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 Artificiels (RNA)

Plan du chapitre :

1. 1943 - Neurones artificiels : Warren McCulloch & Walter Pitts
2. 1949 - L'apprentissage (Règle de Hebb) : Donald Hebb
3. 1957 - Perceptron : Frank Rosenblatt
4. 1960 - Période de désillusion
5. 1980 - Connexionnisme vs Computationnalisme
6. 2010 - Deep Learning (Yann LeCun, Geoffrey Hinton, Yoshua Bengio)

Principe de base des neurones logiques (1943)

- 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 par la logique mathématique.
- Hypothèse : Les neurones se comportent comme des dispositifs de calcul logique.

Fonction de sortie

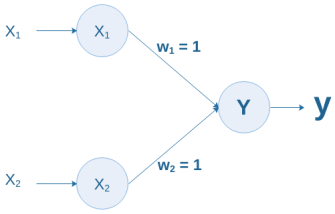
$$\text{Sortie} = \begin{cases} 1 & \text{si } \sum w_i x_i \geq \Theta \\ 0 & \text{sinon} \end{cases}$$

- x_i : entrées binaires
- w_i : poids des entrées
- Θ : seuil d'activation

Neurone logique : Porte AND

- Table de vérité de la porte *AND*
- Poids : $W_1 = 1$ et $W_2 = 1$
- Seuil : $\Theta = 2$

$(0,0) : y_{in} = 0 \times 1 + 0 \times 1 = 0 < \Theta \Rightarrow y = 0$
$(0,1) : y_{in} = 0 \times 1 + 1 \times 1 = 1 < \Theta \Rightarrow y = 0$
$(1,0) : y_{in} = 1 \times 1 + 0 \times 1 = 1 < \Theta \Rightarrow y = 0$
$(1,1) : y_{in} = 1 \times 1 + 1 \times 1 = 2 \geq \Theta \Rightarrow y = 1$

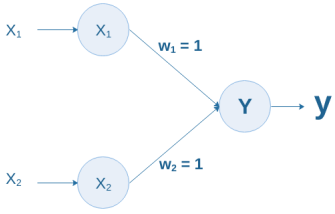


X_1	X_2	y = X_1 and X_2
0	0	0
0	1	0
1	0	0
1	1	1

Neurone logique : Porte OR

- Table de vérité de la porte *OR*
- Poids : $W_1 = 1$ et $W_2 = 1$
- Seuil : $\Theta = 1$

$(0,0) : y_{in} = 0 \times 1 + 0 \times 1 = 0 < \Theta \Rightarrow y = 0$
$(0,1) : y_{in} = 0 \times 1 + 1 \times 1 = 1 \geq \Theta \Rightarrow y = 1$
$(1,0) : y_{in} = 1 \times 1 + 0 \times 1 = 1 \geq \Theta \Rightarrow y = 1$
$(1,1) : y_{in} = 1 \times 1 + 1 \times 1 = 2 \geq \Theta \Rightarrow y = 1$

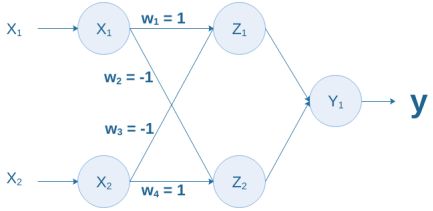


X_1	X_2	y = X_1 or X_2
0	0	0
0	1	1
1	0	1
1	1	1

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

Neurone logique : Porte XOR - Le problème

- Table de vérité de la porte *XOR* :
  - $y = X_1\overline{X_2} + \overline{X_1}X_2$
  - $y = z_1 + z_2$
- **Problème majeur :** Un seul neurone ne peut pas implémenter XOR !
- Solution : Nécessite un réseau multi-couches



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

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

Limites des neurones logiques de McCulloch-Pitts

- **Incapacité à apprendre :** Les poids et le seuil doivent être définis manuellement (pas d'algorithme d'apprentissage).
- **Limitation aux problèmes linéairement séparables :** Impossible de résoudre XOR avec un seul neurone.
- **Pas de prise en compte du temps :** Modèle statique, sans processus dynamiques.
- **Entrées binaires uniquement :** Ne gère pas les valeurs continues.

Impact historique

Ces limites ont conduit au développement du **Perceptron** (1957) par Frank Rosenblatt, qui introduit l'apprentissage automatique des poids.

Évolution : De McCulloch-Pitts au Deep Learning

- **1949 - Règle de Hebb :** ``Neurons that fire together, wire together'' - Premier principe d'apprentissage.
- **1957 - Perceptron :** Frank Rosenblatt introduit l'apprentissage supervisé.
- **1969 - Crise :** Minsky et Papert démontrent les limites du Perceptron (XOR).
- **1986 - Renaissance :** Rétropropagation du gradient (backpropagation).
- **2012 - Deep Learning :** AlexNet révolutionne la vision par ordinateur.

Perceptron

Le Perceptron de Rosenblatt (1957)

Objectifs du chapitre :

1. Comprendre le fonctionnement du perceptron
2. Résolution de problèmes linéairement séparables
3. Algorithme d'apprentissage du perceptron

Introduction au Perceptron

- Le perceptron est l'un des premiers algorithmes d'apprentissage automatique.
- Développé par **Frank Rosenblatt en 1957-1958**.
- Extension des concepts de McCulloch et Pitts avec ajout de l'**apprentissage automatique**.
- Objectif : Construire un modèle capable de classifier des données grâce à un algorithme d'apprentissage supervisé.
- Utilisé pour des tâches de **classification binaire linéaire**.

Innovation majeure

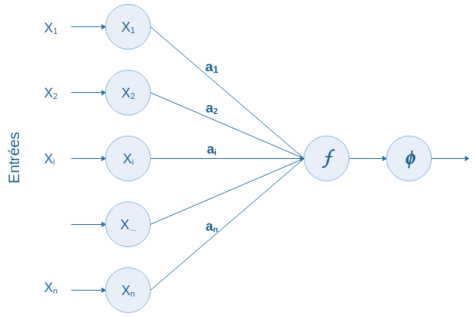
Contrairement au neurone de McCulloch-Pitts, le perceptron peut **apprendre automatiquement** ses poids par ajustement itératif.

