Apprentissage automatique
Chapitre 3 - Réseau de neurones artificiels

Halim Djerroud

LISV
Laboratoire d'ingénierie
des systèmes de Versailles

UVSQ
université PARIS-SACLAY

révision : 0.1

Plan du cours et déroulement

Plan du cours

1. Introduction.
2. Les Données.
3. Apprentissage supervisé et non supervisé.
4. **Les réseaux de neurones.**

Déroulement

- 18 heures de cours, 6 séances de 3 heures.
- Deux contrôles continus (QCM).
- Un projet à faire en binôme.
- Un examen écrit.

Plan

- Historique
- Perceptron
- Perceptron multi couch
- L'apprentissage d'un reseau de neurone
- La descente de gradiant
- Introduction à Keras

Historique

Historique des Réseaux de Neurones Artificiel (RNA)

Plan du chapitre :

1. 1943 - Neurones artificiel :Warren McCulloch & Walter Pitts
2. 1949 - L'apprentissage (Règle de Hebb) : Donald Hebb
3. 1957 - Perceptron : Frank Rosenblatt
4. 1960 - Déception
5. 1980 - Connexionnisme vs Computationnalisme
6. 2010 - Deep learning : (Yann Lecun)

Principe de base des neurones logiques

- Années 1940 : Warren McCulloch (neurophysiologiste) et Walter Pitts (logicien) posent les bases des réseaux de neurones.
- Article fondateur : "A Logical Calculus of Ideas Immanent in Nervous Activity" (1943).
- Objectif : Modéliser les processus neuronaux à l'aide de la logique mathématique.
- Inspiration : Le fonctionnement des neurones biologiques.
- Hypothèse : Les neurones se comportent comme des dispositifs de calcul logique.
- Approche : Combiner des signaux d'entrée pour produire une sortie binaire (0 ou 1).

Définition

$$\text{Sortie} = \begin{cases} 1 & \text{si } \sum w_i x_i \geq \text{seuil}\Theta, \\ 0 & \text{sinon.} \end{cases}$$

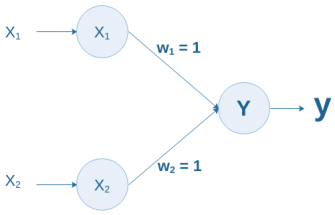
- x_i : les entrées binaires.
- w_i : les poids associés aux entrées.

Neurone logique : AND

- On observe la table de vérité de *AND*
- On fixe $W_1 = 1$ et $W_2 = 1$
- On fixe le seuil $\Theta = 2$

$(0,0), y_{in} = x_1 W_1 + X2_W 2 = 0 \times 1 + 0 \times 1 = 0$
$(0,1), y_{in} = x_1 W_1 + X2_W 2 = 0 \times 1 + 1 \times 1 = 1$
$(1,0), y_{in} = x_1 W_1 + X2_W 2 = 1 \times 1 + 0 \times 1 = 1$
$(1,1), y_{in} = x_1 W_1 + X2_W 2 = 1 \times 1 + 1 \times 1 = 2$

seuil Θ = 2



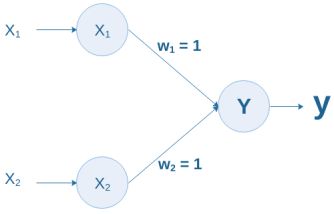
X_1	X_2	y = X_1 and X_2
0	0	0
0	1	0
1	0	0
1	1	1

LISV Laboratoire d'ingénierie des systèmes de Versailles — UVSQ université PARIS-SACLAY

Neurone logique : OR

- On observe la table de vérité de *OR*
- On fixe $W_1 = 1$ et $W_2 = 1$
- On fixe le seuil $\Theta = 1$

