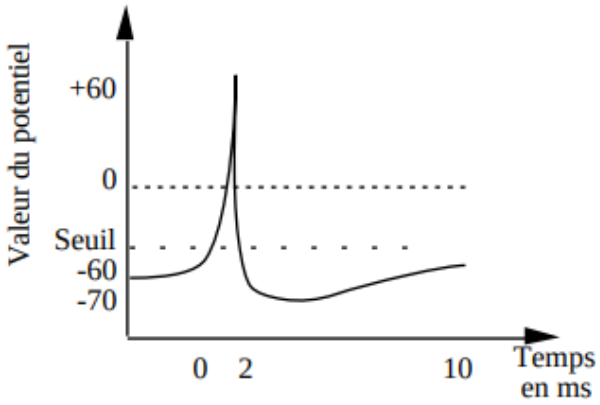
Le perceptron est un modèle de neurone artificiel introduit par Frank Rosenblatt en 1958. Il constitue l'une des premières tentatives de modélisation du fonctionnement des neurones biologiques dans le but de résoudre des problèmes de classification binaire. Plus précisément, le perceptron permet de traiter des données linéairement séparables, c'est-à-dire que les classes peuvent être séparées par une droite (ou un hyperplan dans un espace de dimension supérieure).

Ce modèle s'inspire du neurone biologique, une cellule nerveuse spécialisée dans la transmission de l'information. Un neurone se compose d'un corps cellulaire (ou soma) contenant le noyau, entouré de prolongements appelés dendrites. Ces dendrites, souvent très nombreuses, forment ce que l'on appelle une arborisation dendritique ou chevelure dendritique, et permettent de capter les signaux provenant d'autres neurones ou de l'environnement. L'information ainsi reçue est intégrée par le soma, puis transmise le long d'un prolongement unique nommé axone. Celui-ci conduit le signal nerveux jusqu'aux synapses, points de jonction avec les dendrites d'autres neurones. Ces synapses ne permettent pas un contact direct : un infime espace, appelé fente synaptique, sépare les neurones, mesurant seulement quelques dizaines d'angstroms (10^{-9} m).



Un neurone avec son arborisation dendritique (Clause Touzet 1992)

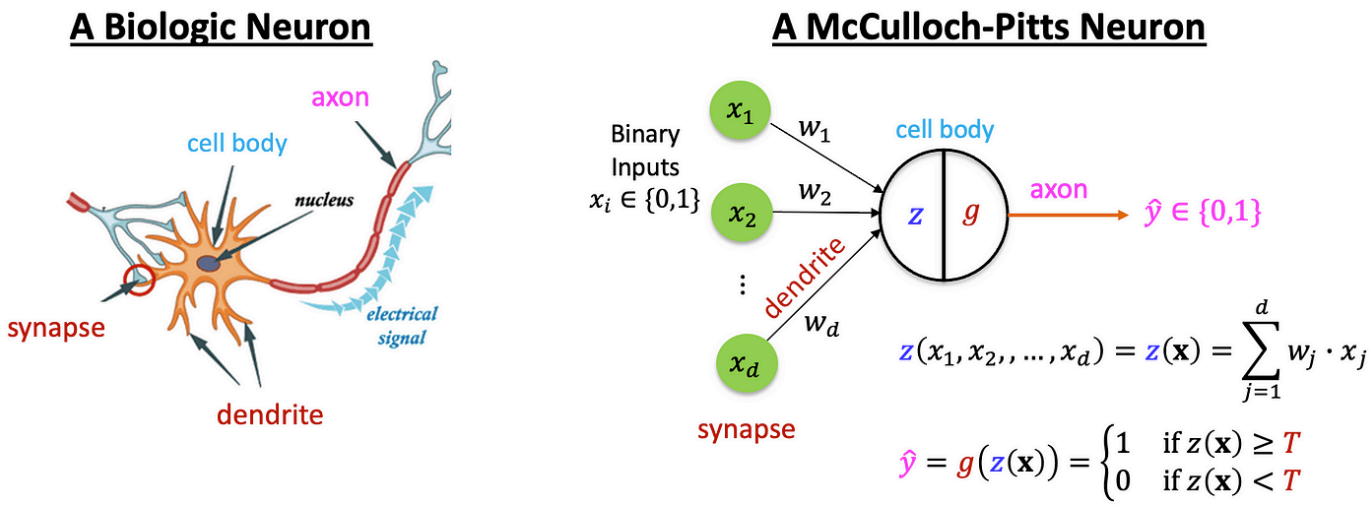
Lorsqu'un neurone est stimulé par un signal provenant de l'environnement ou d'un autre neurone, il peut entrer dans un état dit d'excitation. Si ce changement devient assez important — c'est-à-dire s'il dépasse un certain seuil (autour de -55 mV) — alors le neurone réagit fortement : il envoie un signal électrique appelé potentiel d'action (ou spkie). Ce signal se traduit par un changement rapide de la tension, qui devient soudainement positif, comme on le voit sur la figure suivante. Ce signal est alors transmis aux neurones adjacents.