Structure du perceptron

- **Entrées :** $x_1, x_2, \ldots, x_n$
- **Poids :** $w_1, w_2, \ldots, w_n$
- **Biais :** b (terme constant)
- **Somme pondérée :**

$$z = \sum_{i=1}^{n} w_i x_i + b$$

- **Fonction d'activation :** $\phi(z)$
- **Sortie :** $y = \phi(z)$

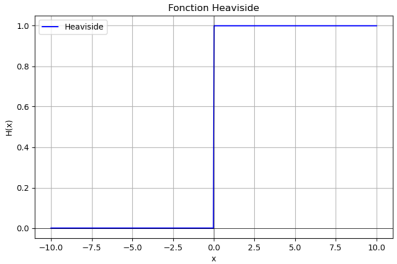


Fonction seuil (Step Function / Heaviside)

- Fonction d'activation binaire :

$$\phi(z) = \begin{cases} 1 & \text{si } z \geq 0 \\ 0 & \text{sinon} \end{cases}$$

- Aussi appelée **fonction de Heaviside**
- Permet une décision binaire (classe 0 ou 1)
- Discontinue en $z = 0$

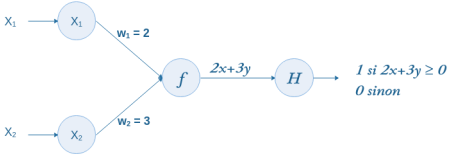


Exemple : Perceptron simple

- Perceptron à 2 entrées
- Poids : $w_1 = 1, w_2 = 1$
- Biais : $b = 0$
- Équation de décision :

$$z = x_1 + x_2$$

- Frontière : $x_1 + x_2 = 0$



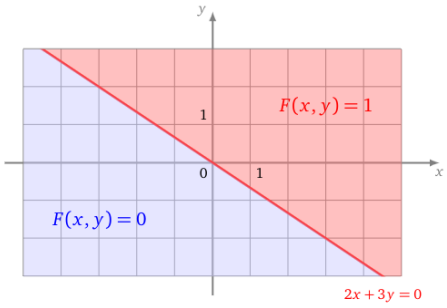


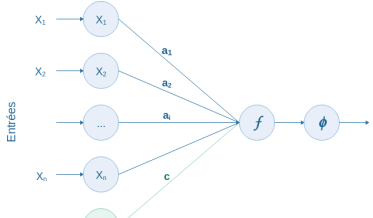
Figure: Source : deepmath

Perceptron affine (avec biais)

- Ajout d'un biais $b = -1$
- Poids : $w_1 = 1$, $w_2 = 1$
- Équation :

$$z = x_1 + x_2 - 1$$

- Frontière : $x_1 + x_2 = 1$
- Le biais **translate** la droite



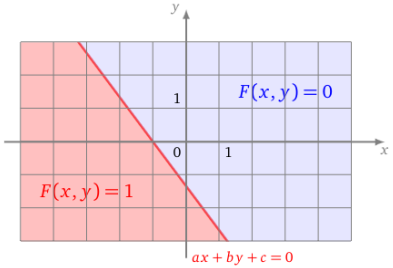


Figure: Source : deepmath

Frontière de décision

- Le perceptron définit une **frontière linéaire** dans l'espace des entrées
- Équation générale :

$$\sum_{i=1}^{n} w_i x_i + b = 0$$

- En 2D : une droite
- En 3D : un plan
- En dimension n : un hyperplan

Séparation linéaire

Le perceptron ne peut classifier correctement que des données **linéairement séparables**.

Algorithme d'apprentissage du perceptron

Principe : Ajuster les poids et le biais pour minimiser les erreurs de classification.

Règle de mise à jour

Pour chaque exemple mal classé $(x^{(i)}, y^{(i)})$:

$$w_j \leftarrow w_j + \eta \cdot (y^{(i)} - \hat{y}^{(i)}) \cdot x_j^{(i)}$$
$$b \leftarrow b + \eta \cdot (y^{(i)} - \hat{y}^{(i)})$$

où :

- η : taux d'apprentissage (learning rate)
- $y^{(i)}$: étiquette réelle
- $\hat{y}^{(i)}$: prédiction du perceptron

Algorithme du perceptron -- Pseudo-code

1. Initialiser les poids w et le biais b (souvent à 0 ou aléatoirement)
2. Répéter jusqu'à convergence (ou nombre max d'itérations) :
   - Pour chaque exemple d'entraînement $(x^{(i)}, y^{(i)})$:
     - Calculer la sortie :

$$\hat{y}^{(i)} = \phi\left(\sum_j w_j x_j^{(i)} + b\right)$$

     - Si $\hat{y}^{(i)} \neq y^{(i)}$, mettre à jour :

$$w_j \leftarrow w_j + \eta(y^{(i)} - \hat{y}^{(i)})x_j^{(i)}$$

$$b \leftarrow b + \eta(y^{(i)} - \hat{y}^{(i)})$$

Convergence garantie si les données sont linéairement séparables.

Exemple d'apprentissage - Porte AND

Données d'entraînement :

x_1	x_2	y (AND)
0	0	0
0	1	0
1	0	0
1	1	1

Après apprentissage :

- Poids trouvés : $w_1 = 1$, $w_2 = 1$
- Biais : $b = -1.5$
- Frontière : $x_1 + x_2 = 1.5$

Le perceptron a appris automatiquement à implémenter la porte AND !

Limites du perceptron

- **Problème majeur :** Ne peut résoudre que des problèmes **linéairement séparables**
- **Exemple classique - XOR :**

x_1	x_2	y (XOR)
0	0	0
0	1	1
1	0	1
1	1	0

- Aucune droite ne peut séparer les classes du XOR
- Cette limitation a été démontrée par **Minsky et Papert (1969)**

Solution

Utiliser un **perceptron multicouche** (MLP - Multi-Layer Perceptron)

Théorème de convergence du perceptron

Théorème (Rosenblatt, 1962)

Si les données d'entraînement sont linéairement séparables, alors l'algorithme du perceptron converge en un nombre fini d'itérations vers une solution qui classifie correctement toutes les données.

Implications :

- Garantie de convergence pour les problèmes linéairement séparables
- Pas de garantie sur le nombre d'itérations nécessaires
- Si les données ne sont pas linéairement séparables : l'algorithme ne converge pas

Comparaison : McCulloch-Pitts vs Perceptron

Critère	**McCulloch-Pitts (1943)**	**Perceptron (1957)**
Apprentissage	Non (poids fixés manuellement)	Oui (algorithme d'apprentissage)
Biais	Non explicite	Oui (terme b)
Type d'entrées	Binaires uniquement	Réelles
Applications	Portes logiques simples	Classification binaire
Limitation	Problèmes linéairement séparables	Problèmes linéairement séparables

Les Fonctions d'Activation

Définition

Les fonctions d'activation introduisent de la **non-linéarité** dans le réseau de neurones, permettant d'apprendre des relations complexes entre les données.