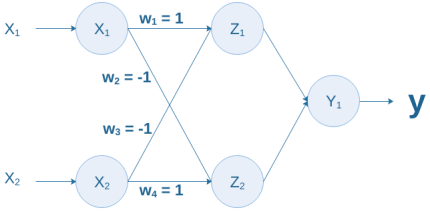
$(0,0), y_{in} = x_1 W_1 + X2_W 2 = 0 \times 1 + 0 \times 1 = 0$
$(0,1), y_{in} = x_1 W_1 + X2_W 2 = 0 \times 1 + 1 \times 1 = 1$
$(1,0), y_{in} = x_1 W_1 + X2_W 2 = 1 \times 1 + 0 \times 1 = 1$
$(1,1), y_{in} = x_1 W_1 + X2_W 2 = 1 \times 1 + 1 \times 1 = 2$
seuil Θ = 1



X_1	X_2	y = X_1 or X_2
0	0	0
0	1	1
1	0	1
1	1	1

Neurone logique : XOR

- On observe la table de vérité de *XOR*:
  - $y = X_1\bar{X_2} + \bar{X_1}X_2$
  - $y = z_1 + z_2$



X_1	X_2	y = X_1 xor X_2
0	0	0
0	1	1
1	0	1
1	1	0

Limites des neurones logiques

- Incapacité à apprendre : Les poids et le seuil doivent être définis manuellement.
- Limitation aux problèmes linéairement séparables.
- Pas de prise en compte du temps ou des processus dynamiques.

Perceptron

Fonctionnement du perceptron

Plan du chapitre :

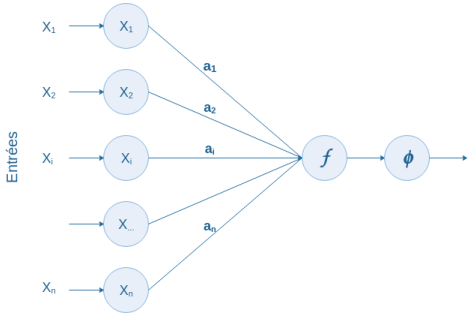
1. Résolution de problèmes linéairement séparables.

Introduction au Perceptron

- Le perceptron est l'un des premiers algorithmes d'apprentissage automatique.
- Développé par Frank Rosenblatt en 1958.
- Extension des concepts de McCulloch et Pitts pour inclure l'apprentissage.
- Objectif : Construire un modèle capable de classifier des données grâce à un algorithme d'apprentissage supervisé.
- Utilisé pour des tâches de classification linéaire.

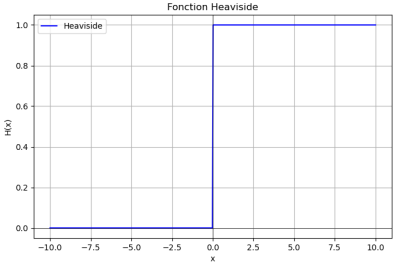
Structure du perceptron

- Entrées : $x_1, x_2, \ldots, x_n$.
- Poids : $a_1, a_2, \ldots, a_n$.
- Fonction linaire :
  $f(x_1, \ldots, x_n) = a_1x_1 + a_2x_2 + \ldots + a_nx_n$.
- Fonction d'activation (exemple : seuil).
- sortie $\phi(a_1x_1 + a_2x_2 + \ldots + a_nx_n)$



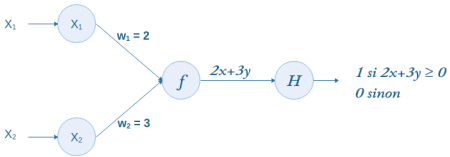
Fonction seuil (Step Function / Heaviside)

- Sortie binaire : $f(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{sinon} \end{cases}$



Exemple

- Utilisation d'un perceptron comme une fonction.