Spike d'activation du neurone(Clause Touzet 1992)

1.2 Fonctionnement du perceptron monocouche

Le **perceptron monocouche** reprend le schéma général du neurone biologique, mais de manière simplifiée et formalisée mathématiquement. Dans ce modèle, les **dendrites** sont remplacées par un **vecteur d'entrée** noté $x = [x_1, x_2, \ldots, x_n]$, qui représente les signaux reçus par le neurone artificiel. À chaque entrée est associé un **poids synaptique** $w = [w_1, w_2, \ldots, w_n]$ qui joue un rôle similaire à celui des connexions synaptiques dans le cerveau : il module l'importance de chaque information reçue.



L'unique neurone de McCulloch et Pitts en 1943

1.3 Combinaison linéaire des signaux

Avant de produire une réponse, le perceptron effectue une **sommation pondérée** des signaux reçus, à laquelle on ajoute un **biais b**, une constante qui permet de décaler le seuil de déclenchement du neurone.. Cette étape correspond, en biologie, à l'intégration des signaux au niveau du **soma**, qui évalue si le seuil d'excitation est atteint. Le calcul effectué est le suivant :

$$z = \sum_i w_i x_i + b = w \cdot x + b$$

où :

- w · x est le produit scalaire entre les poids et les entrées
- b est le biais,
- z est le résultat de cette pré-activation, équivalent au potentiel mesuré dans un neurone biologique.

1.4 Fonction d'activation : le seuil d'excitation

Dans le neurone biologique, un **potentiel d'action** est déclenché si la tension dépasse un certain seuil (comme vu précédemment autour de −55 mV). C'est pour cela que le modèle de 1943 proposais une fonction d'activation contenant un seuil, noté qu'il a simplement été décidé de déplacé le seuil `T` qui étais dans la fonction d'activation vers la somme pondéré, c'est mathématiquement identique mais simplifie la mise en pratique. De manière analogue, dans le perceptron, la **fonction d'activation** décide si le signal sera transmis à la sortie ou non.

Le perceptron classique utilise une **fonction seuil** ou **fonction de Heaviside**, définie ainsi :

```
f(z) = 1  si z ≥ 0
       0  si z < 0
```

Cette fonction modélise le **comportement tout ou rien** du neurone biologique : soit le seuil est atteint et le neurone « s'active », soit il ne l'est pas et il reste silencieux. La **sortie finale du neurone artificiel** est donc :

$$z = \sum x_i w_i + b$$

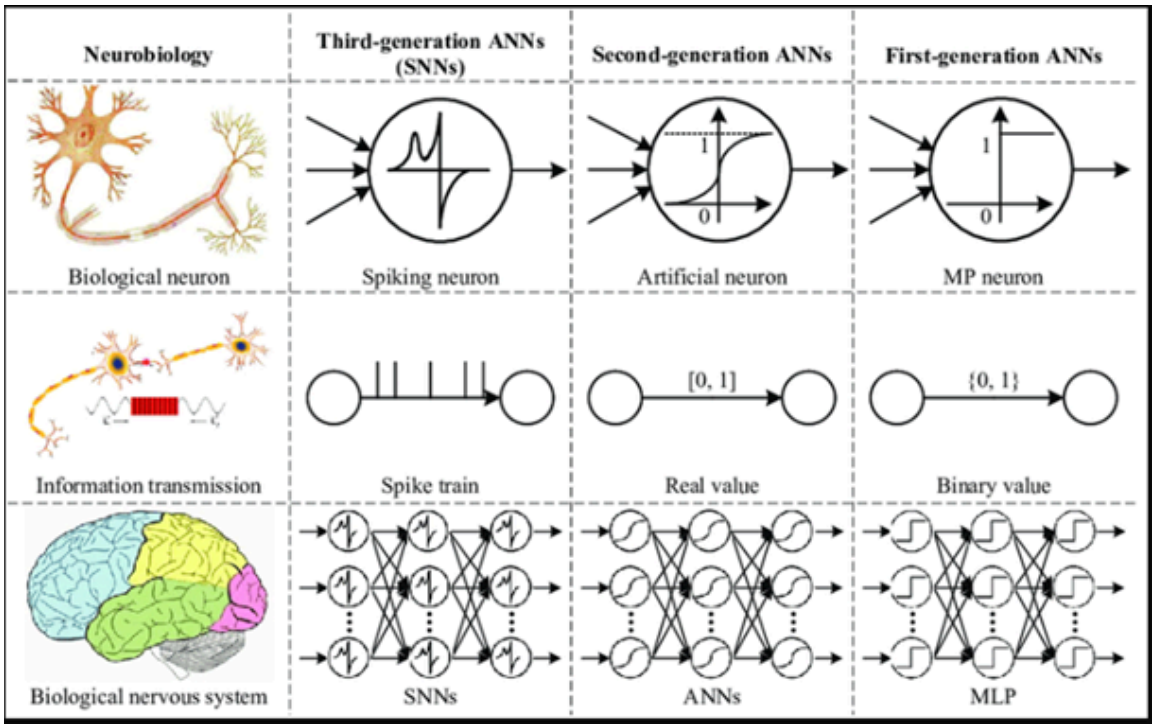
$$y = f(z)$$

La sortie y, qui vaut 0 ou 1, représente la **décision du perceptron** face à la donnée d'entrée. Cela correspond en fait a un **classifieur binaire élémentaire**, capable de séparer linéairement deux classes. Dit autrement, le perceptron a **appris** une droite qui sépare deux **ensembles**. Les valeurs **0, 1** nous **indiquent** de quel côté de la droite nous **nous situons**.