Rôle principal :

- Transforment la somme pondérée des entrées : $y = f\left(\sum_{i=1}^{n} a_i x_i + b\right)$
- Sans fonction d'activation, un réseau reste linéaire (peu importe sa profondeur)

Principales fonctions :

- Fonction seuil (Heaviside) - historique
- Fonction sigmoïde - classification binaire
- Fonction tanh - données centrées
- Fonction ReLU - réseaux profonds
- Fonction softmax - classification multi-classe

Fonction Seuil (Heaviside)

Définition :

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

Caractéristiques :

- Sortie binaire (0 ou 1)
- Utilisée dans le perceptron classique
- Simple mais non différentiable

Limitations :

- Pas de gradient $\Rightarrow$ pas d'apprentissage par rétropropagation
- Rarement utilisée aujourd'hui

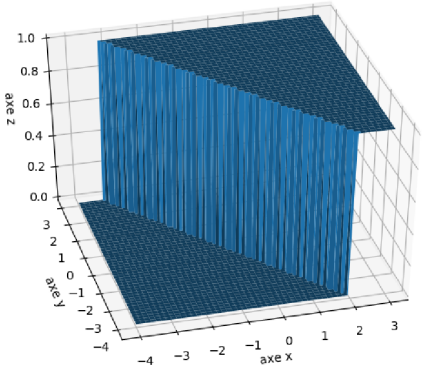


Figure: Fonction seuil

Fonction Sigmoïde

Définition :

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

Caractéristiques :

- Sortie continue entre 0 et 1
- Interprétable comme probabilité
- Dérivée : $f'(x) = f(x)(1 - f(x))$

Applications :

- Classification binaire (couche de sortie)
- Réseaux de neurones peu profonds

Limitations :

- Vanishing gradient (gradient $\to 0$ pour $|x|$ grand)
- Sortie non centrée en zéro

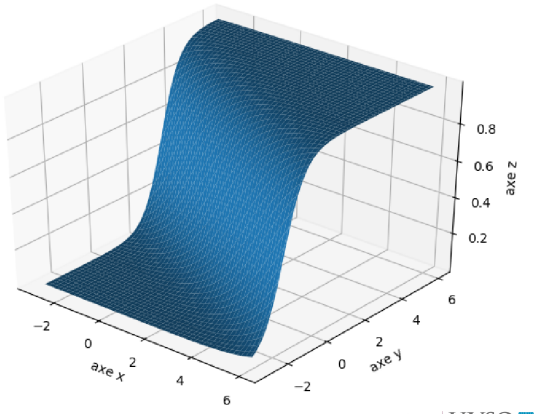


Figure: Fonction sigmoïde

Fonction Tangente Hyperbolique (tanh)

Définition :

$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Caractéristiques :

- Sortie continue entre -1 et 1
- Centre les données autour de zéro
- Dérivée : $f'(x) = 1 - f(x)^2$

Avantages vs sigmoïde :

- Sortie centrée $\Rightarrow$ convergence plus rapide
- Gradient plus fort

Limitations :

- Toujours le problème de vanishing

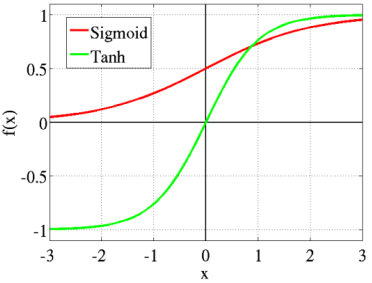


Figure: Fonction tanh

Fonction ReLU (Rectified Linear Unit)

Définition :

$$f(x) = \max(0, x) = \begin{cases} x & \text{si } x > 0 \\ 0 & \text{sinon} \end{cases}$$

Avantages :

- Calcul très rapide
- Résout le vanishing gradient (pour $x > 0$)
- Favorise la parcimonie (neurones inactifs)

Limitations :

- Dying ReLU : neurones "morts" si $x \leq 0$
- Sortie non bornée

Variantes :

- Leaky ReLU : $f(x) = \max(0.01x, x)$

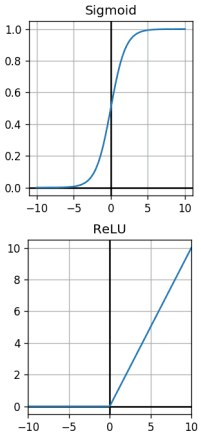


Figure: Fonction ReLU

Fonction Softmax

Définition :

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}}$$

Caractéristiques :

- Transforme un vecteur en distribution de probabilités
- $\sum_{i=1}^{K} f(x_i) = 1$
- $0 \leq f(x_i) \leq 1$ pour tout i

Application principale :

- Classification multi-classe (couche de sortie)
- Exemple : classer une image parmi 10 catégories

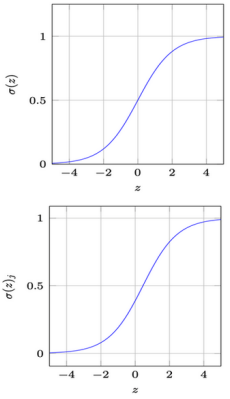


Figure: Fonction softmax

Comparaison et Choix de la Fonction d'Activation

Fonction	Intervalle	Gradient	Usage typique
Seuil	{0, 1}	Non	Historique
Sigmoïde	(0, 1)	Faible	Sortie binaire
tanh	(-1, 1)	Moyen	Couches cachées (ancien)
ReLU	$[0, +\infty)$	Fort	**Couches cachées**
Softmax	(0, 1)	Variable	**Sortie multi-classe**

Recommandations pratiques :

- **Couches cachées** : Utiliser ReLU (ou variantes) par défaut
- **Sortie - Classification binaire** : Sigmoïde
- **Sortie - Classification multi-classe** : Softmax
- **Sortie - Régression** : Pas de fonction d'activation (linéaire)

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 la sortie : $z = \sum_{i=1}^{n} a_i x_i + b$
   - Appliquer l'activation : $y_{\text{prédit}} = f(z)$
   - Calculer l'erreur : $e = y - y_{\text{prédit}}$
   - Mettre à jour les poids :
   $$a_i \leftarrow a_i + \eta \cdot e \cdot f'(z) \cdot x_i$$
3. Répéter jusqu'à convergence ou nombre d'itérations fixé.