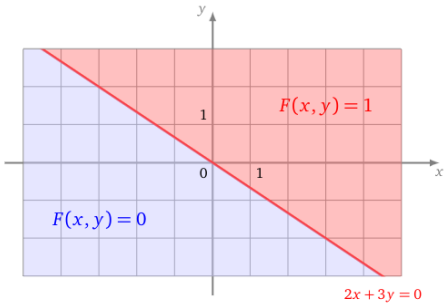
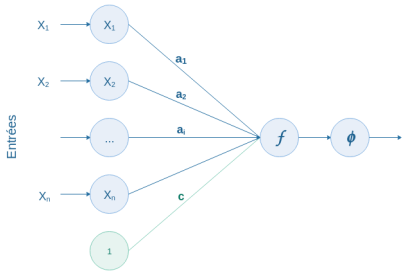


Figure: Source: deepmath

Perceptron affine

- Utilisation d'un perceptron comme une fonction.



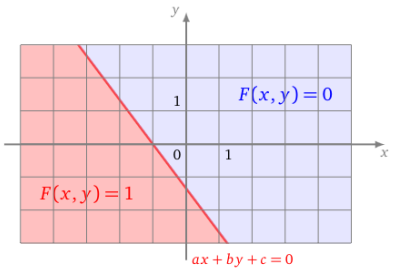


Figure: Source: deepmath

Les Fonctions d'Activation

- Les fonctions d'activation déterminent la sortie d'un neurone en fonction de sa somme pondérée.
- Différents types de fonctions sont utilisées selon les besoins :
  - Fonction seuil (Heaviside).
  - Fonction sigmoïde.
  - Fonction tanh.
  - Fonction ReLU.
  - Fonction softmax.

Fonction d'Activation: seuil (Heaviside)

1. Fonction seuil (Step Function / Heaviside)

- Sortie binaire : $f(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{sinon} \end{cases}$
- Utilisée dans le perceptron classique.

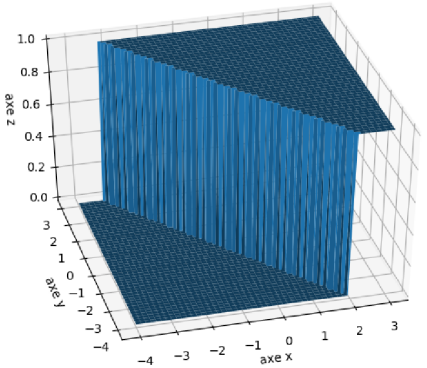


Figure: Source : deepmath

Fonction d'Activation : sigmoïde

- $f(x) = \frac{1}{1+e^{-x}}$
- Sortie continue entre 0 et 1.
- Utile pour les problèmes de classification probabiliste.

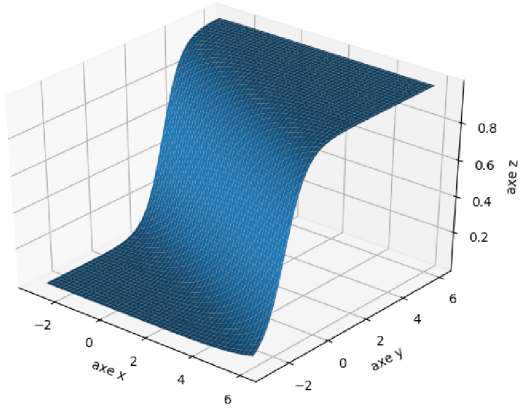


Figure: Source : deepmath

Fonctions d'Activation : tanh

- $f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- Sortie continue entre -1 et 1.
- Centre les données autour de zéro.

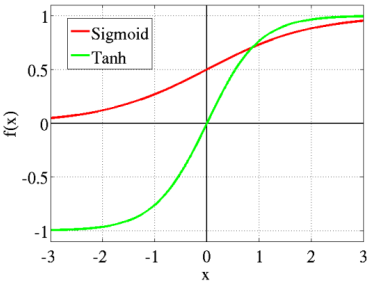


Figure: Source : web

Fonction d'Activation : ReLU (Rectified Linear Unit)

- $f(x) = \max(0, x)$
- Rapide à calculer et résout le problème des gradients qui disparaissent.