Les fonctions d'activation sont essentielles car elles introduisent la non-linéarité nécessaire dans les réseaux de neurones . Sans elles, un réseau multicouche ne serait qu'une succession de transformations linéaires équivalente à une seule couche

1.5 Exercice pratique : exploration des fonctions d'activation

Avant de construire le perceptron complet, il est utile de mieux comprendre le rôle et le comportement des **fonctions d'activation**, qui jouent un rôle essentiel dans la transmission du signal, à l'image du **seuil d'excitation** dans le neurone biologique. En gros on essaye de répliqué le comportement d'activation d'un neurone de façon mathématique. La figure ci-dessous montre l'évolution des conceptes. Dans le cadre de ce cours, nous resterons dans les deux premières générations. Pour aller plus loin je vous consseil de lire l'article suivant.



Spike d'activation du neurone (Xiangwen Wang et all 2020)

Dans cet exercice, vous allez **implémenter et visualiser différentes fonctions d'activation**, ainsi que leurs **dérivées**, en utilisant les bibliothèques **NumPy** et **Matplotlib**. Cela vous permettra d'observer leur forme, leur domaine de sortie, et leurs propriétés analytiques. Mais aussi de prendre en main ces bibliothèques.

Exercice 1 – Implémentation des fonctions d'activation

Implémentez et tracez les courbes suivantes, ainsi que leurs dérivées :

Fonction	Nom	Équation
Heaviside	Fonction échelon	$H(x) = \{0 \quad \text{si} \quad x < 0 \quad 1 \quad \text{si} \quad x \geq 0$
Tanh	Tangente hyperbolique	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
Sigmoïd	Fonction logistique	$\sigma(x) = \frac{1}{1 + e^{-x}}$
ReLU	Rectified Linear Unit	$\text{ReLU}(x) = \{0 \quad \text{si } x < 0 \; x \quad \text{si } x \geq 0$
Leaky ReLU	ReLU « fuyante »	$\text{Leaky ReLU}(x) = \{\alpha x \quad \text{si } x < 0 \; x \quad \text{si } x \geq 0$

Pour les dérivées, rappellez vous de vos cours de lycée. Vous pouvez utiliser le code suivant comme point de départ, mais l'idéal serait de créer un fichier Python dédié à cette classe et de mettre en place une structure de code propre (voir la section 'Prise en main'). Une meilleure architecture est tout à fait possible (voir cours de COO), notamment avec une classe abstraite (ActivationFunction) et de l'héritage (Heaviside, ReLU, etc). Ce n'est cependant pas la vocation principale de ce cours. Adaptez le code en fonction de vos capacités.

```
import numpy as np

class ActivationFunction:
```

```
    def __init__(self, name, alpha=0.01):
        self.name = name.lower()
        self.alpha = alpha  # Pour Leaky ReLU

    def apply(self, z):
        if self.name == "heaviside":
            return 0 # TODO
        elif self.name == "sigmoid":
            retur 0 # TODO
        elif self.name == "tanh":
            return 0 # TODO
        elif self.name == "relu":
            return 0 # TODO
        elif self.name == "leaky_relu":
            return 0 # TODO
        else:
            raise ValueError(f"Activation '{self.name}' non reconnue.")

    def derivative(self, z):
        if self.name == "heaviside":
            # La dérivée de Heaviside est la distribution de Dirac
            return  0 # TODO
        elif self.name == "sigmoid":
            return  0 # TODO
        elif self.name == "tanh":
            return  0 # TODO
        elif self.name == "relu":
            return  0 # TODO
        elif self.name == "leaky_relu":
            return  0 # TODO
        else:
            raise ValueError(f"Dérivée de '{self.name}' non définie.")
```

> Attention, utilisé numpy pour les calcules et la prise en main

Exemple pour l'utilisation :

```
import matplotlib.pyplot as plt
z = np.linspace(-10, 10, 100)

for name in ['heaviside', 'sigmoid']:
    act = ActivationFunction(name)
    g = act.apply(z)

    # ajouter la dérivé sur le graphique

    plt.figure()
    plt.plot(z, z)
    plt.plot(z, g)
    plt.savefig('figures/' + name + '.png')
```