Note : Le choix de la fonction d'activation f influence directement la vitesse et la qualité de l'apprentissage via sa dérivée f'.

Les Fonctions de Perte (Loss Functions)

Définition

Une fonction de perte mesure l'**écart** entre les prédictions du modèle $\hat{y}$ et les valeurs réelles y. Elle quantifie la qualité des prédictions.

Rôle principal :

- Guide l'apprentissage en indiquant la "direction" d'amélioration
- Permet d'optimiser les paramètres du modèle
- Essentielle pour les algorithmes d'apprentissage supervisé

Formule générale :

$$\mathcal{L}(y, \hat{y}) = \text{Mesure d'erreur entre } y \text{ (réel) et } \hat{y} \text{ (prédit)}$$

Objectif : Minimiser $\mathcal{L} \Rightarrow$ améliorer les performances du modèle

Erreur Quadratique Moyenne (MSE)

Définition :

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

Caractéristiques :

- Pénalise fortement les grandes erreurs (carré)
- Toujours positive : $\mathcal{L}_{MSE} \geq 0$
- Différentiable partout

Applications :

- Problèmes de régression
- Prédiction de valeurs continues
- Ex : prix, température, distance

Variante : RMSE = $\sqrt{MSE}$ (même unité que y)

Avantages :

- Simple à calculer
- Convexe $\Rightarrow$ optimisation facile
- Gradient clair

Limitations :

- Sensible aux valeurs aberrantes
- Peut donner trop d'importance aux grandes erreurs

Dérivée :

$$\frac{\partial \mathcal{L}_{MSE}}{\partial \hat{y}_i} = -\frac{2}{n}(y_i - \hat{y}_i)$$

Erreur Absolue Moyenne (MAE)

Définition :

$$\mathcal{L}_{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Caractéristiques :

- Pénalise linéairement les erreurs
- Moins sensible aux valeurs aberrantes que MSE
- Plus robuste

Applications :

- Régression avec outliers
- Données bruitées

Comparaison MSE vs MAE :

- MSE : grandes erreurs → forte pénalité
- MAE : toutes erreurs → pénalité proportionnelle

Exemple :

- Erreur = 10 :
  - MSE contribue : $10^2 = 100$
  - MAE contribue : $|10| = 10$

Limitation :

- Non différentiable en 0

Entropie Croisée Binaire (Binary Cross-Entropy)

Définition :

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

Contexte :

- $y_i \in \{0, 1\}$: classe réelle
- $\hat{y}_i \in (0, 1)$: probabilité prédite

Applications :

- Classification binaire
- Couplée avec fonction sigmoïde
- Ex : spam/non-spam, malade/sain

Interprétation :

- Si $y_i = 1 : \mathcal{L} = -\log(\hat{y}_i)$
  - $\hat{y}_i \to 1$: perte $\to 0$
  - $\hat{y}_i \to 0$: perte $\to \infty$
- Si $y_i = 0 : \mathcal{L} = -\log(1-\hat{y}_i)$

Avantages :

- Pénalise fortement les mauvaises prédictions confiantes
- Gradient bien défini
- Interprétation probabiliste

Entropie Croisée Catégorielle (Categorical Cross-Entropy)

Définition :

$$\mathcal{L}_{CCE} = -\frac{1}{n}\sum_{l=1}^{n}\sum_{c=1}^{C} y_{l,c}\log(\hat{y}_{l,c})$$

Contexte :

- C : nombre de classes
- $y_{i,c} \in \{0, 1\}$: encodage one-hot
- $\hat{y}_{i,c} \in (0, 1)$: probabilité prédite
- $\sum_{c=1}^{C} \hat{y}_{i,c} = 1$

Applications :

- Classification multi-classe
- Couplée avec fonction softmax
- Ex : reconnaissance de chiffres (0-9)

Exemple :

- 3 classes : chat, chien, oiseau
- Vraie classe : chat
- $y = [1, 0, 0]$ (one-hot)
- $\hat{y} = [0.7, 0.2, 0.1]$ (prédiction)
- $\mathcal{L} = -\log(0.7) \approx 0.36$

Note :

- Si vraie classe c^* :

$$\mathcal{L} = -\log(\hat{y}_{c^*})$$

- Encourage $\hat{y}_{c^*} \to 1$

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

Hinge Loss

Définition :

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

Contexte :

- $y_i \in \{-1, +1\}$: classe réelle
- $\hat{y}_i \in \mathbb{R}$: score prédit (non probabilité)

Applications :

- Machines à Vecteurs de Support (SVM)
- Classification avec marge maximale
- Cherche à séparer les classes avec une "marge"

Interprétation :

- Si $y_i \hat{y}_i \geq 1$: perte = 0
  - Bonne classification avec marge
- Si $y_i \hat{y}_i < 1$: perte = $1 - y_i \hat{y}_i$
  - Mauvaise classification ou marge insuffisante

Caractéristiques :

- Pénalise même les bonnes classifications proches de la frontière
- Force une "zone de sécurité"

Comparaison des Fonctions de Perte

Fonction	Type de problème	Fonction de sortie
MSE	Régression	Linéaire
MAE	Régression (robuste)	Linéaire
Binary Cross-Entropy	Classification binaire	Sigmoïde
Categorical Cross-Entropy	Classification multi-classe	Softmax
Hinge Loss	SVM / Classification	Score brut

Guide de sélection :

- **Régression** : MSE (standard) ou MAE (robuste aux outliers)
- **Classification binaire** : Binary Cross-Entropy + Sigmoïde
- **Classification multi-classe** : Categorical Cross-Entropy + Softmax
- **SVM** : Hinge Loss

Optimisation des Fonctions de Perte

Principe

L'entraînement consiste à **minimiser** la fonction de perte en ajustant les paramètres ∂ du modèle.