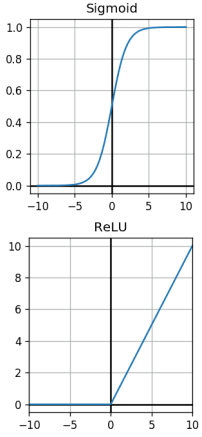


Figure: Source : web

Fonction d'Activation : softmax

- Utilisée pour la classification multi-classe.
- $f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$
- Transforme les sorties en probabilités.

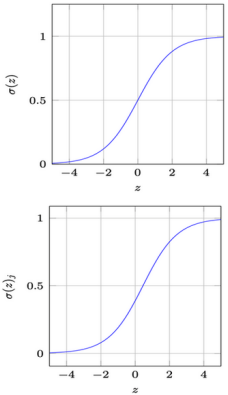


Figure: Source : web

Algorithme d'apprentissage

Étapes principales :

1. Initialiser les poids a_i à des petites valeurs aléatoires.
2. Pour chaque exemple d'entraînement (x, y) :
   - Calculer $y_{\text{prédit}} = \text{activation}(\sum_{i=1}^{n} a_i x_i)$.
   - Mettre à jour les poids :
   $$a_i \leftarrow a_i + \eta(y - y_{\text{prédit}})x_i$$
3. Répéter jusqu'à convergence ou nombre d'itérations fixé.

Introduction aux fonctions de perte

- Une fonction de perte mesure l'écart entre la sortie prédite $\hat{y}$ et la sortie réelle y.
- Objectif : Minimiser cette perte pour améliorer les performances du modèle.
- Les fonctions de perte sont essentielles dans les algorithmes d'apprentissage supervisé.

Formule générale

$$\mathcal{L}(y, \hat{y}) = \text{Une mesure d'erreur entre } y \text{ et } \hat{y}.$$

Exemples de fonctions de perte

- **Erreur quadratique moyenne (MSE)** :

$$\mathcal{L}_{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

Utilisée pour les problèmes de régression.

- **Entropie croisée (Cross-Entropy) / Log Loss** :

$$\mathcal{L}_{CE} = -\frac{1}{n}\sum_{i=1}^{n}\Big(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\Big)$$

Utilisée pour les problèmes de classification binaire.

- **Hinge Loss** :

$$\mathcal{L}_{Hinge} = \sum_{i=1}^{n}\max(0, 1 - y_i\hat{y}_i)$$

Utilisée pour les machines à vecteurs de support (SVM).

Importance des fonctions de perte

- Guident l'optimisation : L'objectif de l'entraînement est de minimiser la fonction de perte.
- Dépendent du type de problème : Régression, classification, etc.
- Impactent la convergence et les performances finales du modèle.

Optimisation

- Algorithmes courants : Descente de gradient, Adam, RMSProp.
- Mise à jour des paramètres :

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

Visualisation des fonctions de perte

- La forme de la fonction de perte influence l'apprentissage.
- Visualiser les courbes de perte aide à comprendre les dynamiques d'optimisation.

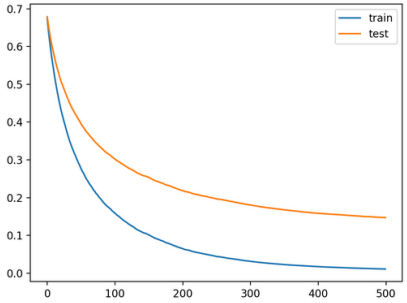


Figure: Source web.

Historique

Réseau de neurones (MLP : multilayer perceptron)

Plan du chapitre :

1. Réseau de neurones / Perceptron multicouche

Réseau de neurones

- Un réseau de neurones est la juxtaposition de plusieurs neurones, regroupés par couches.