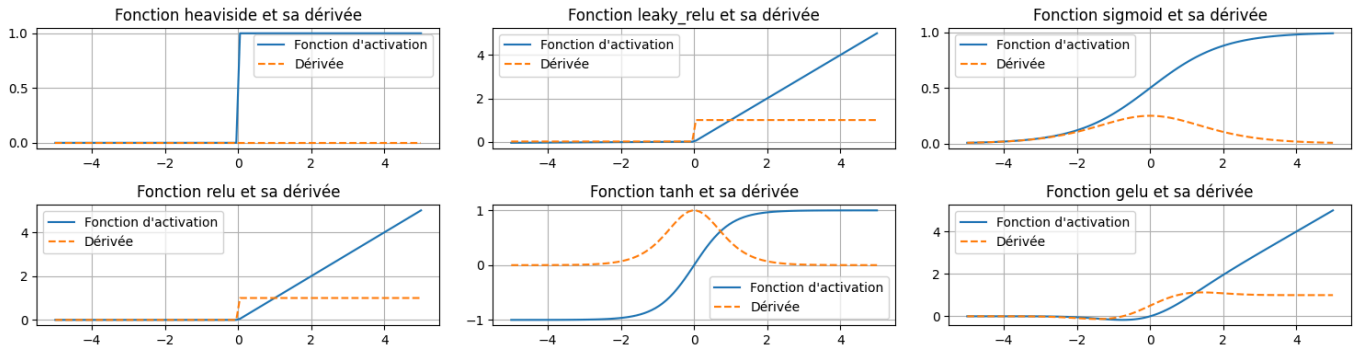
Propriétés à observer

Fonction	Domaine de sortie	Dérivabilité	Année
Heaviside	{0, 1}	Non	1900
Tanh	(−1, 1)	Oui	1950
Sigmoïde	(0, 1)	Oui	1958
ReLU	[0, +∞)	Presque partout	2010
Leaky ReLU	(−∞, +∞)	Presque partout	2015

Exercice 2 : Questions d'analyse :

1. Pourquoi la fonction de **`Heaviside`** pose-t-elle problème pour l'apprentissage par gradient ?
2. Dans quels cas utiliser **`sigmoid`** vs **`tanh`** ?
3. Pourquoi **`ReLU`** est-elle si populaire dans les réseaux profonds ?
4. Quel est l'avantage du **`Leaky ReLU`** ?

Résultat attendu



Partie 2 : Apprentissage du perceptron

Comme les connexions synaptiques se renforcent ou s'affaiblissent dans un cerveau vivant en fonction de l'expérience (phénomène appelé **plasticité synaptique**), le perceptron ajuste ses **poids** et **biais** en fonction des erreurs commises lors de l'entraînement. On parle d'**apprentissage**.

Les différentes approches d'apprentissage (supervisé, non-supervisé, semi-supervisé, par renforcement, etc.) se distinguent principalement par la manière dont on définit et calcule l'erreur pendant l'entraînement. Chaque paradigme implique :

- Un type de feedback différent (labels, récompenses, absence de supervision explicite)
- Des mécanismes d'optimisation adaptés à ces signaux
- Des mesures d'erreur spécifiques (erreur quadratique, divergence de probabilités, reward shaping, etc.)

Dans le cadre de ce TP et pour la suite, nous resterons dans le domaine de l'apprentissage supervisé.

2.1 Règle d'apprentissage du perceptron

À chaque itération, le perceptron reçoit un **exemple d'apprentissage** composé :

- d'un **vecteur d'entrée** :

$$x = [x_1, x_2, \ldots, x_n]$$

- d'une **étiquette attendue** :

$$y \in 0, 1$$

Le fonctionnement se déroule en trois étapes :

1. **Propagation du signal** Comme dans un neurone biologique, le perceptron combine les entrées en une **sommation pondérée**, puis applique une **fonction d'activation**. Cela donne la sortie prédite :

$$\hat{y} = f(w \cdot x + b)$$

1. **Vérification de la réponse** On compare la sortie $\hat{y}$ avec la sortie attendue y. Si la prédiction est correcte, rien ne change. Si elle est incorrecte ($\hat{y} \neq y$), alors une mise à jour est nécessaire.
2. **Mise à jour des poids et du biais** Le perceptron ajuste ses paramètres comme suit :

$$e = y - \hat{y}$$

$$w \leftarrow w + \eta \cdot e \cdot x$$

$$b \leftarrow b + \eta \cdot e$$

où η (appelé **taux d'apprentissage**) est un petit nombre positif qui contrôle la vitesse d'ajustement.

Cette règle permet de corriger les poids dans la bonne direction pour que, progressivement, le perceptron apprenne à classer correctement les exemples. Il s'agit d'une version simplifiée de la manière dont les connexions entre neurones peuvent se renforcer ou s'affaiblir dans le cerveau en fonction de l'expérience.

Exercice 3 : Questions d'analyse :

1. Que se passe-t-il si η est trop grand ?
2. Et s'il est trop petit ?
3. Existe-t-il une valeur idéale de η ?
4. Peut-on faire varier η au cours du temps ?
5. Quelle stratégie pouvez vous imaginer ?

> Vous pouvez y répondre plus tard en testant ce qui se passe lors de l'apprentissage et en observant les fluctuations de la fonction de perte (graphiques).

2.2 Implémentation du perceptron simple

Dans cette section, nous allons donner vie au modèle théorique du perceptron à travers une première implémentation en Python, en utilisant les bibliothèques NumPy (pour le calcul numérique) et Matplotlib (pour la visualisation). Ce travail pratique permettra de consolider les notions vues jusqu'ici : les poids synaptiques, la fonction d'activation, la propagation de l'information et la règle d'apprentissage.