Algorithmes d'optimisation courants :

- **Descente de gradient** : Mise à jour des paramètres dans la direction opposée au gradient

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

où η est le taux d'apprentissage

- **Variantes avancées** :
  - SGD (Stochastic Gradient Descent) : calcul sur mini-batches
  - Adam : taux d'apprentissage adaptatif
  - RMSProp : normalisation du gradient

Impact de la fonction de perte :

- Forme de $\mathcal{L} \Rightarrow$ influence la convergence
- Gradient $\nabla_{\partial} \mathcal{L} \Rightarrow$ direction et vitesse de mise à jour

Visualisation et Suivi de l'Apprentissage

Courbes d'apprentissage :

- Évolution de $\mathcal{L}$ au cours des époques
- Training loss : perte sur données d'entraînement
- Validation loss : perte sur données de validation

Interprétation :

- Convergence : les deux courbes diminuent
- Surapprentissage : training ↓, validation ↑
- Sous-apprentissage : les deux restent élevées

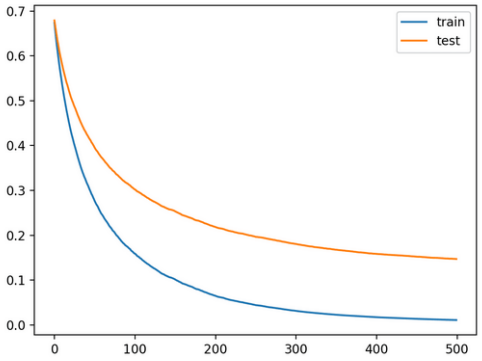


Figure: Évolution de la perte pendant l'entraînement

Importance : Surveiller ces courbes permet d'ajuster les hyperparamètres et détecter les problèmes d'apprentissage.

Introduction

Réseau de Neurones (MLP : MultiLayer Perceptron)

Objectifs du chapitre :

1. Comprendre l'architecture des réseaux de neurones multicouches
2. Étudier la propagation de l'information dans le réseau
3. Analyser des exemples concrets (portes logiques)
4. Comprendre la capacité d'approximation universelle

Rappel : Un neurone seul est limité (pas de XOR). Les réseaux multicouches permettent de résoudre des problèmes complexes non-linéaires.

Architecture d'un Réseau de Neurones

Définition

Un réseau de neurones est la **juxtaposition de plusieurs neurones**, organisés en **couches successives**.

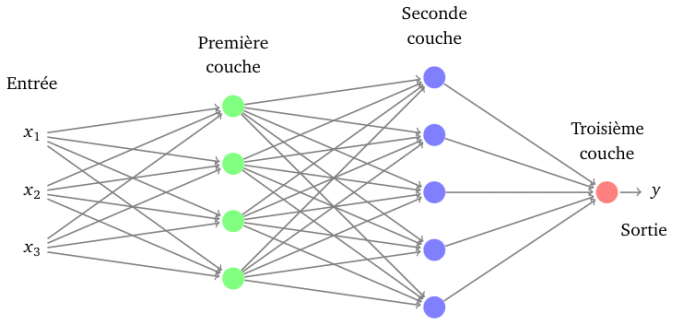
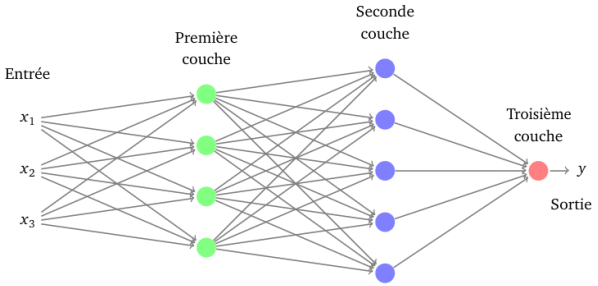


Figure: Architecture d'un réseau multicouche (Source : deepmath)

Propagation de l'Information (Forward Pass)

Principe :

1. Les données d'entrée sont propagées de gauche à droite
2. Chaque neurone calcule : $z = \sum w_i x_i + b$ puis $a = f(z)$
3. La sortie d'une couche devient l'entrée de la suivante



Notation :

- $a^{(l)}$: activations de la couche l
- $W^{(l)}$: matrice des poids entre couche $l-1$ et l

Réseaux à Sorties Multiples

Caractéristiques :

- Un réseau peut avoir plusieurs neurones de sortie
- Chaque sortie peut représenter une classe ou une valeur différente

Applications :

- Classification multi-classe
- Prédictions multiples simultanées
- Régression multi-variables

Exemple :

- Reconnaissance de chiffres : 10 sorties (0-9)
- Chaque sortie = probabilité d'une classe

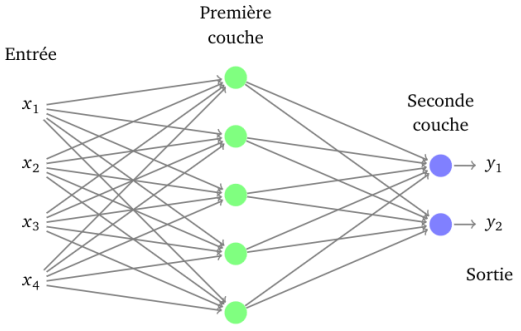


Figure: Réseau avec plusieurs sorties

Exemple 1 : Porte Logique ET (AND)

Objectif : Construire un réseau qui réalise la fonction ET

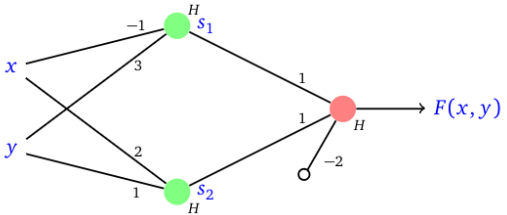


Figure: Architecture pour la porte ET

Rappel - Table de vérité du ET :

x_1	x_2	x_1 ET x_2
0	0	0
0	1	0
1	0	0

Exemple 1 : Configuration des Poids pour ET

Poids et biais choisis :

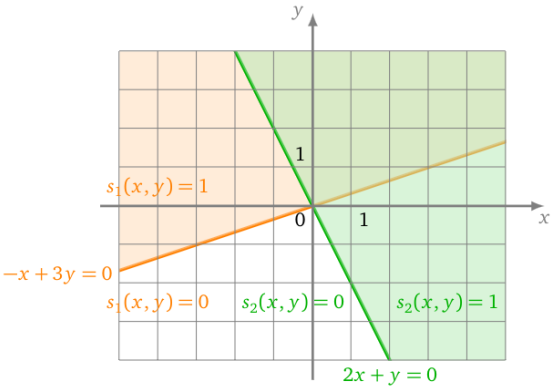


Figure: Poids : $w_1 = 1, w_2 = 1$, Biais : $b = -1.5$

LISV Laboratoire d'Ingénierie des systèmes de Versailles | UVSQ université PARIS-SACLAY

Exemple 1 : Vérification Complète pour ET

Formule : $y = H(x_1 + x_2 - 1.5)$

Vérification :

- $(0,0) : z = -1.5 < 0 \Rightarrow y = 0$
- $(0,1) : z = -0.5 < 0 \Rightarrow y = 0$
- $(1,0) : z = -0.5 < 0 \Rightarrow y = 0$
- $(1,1) : z = 0.5 \geq 0 \Rightarrow y = 1$

Le réseau implémente correctement la porte ET !