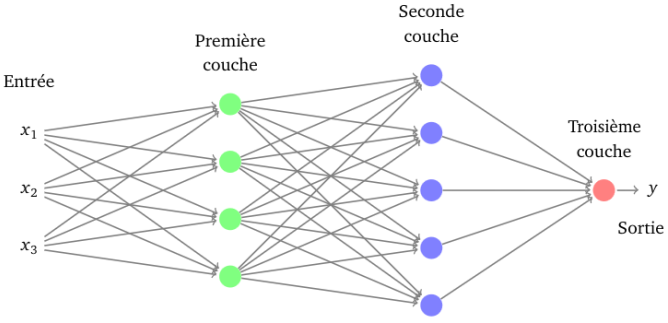


Figure: Source seepmath

Réseau de neurones

- Un réseau de neurones peut avoir plusieurs sorties.

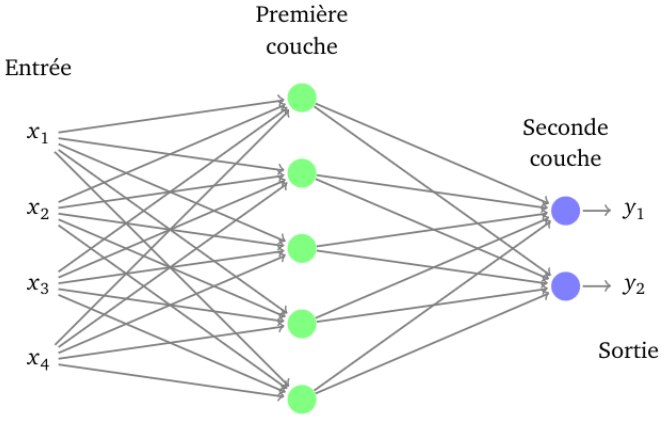


Figure: Source seepmath

Exemple 1

- Un réseau de neurones peut avoir plusieurs sorties.

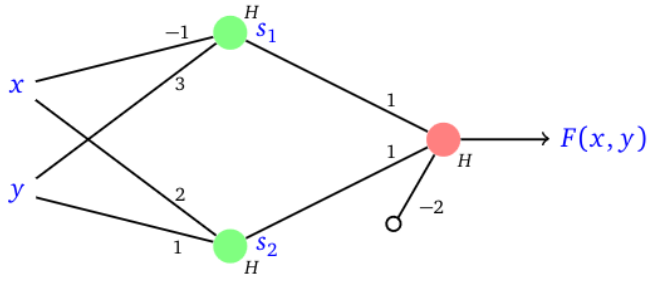


Figure: Source seepmath

Exemple 1

- Soit un réseau de neurones avec les poinds suivants :

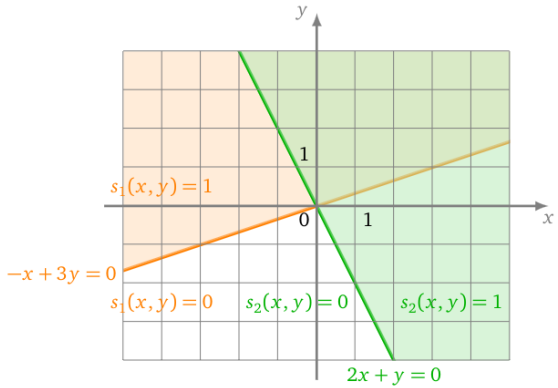


Figure: Source seepmath

Exemple 1

- On reconnaît dans le neurone de sortie un neurone qui réalise le ET .

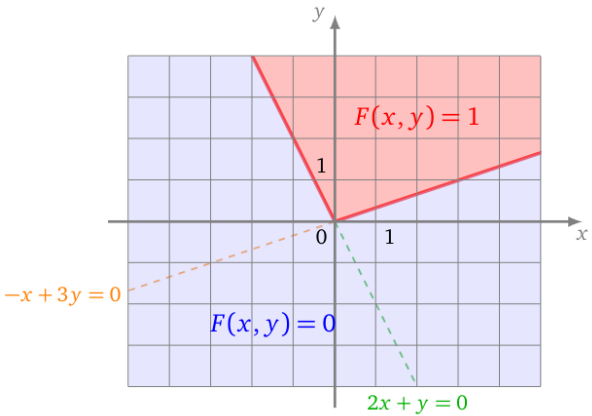


Figure: Source seepmath

Exemple 1

- Le même réseau avec un biais de -1 pour le neurone de sortie.