À travers cet exercice, vous programmerez un perceptron monocouche (un seul neurone), capable de traiter des données linéairement séparables. C'est une première étape avant d'envisager des architectures plus complexes (perceptron multicouches).

Exercice 4 :

Implémentez la classe `PerceptronSimple` suivante :

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

class PerceptronSimple:
    def __init__(self, learning_rate=0.1):
        self.learning_rate = learning_rate
        self.weights = None
        self.bias = None

    def fit(self, X, y, max_epochs=100):
        """
        Entraîne le perceptron
        X: matrice des entrées (n_samples, n_features)
```

```
        y: vecteur des sorties désirées (n_samples,)
        """
        # Initialisation les poids et le biais
        self.weights = np.random.randn(X.shape[1])
        self.bias = 0.0

        for e in tqdm(max_epochs):
            for b in range(X.shape[0]):
                x = X[i]
                y_true = y[i]
                # TODO: Implémenter l'algorithme d'apprentissage
                # Possible d'optimiser d'aventage numpy
                y_pred = 0

    def predict(self, X):
        """Prédit les sorties pour les entrées X"""
        y_pred = np.zeros(X.shape[0])

        for b in range(X.shape[0]):
            # TODO: Calculer les prédictions
            x = X[i] # (n_features,)
            y_pred[i] = 0

        return y_pred

    def score(self, X, y):
        """Calcule l'accuracy"""
        predictions = self.predict(X)
        return np.mean(predictions == y)
```

Une fois cette classe complétée, vous pourrez l'utiliser pour entraîner un perceptron sur des jeux de données jouets (comme les portes logiques AND / OR) et visualiser son comportement. Noté que vous pouvez tester votre classe sur les premières données de l'exercice suivant qui devrais converger facilement.

2.3 Test sur les fonctions logiques

Maintenant que vous avez implémenté un perceptron simple, il est temps de le confronter à des problèmes concrets — en commençant par des situations très classiques : les fonctions logiques booléennes. Ces cas simples permettent de vérifier le bon fonctionnement de l'algorithme et de visualiser la capacité du perceptron à séparer des classes.

> Une erreur a été volontairement introduite. Faites attention au domaine des fonctions utilisées.

Exercice 5 :

```
# Données pour la fonction AND
X_and = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_and = np.array([-1, -1, -1, 1])  # -1 pour False, 1 pour True

# Données pour la fonction OR
X_or = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
```

```
y_or = np.array([-1, 1, 1, 1])
```

Pour chaque cas :

1. Combien d'époques sont nécessaires pour converger ?
2. Visualisez la droite de séparation trouvée
3. Le perceptron converge-t-il toujours vers la même solution ? (ie les mêmes poids)

Exercice 6 :

Testez votre perceptron avec la fonction logique suivante :

```
# Données pour la fonction XOR
X_xor = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y_xor = np.array([-1, 1, 1, -1])
```

- Quelles sont vos constatations ?
- Quel lien peut-on faire avec la notion de séparabilité linéaire évoquée plus tôt dans le cours ?

Cette étape est souvent un point charnière dans la compréhension des limites du perceptron : c'est à ce moment que l'on réalise qu'un seul neurone ne suffit pas pour traiter certains cas, ce qui justifie l'introduction de réseaux multicouches dans la suite du TP.

Partie 3 : Générateur de Données et Visualisation

3.1 Générateur de données linéairement séparables

Jusqu'ici, vous avez testé votre perceptron sur des cas simples et discrets comme les fonctions logiques AND, OR et XOR. Pour aller plus loin, il est intéressant de tester votre modèle sur des **nuages de points dans le plan** — des jeux de données plus réalistes, mais que l'on contrôle totalement.

L'objectif est de produire des **données synthétiques** appartenant à deux classes bien séparées par une droite. Ce genre de configuration correspond exactement à ce que sait résoudre un perceptron monocouche, car les classes sont **linéairement séparables**. En variant les paramètres (nombre de points, bruit...), on pourra aussi observer l'influence de l'environnement sur l'apprentissage.

Exercice 7 :

Créez un générateur de données aléatoires linéairement séparables :

```
def generer_donnees_separables(n_points=100, noise=0.1):
    """
    Génère deux classes de points linéairement séparables
    """
    np.random.seed(42)
    # TODO: Générer deux nuages de points séparables
    # Classe 1: points autour de (2, 2)
    # Classe 2: points autour de (-2, -2)
    return X, y
```

> Ce générateur vous permettra de créer des situations d'entraînement variées à volonté, utiles pour tester la robustesse du perceptron. Ou plustard de l'architecture de votre réseau de neurone.

Une bonne habitude à prendre est de **visualiser vos données**, notamment pour vérifier que la séparation est bien possible, et que l'apprentissage a convergé vers une solution correcte. Voici une fonction de visualisation. Elle permet, en plus des points, d'afficher la **droite de décision** trouvée par le perceptron :