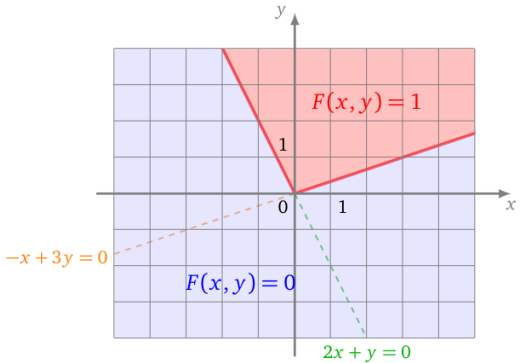


Figure: Neurone réalisant le ET

Exemple 2 : Porte Logique OU (OR)

Configuration modifiée : Changement du biais uniquement

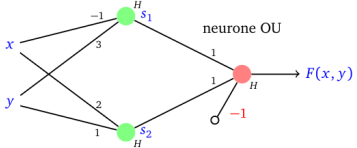


Figure: Poids : $w_1 = 1, w_2 = 1$, Biais : $b = -0.5$

Table de vérité du OU :

x_1	x_2	x_1 OU x_2
0	0	0
0	1	1
1	0	1
1	1	1

Exemple 2 : Vérification pour OU

Formule : $y = H(x_1 + x_2 - 0.5)$

Vérification :

- $(0,0) : z = -0.5 < 0 \Rightarrow y = 0$
- $(0,1) : z = 0.5 \geq 0 \Rightarrow y = 1$
- $(1,0) : z = 0.5 \geq 0 \Rightarrow y = 1$
- $(1,1) : z = 1.5 \geq 0 \Rightarrow y = 1$

Le réseau implémente correctement la porte OU !

Observation : Seul le biais change entre ET et OU !

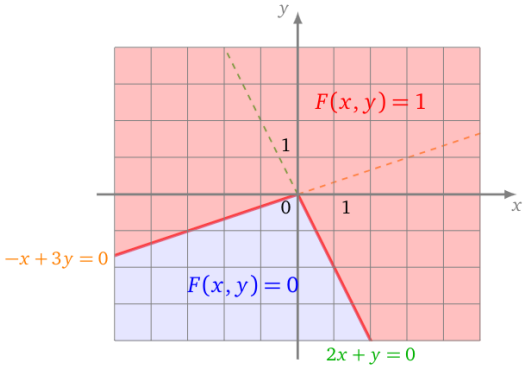


Figure: Neurone réalisant le OU

Exercice 1 : Analyse d'un Réseau Multicouche

Énoncé :

- Soit le réseau de neurones suivant
- La fonction d'activation est Heaviside (H) partout
- **Question :** Dessiner le graphe de la sortie en fonction de x

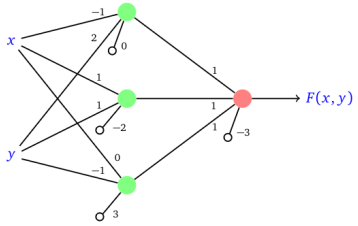


Figure: Réseau à analyser

Indice : Analyser d'abord la sortie de chaque neurone de la couche cachée, puis combiner les résultats.

Exercice 1 : Démarche de Résolution

Étapes :

1. **Neurone cachée 1 :**
   - $z_1 = 2x - 1$
   - $a_1 = H(z_1) = \begin{cases} 1 & \text{si } x \geq 0.5 \\ 0 & \text{sinon} \end{cases}$
2. **Neurone cachée 2 :**
   - $z_2 = -2x + 2$
   - $a_2 = H(z_2) = \begin{cases} 1 & \text{si } x \leq 1 \\ 0 & \text{sinon} \end{cases}$
3. **Neurone de sortie :**
   - $z_{\text{out}} = a_1 + a_2 - 1.5$
   - $y = H(z_{\text{out}})$

Question : Pour quelles valeurs de x la sortie vaut-elle 1 ?

Exercice 1 : Solution

Analyse par intervalles :

- Si $x < 0.5 : a_1 = 0, a_2 = 1$
  - $z = 0 + 1 - 1.5 = -0.5$
  - $y = 0$
- Si $0.5 \leq x \leq 1 : a_1 = 1, a_2 = 1$
  - $z = 1 + 1 - 1.5 = 0.5$
  - $y = 1$
- Si $x > 1 : a_1 = 1, a_2 = 0$
  - $z = 1 + 0 - 1.5 = -0.5$
  - $y = 0$

Résultat : Le réseau détecte si $x \in [0.5, 1]$

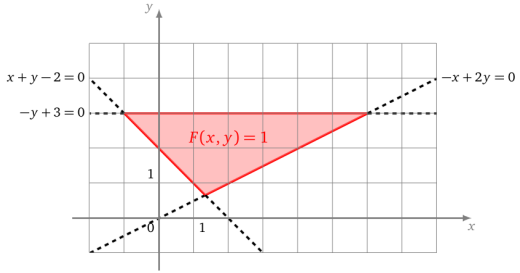


Figure: Graphe de la sortie : fonction porte

Interprétation :

- Le réseau crée une "fenêtre"
- Activation uniquement dans $[0.5, 1]$

Capacité d'Approximation des Réseaux Multicouches

Théorème d'approximation universelle

Un réseau de neurones avec **une seule couche cachée** contenant suffisamment de neurones peut approximer **n'importe quelle fonction continue** sur un intervalle compact avec une précision arbitraire.

Implications :

- Les réseaux multicouches sont extrêmement puissants
- Peuvent apprendre des relations complexes et non-linéaires
- Plus de neurones = plus de flexibilité (attention au surapprentissage !)

Exemple illustré :

- L'exercice montre qu'avec 2 neurones cachés, on peut créer une fonction "porte"
- Avec plus de neurones, on peut approximer des fonctions encore plus complexes
- En combinant plusieurs "portes", on peut reconstruire toute fonction continue !