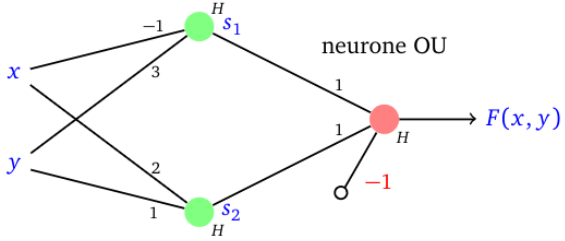


Figure: Source seepmath

Exemple 1

- On reconnaît dans le neurone de sortie un neurone qui réalise le OU .

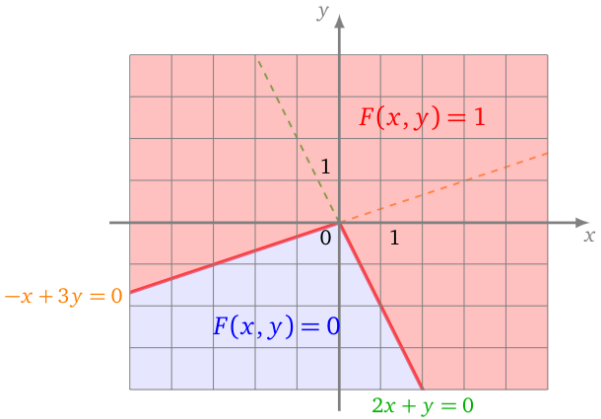


Figure: Source seepmath

Exercice 1

- Soit le réseau de neurones suivant (la fonction d'activation est H partout). Dessiner le graphe de la sortie.

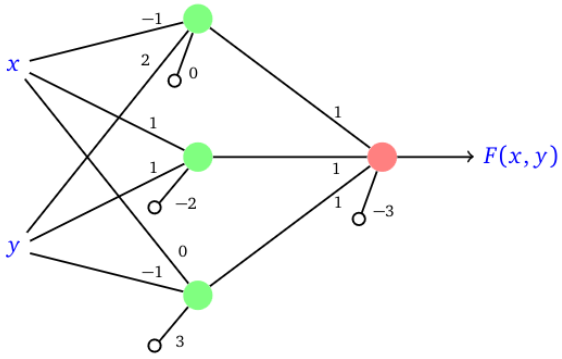


Figure: Source seepmath

Exercice 1 : solution

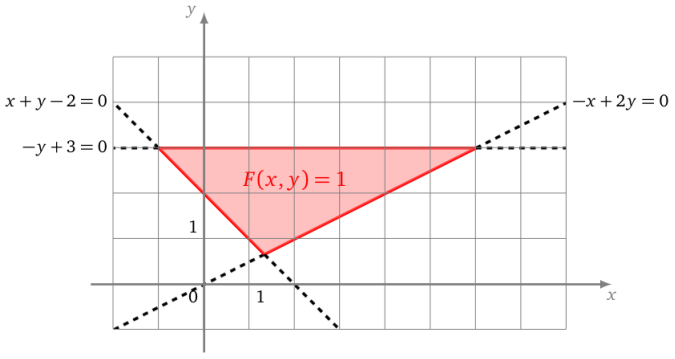


Figure: Source deepmath

Descente de gradient : la taille des données VS la taille de la mémoire

Mise à Jour des Poids dans un Réseau de Neurones

Plan du chapitre :

1. Descente de gradient stochastique (SGD)
2. Descente par mini-lots
3. Descente par batch

Introduction

- L'entraînement d'un réseau de neurones implique la mise à jour des poids pour minimiser l'erreur.
- **Plusieurs approches existent, basées sur** :
  - La taille des données utilisées pour calculer l'erreur.
  - La fréquence des mises à jour.
- **Trois principales méthodes** :
  1. Descente de gradient stochastique (SGD).
  2. Descente par mini-lots (mini-batch gradient descent).
  3. Descente par batch (batch gradient descent).