```
def visualiser_donnees(X, y, w=None, b=None, title="Données"):
    """
    Visualise les données et optionnellement la droite de séparation
    """
    plt.figure(figsize=(8, 6))
    # Afficher les points
    mask_pos = (y == 1)
    plt.scatter(X[mask_pos, 0], X[mask_pos, 1], c='blue', marker='+', s=100, label='Classe +1')
    plt.scatter(X[~mask_pos, 0], X[~mask_pos, 1], c='red', marker='*', s=100, label='Classe -1')
    # Afficher la droite de séparation si fournie
    if w is not None and b is not None:
        # TODO: Tracer la droite w·x + b = 0
        pass
    plt.xlabel('x1')
    plt.ylabel('x2')
    plt.legend()
    plt.title(title)
    plt.grid(True, alpha=0.3)
    plt.show()
```

La droite tracée correspond à l'endroit où le modèle considère qu'un changement de classe doit se produire. Lancez plusieurs fois votre programme, que constatez vous sur la droite apprise ?

3.2 Analyse de la convergence

Exercice 8

Dans le cerveau, l'apprentissage est un processus graduel, où les connexions synaptiques sont renforcées ou affaiblies petit à petit en fonction des expériences. De la même manière, dans un perceptron, les **ajustements des poids** sont régis par un paramètre fondamental : le **taux d'apprentissage**, noté η (eta).

Ce paramètre contrôle **l'amplitude des modifications** effectuées sur les poids à chaque erreur. Sa valeur influence fortement la dynamique de l'entraînement.

```
def analyser_convergence(X, y, learning_rates=[0.0001, 0.001, 0.01, 0.1, 1.0, 3.0, 10.0]):
    """
    Analyse la convergence pour différents taux d'apprentissage
    """
    plt.figure(figsize=(12, 8))
    for i, lr in enumerate(learning_rates):
        # TODO: Entraîner le perceptron avec ce taux d'apprentissage
        # TODO: Enregistrer l'évolution de l'erreur à chaque époque
        # TODO: Tracer les courbes de convergence
        pass
    plt.xlabel('Époque')
    plt.ylabel("Nombre d'erreurs")
    plt.title("Convergence pour différents taux d'apprentissage")
    plt.legend()
    plt.grid(True, alpha=0.3)
```

```
    plt.show()
```

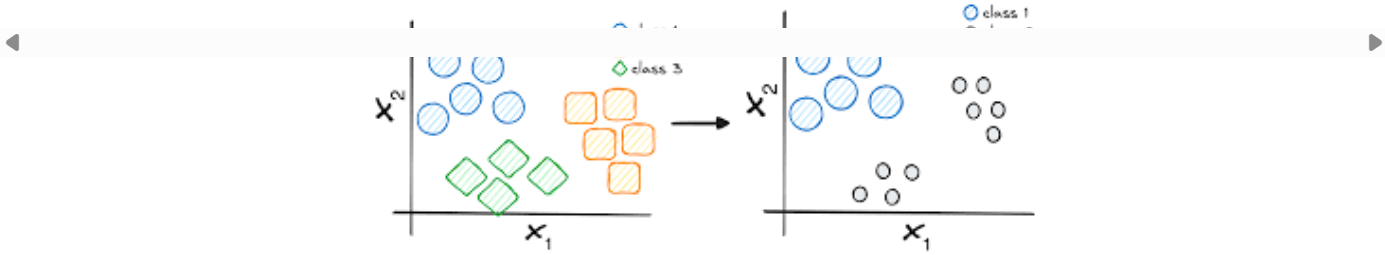
Les constatations devront renforcer, confirmer ou invalider vos réponses de l'exercice 3.

1. Quel comportement observez-vous lorsque η est très petit ?
2. Que se passe-t-il lorsque η est trop grand ?
3. Existe-t-il un η optimal dans votre cas ?
4. Comment la structure des données (dispersion, bruit...) peut-elle interagir avec η ?

Partie 4 : Classification Multi-Classes

Jusqu'à présent, nous avons travaillé avec des problèmes de **classification binaire**, où le perceptron doit distinguer entre deux classes seulement. Dans le monde réel, de nombreux problèmes impliquent plus de deux catégories : reconnaissance de chiffres manuscrits (0 à 9), classification d'espèces de fleurs, diagnostic médical multi-catégoriel, etc.

Le perceptron simple ne peut naturellement traiter que des problèmes binaires, puisqu'il ne produit qu'une seule sortie (0 ou 1, -1 ou +1). Pour étendre ses capacités à la classification multi-classes, plusieurs stratégies existent. La plus intuitive et largement utilisée est l'approche **"Un contre Tous"** (One-vs-All ou One-vs-Rest).



4.1 Stratégie "Un contre Tous"

Le principe de cette stratégie est simple mais efficace : pour un problème à **K classes**, on entraîne **K perceptrons différents**. Chaque perceptron spécialisé apprend à distinguer **une classe particulière** de **toutes les autres classes réunies**.

Par exemple, pour classifier des iris en 3 espèces (Setosa, Versicolor, Virginica) : - **Perceptron 1** : Setosa vs (Versicolor + Virginica) - **Perceptron 2** : Versicolor vs (Setosa + Virginica)
- **Perceptron 3** : Virginica vs (Setosa + Versicolor)

Au moment de la prédiction, on interroge tous les perceptrons et on choisit la classe correspondant au perceptron le plus "confiant" dans sa réponse.

Cette approche présente plusieurs avantages : - **Simplicité conceptuelle** : on réutilise directement l'algorithme binaire - **Parallélisation possible** : chaque perceptron peut être entraîné indépendamment - **Modularité** : on peut facilement ajouter ou retirer des classes

Cependant, elle présente aussi quelques inconvénients : - **Déséquilibre des classes** : chaque perceptron voit sa classe positive minoritaire face à toutes les autres - **Zones d'ambiguïté** : certaines régions peuvent être revendiquées par plusieurs perceptrons ou par aucun

Exercice 9 : Implémentation du perceptron multi-classes

Implémentez un perceptron multi-classes en utilisant la stratégie "Un contre Tous" :