Résumé : Réseaux de Neurones Multicouches

Points clés :

1. **Architecture** : Couches successives de neurones (entrée → cachées → sortie)
2. **Propagation** : Les informations se propagent de gauche à droite
3. **Non-linéarité** : Les fonctions d'activation permettent d'apprendre des relations complexes
4. **Expressivité** : Un réseau peut implémenter des fonctions logiques et approximer toute fonction continue
5. **Apprentissage** : Les poids et biais sont ajustés pour minimiser l'erreur (vu dans les chapitres suivants : rétropropagation)

Prochaine étape : Comment *entraîner* ces réseaux ? ⇒ Algorithme de rétropropagation du gradient (backpropagation)

Introduction

Mise à Jour des Poids dans un Réseau de Neurones

Problématique :

- Comment gérer la taille des données VS la capacité mémoire ?
- Quelle fréquence de mise à jour des poids choisir ?

Objectifs du chapitre :

1. Comprendre les différentes stratégies de descente de gradient
2. Comparer SGD, mini-batch et batch gradient descent
3. Choisir la méthode adaptée au contexte

Vocabulaire Essentiel

Époque (Epoch)

Un **passage complet** sur l'ensemble des données d'entraînement.

- Exemple : 1 000 échantillons $\Rightarrow$ 1 époque = tous les 1 000 échantillons utilisés une fois

Batch

Un **sous-ensemble** des données utilisé pour une mise à jour.

- Taille du batch (`batch_size`) : nombre d'échantillons par batch

Itération

Une **mise à jour des poids** après traitement d'un batch.

$$\text{Itérations par époque} = \frac{\text{Nombre total d'échantillons}}{\text{Batch size}}$$

Exemple Concret de Calcul

Configuration :

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

Calculs :

- Itérations par époque : $\frac{1000}{100} = 10$ itérations
- Total d'itérations : $10 \times 5 = 50$ mises à jour

Déroulement :

1. Le modèle traite 100 échantillons (1 batch)
2. Calcul de l'erreur sur ces 100 échantillons
3. Mise à jour des poids (1 itération)
4. Répéter 10 fois $\Rightarrow$ 1 époque complète
5. Répéter ce processus 5 fois $\Rightarrow$ 5 époques

Les Trois Approches

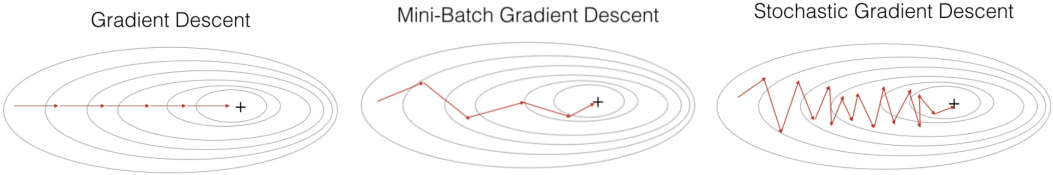


Figure: Visualisation des trajectoires selon la méthode

Différences clés :

- Taille du batch utilisé pour calculer le gradient
- Fréquence de mise à jour des poids
- Compromis entre vitesse, stabilité et mémoire

Descente de Gradient Stochastique (SGD)

Principe

Mise à jour des poids **après chaque échantillon individuel**.

$$\text{Batch size} = 1$$

Processus :

1. Prendre un échantillon (x_i, y_i)
2. Calculer la prédiction : $\hat{y}_i = f(x_i; \partial)$
3. Calculer l'erreur : $\mathcal{L}(y_i, \hat{y}_i)$
4. Mettre à jour les poids : $\partial \leftarrow \partial - \eta \nabla_\partial \mathcal{L}$
5. Répéter pour l'échantillon suivant

Caractéristiques :

- Nombre d'itérations par époque = Nombre total d'échantillons
- Exemple : 1 000 échantillons $\Rightarrow$ 1 000 mises à jour par époque

SGD : Avantages et Inconvénients

Avantages :

- Convergence rapide au début de l'entraînement
- Exploration efficace de l'espace des paramètres
- Peut échapper aux minima locaux (grâce au bruit)
- Faible consommation mémoire
- Mise à jour fréquente

Inconvénients :

- Très bruyant : trajectoire erratique
- Convergence instable
- Difficile de converger précisément
- Ne converge jamais vraiment (oscille autour du minimum)
- Lent en pratique (pas de parallélisation GPU)

Usage typique :

- Rarement utilisé seul en pratique moderne
- Utile pour l'apprentissage en ligne (streaming data)

Exemple Code : SGD

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Donnees fictives
X = np.random.rand(100, 10)  # 100 echantillons, 10 caracteristiques
y = np.random.rand(100, 1)   # 100 etiquettes
# Modèle simple
model = Sequential([
    Dense(32, activation='relu', input_shape=(10,)),
    Dense(1, activation='linear')])
# Compilation avec SGD
model.compile(
    optimizer=tf.keras.optimizers.SGD(learning_rate=0.01),
    loss='mse')
# Entraînement avec batch_size=1 (SGD pur)
# 100 echantillons => 100 iterations par epoque
model.fit(X, y, batch_size=1, epochs=10, verbose=1)
```

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

Principe

Mise à jour des poids **après chaque mini-lot** de données.

$$1 < \text{Batch size} < \text{Nombre total d'échantillons}$$

Processus :

1. Diviser les données en mini-lots de taille b
2. Prendre un mini-lot $\{(x_i, y_i)\}_{i=1}^{b}$
3. Calculer la perte moyenne sur le mini-lot : $\mathcal{L} = \frac{1}{b} \sum_{i=1}^{b} \mathcal{L}_i$
4. Calculer le gradient moyen : $\nabla_\partial \mathcal{L}$
5. Mettre à jour les poids : $\partial \leftarrow \partial - \eta \nabla_\partial \mathcal{L}$
6. Passer au mini-lot suivant

Tailles courantes :

- Petits modèles : 16, 32, 64
- Grands modèles : 128, 256, 512

Mini-Batch : Le Meilleur Compromis

Avantages :

- Équilibre optimal entre stabilité et vitesse
- Parallélisation GPU efficace
- Gradient moins bruité que SGD
- Plus rapide que Batch GD
- Utilisation mémoire raisonnable
- Méthode standard en pratique

Points d'attention :

- Nécessite de choisir la batch size
- Compromis à trouver selon :
  - Taille des données
  - Mémoire GPU disponible
  - Stabilité souhaitée

Règle empirique :

- Commencer avec 32
- Augmenter si possible (64, 128, 256)
- Batch size = puissance de 2 (optimisation GPU)

Exemple Code : Mini-Batch