Lexique

- **Époch** : Un passage complet sur l'ensemble des données d'entraînement.
  - Exemple : Si l'ensemble de données contient 1 000 échantillons, une époch signifie que tous les 1 000 échantillons ont été utilisés une fois.
- **Itération** : Une mise à jour des poids après traitement d'un batch.
  - Nombre d'itérations par époch :

$$\text{Itérations par époch} = \frac{\text{Taille totale des données}}{\text{Batch size}}$$

- **Batch** : Un sous-ensemble des données utilisé pour une itération.
  - Taille du batch (`batch_size`) : Nombre d'échantillons dans chaque batch.

Exemple

Données :

- Taille des données : 1 000 échantillons.
- Batch size : 100.
- Nombre d'époques : 5.

Calculs :

- Itérations par époch : $\frac{1000}{100} = 10$.
- Total d'itérations : $10 \times 5 = 50$.

Processus :

- Une mise à jour des poids est effectuée après chaque itération.
- Une époch se termine après 10 itérations.
- Ce processus est répété pour 5 époques.

Descente de Gradient Stochastique (SGD)

- **Fréquence de mise à jour** : Après chaque échantillon individuel.
- **Étapes** :
  1. Un échantillon est passé dans le réseau.
  2. L'erreur est calculée.
  3. Les poids sont mis à jour immédiatement.
- **Avantages** :
  - Convergence rapide au début.
  - Bonne exploration de l'espace des solutions.
- **Inconvénients** :
  - Mise à jour bruyante.
  - Convergence moins stable.

Exemple: Descente de Gradient Stochastique

```
import ...
...

# Donnees fictives
X = np.random.rand(100, 10)  # 100 echantillons, 10 caracteristiques
y = np.random.rand(100, 1)   # 100 etiquettes

# Modele simple
model = Sequential([
    Dense(32, activation='relu', input_shape=(10,)),
    Dense(1, activation='linear')
])

# Compilation du modele
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
              loss='mse')

# Entrainement avec batch_size=1 (SGD)
model.fit(X, y, batch_size=1, epochs=10, verbose=1)
```

Descente par Mini-Lots (Mini-Batch Gradient Descent)

- **Fréquence de mise à jour** : Après chaque mini-lot (batch) de données.
- Étapes :
  1. Les données sont divisées en mini-lots.
  2. Chaque mini-lot passe dans le réseau.
  3. L'erreur est calculée sur le mini-lot.
  4. Les poids sont mis à jour après chaque mini-lot.
- **Avantages** :
  - Compromis entre stabilité et rapidité.
  - Exploite la parallélisation sur GPU.
- **Inconvénients** :
  - Nécessite un choix judicieux de la taille des mini-lots.

Exemple: Descente par Mini-Lots

```
# Compilation du modele
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
              loss='mse')

# Entrainement avec batch_size=16 (Mini-batch gradient descent)
model.fit(X, y, batch_size=16, epochs=10, verbose=1)
```

Descente par Batch (Batch Gradient Descent)

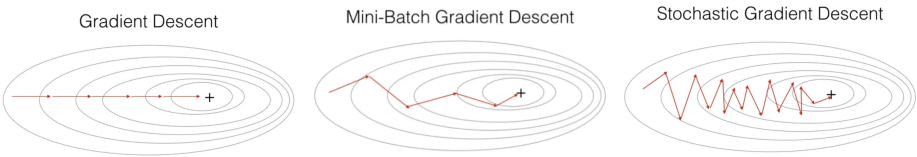
- **Fréquence de mise à jour** : Une fois après chaque epoch (ensemble des données).
- **Étapes** :
  1. L'ensemble des données passe dans le réseau.
  2. L'erreur totale est calculée.
  3. Les poids sont mis à jour une fois par epoch.
- **Avantages** :
  - Mise à jour stable.
  - Convergence plus précise dans certains cas.
- **Inconvénients** :
  - Coût élevé en mémoire.
  - Moins efficace pour les grands ensembles de données.