```
import numpy as np
from tqdm import tqdm


class PerceptronMultiClasse:
    def __init__(self, learning_rate=0.1):
        self.learning_rate = learning_rate
        self.perceptrons = {}
        self.classes = None
```

```python
    def fit(self, X, y, max_epochs=100):
        """
        Entraîne un perceptron par classe (stratégie un-contre-tous)
        """
        self.classes = np.unique(y)

        for classe in tqdm(self.classes, desc="Entraînement des perceptrons"):
            # TODO: Créer un problème binaire pour cette classe
            # Transformer y en problème binaire : classe courante vs toutes les autres

            # TODO: Entraîner un perceptron pour ce problème binaire
            perceptron = PerceptronSimple(learning_rate=self.learning_rate)

            # Stocker le perceptron entraîné
            self.perceptrons[classe] = perceptron

    def predict(self, X):
        """Prédit en utilisant le vote des perceptrons"""
        if not self.perceptrons:
            raise ValueError("Le modèle n'a pas été entraîné. Appelez fit() d'abord.")

        # TODO: Calculer les scores de tous les perceptrons
        scores = np.zeros((X.shape[0], len(self.classes)))

        for i, classe in enumerate(self.classes):
            # Calculer la sommation pondérée (avant fonction d'activation)
            # pour obtenir un score de confiance

        # TODO: Retourner la classe avec le score maximum
        predicted_indices = np.argmax(scores, axis=1)
        return self.classes[predicted_indices]

    def predict_proba(self, X):
        """Retourne les scores de confiance pour chaque classe"""
        if not self.perceptrons:
            raise ValueError("Le modèle n'a pas été entraîné.")

        scores = np.zeros((X.shape[0], len(self.classes)))

        for i, classe in enumerate(self.classes):
            perceptron = self.perceptrons[classe]
            raw_scores = X.dot(perceptron.weights) + perceptron.bias
            scores[:, i] = raw_scores

        return scores
```

Questions d'analyse :

1. **Cohérence des prédictions** : Que se passe-t-il si plusieurs perceptrons prédisent positivement pour le même exemple ?
2. **Gestion des ambiguïtés** : Comment gérer le cas où aucun perceptron ne prédit positivement ?

3. **Équilibrage** : Comment l'approche "Un contre Tous" gère-t-elle le déséquilibre naturel qu'elle crée ?

4.2 Données réelles – Dataset Iris (simplifié)

Le **dataset Iris** est un grand classique en apprentissage automatique, introduit par le statisticien Ronald Fisher en 1936. Il contient des mesures de 150 fleurs d'iris réparties en 3 espèces (Setosa, Versicolor, Virginica), avec 4 caractéristiques par fleur : longueur et largeur des sépales, longueur et largeur des pétales.

Ce jeu de données est particulièrement intéressant car : – Il présente un **problème de classification à 3 classes** – Les classes ne sont **pas toutes linéairement séparables** (défi intéressant pour le perceptron) – Il est suffisamment simple pour être visualisé en 2D – Il permet de tester la robustesse de notre approche multi-classes

Exercice 10 : Chargement et préparation des données

Téléchargez et préparez les données Iris pour nos expériences :

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt

def charger_donnees_iris_binaire():
    """
    Charge le dataset Iris en version binaire (2 classes) pour commencer
    """
    # Chargement du dataset Iris
    iris = load_iris()

    # Ne garder que 2 features pour la visualisation
    X = iris.data[:, [0, 2]]  # longueur des sépales et longueur des pétales
    y = iris.target

    # Ne garder que 2 classes pour commencer (Setosa vs Versicolor)
    mask = y < 2
    X_binary = X[mask]
    y_binary = y[mask]
    y_binary = 2 * y_binary - 1  # Convertir 0,1 en -1,1

    return X_binary, y_binary

def charger_donnees_iris_complete():
    """
    Charge le dataset Iris complet avec les 3 classes
    """
    iris = load_iris()
    X = iris.data[:, [0, 2]]  # longueur des sépales et longueur des pétales
    y = iris.target

    return X, y, iris.target_names

def visualiser_iris(X, y, target_names=None, title="Dataset Iris"):
    """
    Visualise le dataset Iris avec ses différentes classes
    """
    plt.figure(figsize=(10, 8))
```

```
    # Couleurs pour chaque classe
    colors = ['red', 'blue', 'green']
    markers = ['*', '+', 'o']

    for i in range(len(np.unique(y))):
        mask = (y == i)
        label = target_names[i] if target_names else f'Classe {i}'
        plt.scatter(X[mask, 0], X[mask, 1],
                    c=colors[i], marker=markers[i], s=100,
                    label=label, alpha=0.7)

    plt.xlabel('Longueur des sépales (cm)')
    plt.ylabel('Longueur des pétales (cm)')
    plt.title(title)
    plt.legend()
    plt.grid(True, alpha=0.3)
    plt.show()

# Test de la fonction
if __name__ == "__main__":
    # Version binaire d'abord
    X_bin, y_bin = charger_donnees_iris_binaire()
    print(f"Données binaires : {X_bin.shape[0]} échantillons, {X_bin.shape[1]} features")

    # Version complète
    X_full, y_full, noms = charger_donnees_iris_complete()
    print(f"Données complètes : {X_full.shape[0]} échantillons, {len(np.unique(y_full))} classes")

    # Visualisation
    visualiser_iris(X_full, y_full, noms)
```

Observations importantes :

En examinant la visualisation, vous devriez constater que : - **Setosa** (rouge) est clairement séparée des deux autres classes - **Versicolor** (bleu) et **Virginica** (vert) se chevauchent partiellement - Il existe des zones d'ambiguïté où la classification sera difficile

4.3 Évaluation rigoureuse des performances

Dans le domaine de l'apprentissage automatique, une évaluation rigoureuse nécessite de séparer les données en plusieurs ensembles distincts, chacun ayant un rôle spécifique :

1. **Ensemble d'entraînement (Train)** : utilisé pour ajuster les paramètres du modèle (poids et biais)
2. **Ensemble de validation (Validation)** : utilisé pour ajuster les hyperparamètres et surveiller le sur-apprentissage
3. **Ensemble de test (Test)** : utilisé une seule fois, après l'entraînement, pour évaluer les performances finales

Cette séparation est cruciale pour éviter le **sur-apprentissage** (overfitting) et obtenir une estimation réaliste des performances de généralisation.

Exercice 11 : Évaluation complète du perceptron multi-classes

Implémentez une fonction d'évaluation complète avec métriques et visualisations :

```python
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns

def evaluer_perceptron_multiclasse(X, y, target_names=None, test_size=0.3, val_size=0.5):
    """
    Évalue le perceptron multi-classes avec une méthodologie rigoureuse
    """

    # TODO: Diviser les données en train/validation/test
    # Première division : train+val / test
    X_temp, X_test, y_temp, y_test = train_test_split(
        X, y, test_size=test_size, random_state=42, stratify=y
    )

    # Deuxième division : train / validation
    X_train, X_val, y_train, y_val = train_test_split(
        X_temp, y_temp, test_size=val_size, random_state=42, stratify=y_temp
    )

    print(f"Répartition des données :")
    print(f"  - Entraînement : {X_train.shape[0]} échantillons")
    print(f"  - Validation   : {X_val.shape[0]} échantillons")
    print(f"  - Test         : {X_test.shape[0]} échantillons")

    # TODO: Entraîner le perceptron multi-classes
    perceptron_mc = PerceptronMultiClasse(learning_rate=0.1)
    perceptron_mc.fit(X_train, y_train, max_epochs=100)

    # L'ensemble de validation devrait être passé en paramètre de la fonction fit.
    # Ainsi, à chaque epoch, vous pourrez tester les paramètres du modèle sur cet ensemble.
    # À la fin de l'apprentissage, ...pprentissage, on restaure les paramètres dont le fit est le meilleur sur l'enssembl

    # TODO: Calculer les prédictions sur tous les ensembles
    y_train_pred = perceptron_mc.predict(X_train)
    y_val_pred = perceptron_mc.predict(X_val)
    y_test_pred = perceptron_mc.predict(X_test)

    # TODO: Calculer les métriques (accuracy, matrice de confusion)
    accuracy_train = np.mean(y_train_pred == y_train)
    accuracy_val = np.mean(y_val_pred == y_val)
    accuracy_test = np.mean(y_test_pred == y_test)

    print(f"\nPerformances :")
    print(f"  - Accuracy train      : {accuracy_train:.3f}")
    print(f"  - Accuracy validation : {accuracy_val:.3f}")
    print(f"  - Accuracy test       : {accuracy_test:.3f}")
```

Questions de réflexion et d'nalyse

1. **Convergence** : Dans quelles conditions le perceptron est-il garanti de converger ?
2. **Initialisation** : L'initialisation des poids influence-t-elle la solution finale ?
3. **Taux d'apprentissage** : Comment choisir le taux d'apprentissage optimal ?
4. **Généralisation** : Comment évaluer la capacité de généralisation du perceptron ?
5. **XOR Revisité** : Proposez des solutions pour résoudre le problème XOR
6. **Données bruitées** : Comment le perceptron se comporte-t-il avec des données bruitées ?
7. **Classes déséquilibrées** : Que se passe-t-il si une classe est très minoritaire ?
8. **Normalisation** : Faut-il normaliser les données avant l'entraînement ?