Exemple : Descente par Batch

```
# Compilation du modele
model.compile(optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
              loss='mse')

# Entrainement avec batch_size egal au nombre
# total d'echantillons (Batch gradient descent)
model.fit(X, y, batch_size=100, epochs=10, verbose=1)
```

Comparaison des Méthodes



Méthode	Fréquence de Mise à Jour	Avantages	Inconvénients
SGD	Après chaque échantillon	Rapide, exploration	Bruyante, instable
Mini-Batch	Après chaque mini-lot	Équilibre, parallélisation	Nécessite ajustement du batch
Batch	Après chaque epoch	Stable, précise	Lent, coûteux en mémoire

Choix de la méthode

- Le choix de la méthode dépend de :
  - La taille des données.
  - Les contraintes matérielles.
  - Les besoins en stabilité ou en vitesse.
- En pratique :
  - La descente par mini-lots est la méthode la plus utilisée.
  - Les frameworks modernes comme TensorFlow ou PyTorch permettent de configurer facilement la taille des lots.

Pourquoi utiliser Keras ?

- Keras est une API de haut niveau pour construire et entraîner des modèles de réseaux de neurones.
- Intégré dans TensorFlow.
- Points forts :
  - Simple d'utilisation.
  - Modulaire et extensible.
  - Large communauté de support.

Installation de Keras

- Pour commencer, installez TensorFlow (qui inclut Keras) :

```
pip install tensorflow
```

- Vérifiez l'installation :

```
python -c "import tensorflow as tf; print(tf.__version__)"
```

Les Bibliothèques

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
```

- TensorFlow inclut toutes les fonctions nécessaires pour utiliser Keras.
- Nous utiliserons un modèle séquentiel dans la suite de ce cours.

Qu'est-ce qu'une couche Dense ?

- Une couche Dense est une couche entièrement connectée.
- Chaque neurone est connecté à tous les neurones de la couche précédente.
- Formule mathématique : $y = \mathit{activation}(W \cdot x + b)$, où :
  - W : poids.
  - b : biais.
  - *activation* : fonction d'activation.
- Utilisée pour des tâches telles que la classification, la régression, etc.

Définir une couche Dense dans Keras

```
from tensorflow.keras.layers import Dense
# Exemple d'une couche Dense
layer = Dense(units=10, activation='relu')
# Ajout a un modele
from tensorflow.keras.models import Sequential
model = Sequential([
    Dense(units=64, activation='relu', input_shape=(100,)),
    Dense(units=10, activation='softmax')  # Couche de sortie
])
```

- 'units' : Nombre de neurones dans la couche.
- 'activation' : Fonction d'activation (e.g., ReLU, sigmoid, softmax).
- 'input_shape' : Forme des données en entrée (seulement pour la première couche).

Exemple pratique : Modèle Dense

```
...
X = np.random.random((100, 20))    # Donnees simples
y = np.random.randint(2, size=(100, 1))
# Modele Dense
model = Sequential([
    Dense(32, activation='relu', input_shape=(20,)),
    Dense(16, activation='relu'),
    Dense(1, activation='sigmoid')  # Couche de sortie pour classification binaire
])
# Compilation et entrainement
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])
model.fit(X, y, epochs=10, batch_size=8)
```

- Modèle Dense utilisé pour une classification binaire.
- Fonction d'activation finale : 'sigmoid' pour produire une probabilité.