```
# Même modèle que precedemment
model = Sequential([
    Dense(32, activation='relu', input_shape=(10,)),
    Dense(1, activation='linear')
])

# Compilation avec Adam (optimiseur moderne recommande)
model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=0.001),
    loss='mse'
)

# Entraînement avec batch_size=16 (Mini-batch)
# 100 echantillons / 16 = 6.25 => 7 iterations par epoque
model.fit(X, y, batch_size=16, epochs=10, verbose=1)

# Autres tailles courantes à essayer :
# batch_size=32  (3 iterations/epoque)
# batch_size=64  (2 iterations/epoque)
```

Note : Adam est généralement préféré à SGD car il adapte automatiquement le taux d'apprentissage pour chaque paramètre.

Descente par Batch (Batch Gradient Descent)

Principe

Mise à jour des poids **une seule fois par époque**, après avoir traité **toutes** les données.

Batch size = Nombre total d'échantillons

Processus :

1. Prendre l'ensemble complet des données : $\{(x_i, y_i)\}_{i=1}^{N}$
2. Calculer la perte totale : $\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}_i$
3. Calculer le gradient exact : $\nabla_\partial \mathcal{L}$
4. Mettre à jour les poids : $\partial \leftarrow \partial - \eta \nabla_\partial \mathcal{L}$
5. Une époque = une itération

Caractéristiques :

- Gradient calculé sur l'ensemble complet ⇒ plus précis
- 1 seule mise à jour par époque

Batch GD : Stabilité au Prix de l'Efficacité

Avantages :

- Gradient exact (pas de bruit)
- Trajectoire stable vers le minimum
- Convergence précise
- Comportement déterministe
- Théoriquement optimal

Inconvénients :

- Très coûteux en mémoire
- Lent : une mise à jour par époque
- Impossible pour grands datasets
- Risque de rester coincé dans un minimum local
- Pas adapté aux données massives

Usage typique :

- Petits datasets (< 10 000 échantillons)
- Problèmes convexes où la précision est cruciale
- Rarement utilisé en deep learning moderne

Exemple Code : Batch Gradient Descent

```
# Même modèle
model = Sequential([
    Dense(32, activation='relu', input_shape=(10,)),
    Dense(1, activation='linear')
])

# Compilation avec RMSprop (autre optimiseur moderne)
model.compile(
    optimizer=tf.keras.optimizers.RMSprop(learning_rate=0.001),
    loss='mse'
)

# Entraînement avec batch_size = nombre total d'echantillons
# 100 echantillons => 1 iteration par epoque (Batch GD)
model.fit(X, y, batch_size=100, epochs=10, verbose=1)

# Attention : Pour un dataset de 1 million d'echantillons,
# batch_size=1000000 pourrait saturer la memoire GPU !
```

Attention : En pratique, cette approche est déconseillée pour les grands datasets.

Tableau Comparatif

Méthode	Batch Size	Avantages	Inconvénients
SGD	1	• Rapide initialement • Échappe minima locaux • Peu de mémoire	• Très bruyant • Convergence instable • Pas de parallélisation
Mini-Batch	16-512	• Équilibre optimal • Parallélisation GPU • Standard industrie	• Nécessite choix batch size
Batch	N (tous)	• Gradient exact • Convergence stable	• Coûteux mémoire • Très lent • Impossible grands datasets

Itérations par époque :

- SGD : N itérations (si N échantillons)
- Mini-Batch : $\lceil N/b \rceil$ itérations (si batch size $= b$)
- Batch : 1 itération

Guide de Sélection de la Méthode

Facteurs de décision :

1. **Taille des données**
   - Petit dataset (< 10k) : Batch GD possible
   - Dataset moyen (10k-1M) : Mini-Batch (32-128)
   - Grand dataset (> 1M) : Mini-Batch (128-512)
2. **Contraintes matérielles**
   - GPU avec peu de mémoire : batch size plus petit (16-32)
   - GPU puissant : batch size plus grand (256-512)
3. **Objectifs**
   - Vitesse prioritaire : Mini-Batch avec batch size élevé
   - Stabilité prioritaire : Batch size plus grand ou Batch GD
   - Généralisation : Mini-Batch (le bruit aide)

Recommandation par défaut : Mini-Batch avec `batch_size = 32`

Conseils Pratiques

Choix du batch size :

- **Commencer par 32**, puis expérimenter
- Utiliser des puissances de 2 : 16, 32, 64, 128, 256, 512
- Plus grand batch $\Rightarrow$ convergence plus stable mais moins de généralisation
- Plus petit batch $\Rightarrow$ plus de bruit, meilleure exploration

Optimiseurs modernes :

- **Adam** : excellent choix par défaut (adapte le learning rate)
- **SGD with momentum** : bon pour la convergence
- **RMSprop** : efficace pour RNN

En pratique :

- Mini-Batch est utilisé dans > 95% des cas
- Frameworks (TensorFlow, PyTorch) optimisés pour Mini-Batch
- Le terme "SGD" dans la littérature désigne souvent le Mini-Batch !

Résumé

Points clés à retenir

1. **SGD** (batch size = 1) : Théorique mais peu utilisé
2. **Mini-Batch** (batch size = 16-512) :
   - Méthode standard en pratique
   - Meilleur compromis vitesse/stabilité/mémoire
3. **Batch GD** (batch size = N) : Limité aux petits datasets
4. Le choix dépend de : taille données, mémoire GPU, objectifs
5. Par défaut : Mini-Batch avec Adam et batch size = 32

Prochaine étape : Rétropropagation du gradient (backpropagation) pour calculer les gradients efficacement

Keras : Framework de Deep Learning

Objectifs du chapitre :

1. Découvrir Keras et son intégration dans TensorFlow
2. Comprendre les couches Dense (fully connected)
3. Construire un premier modèle de réseau de neurones
4. Entraîner et évaluer un modèle

Qu'est-ce que Keras ?

Définition

Keras est une **API de haut niveau** pour construire et entraîner des modèles de réseaux de neurones, intégrée dans TensorFlow.

Caractéristiques principales :

- Interface simple et intuitive : code lisible et concis
- Modulaire : construction par assemblage de couches
- Extensible : ajout facile de nouvelles fonctionnalités
- Abstraction de la complexité de TensorFlow
- Compatible avec CPU et GPU
- Large communauté et documentation riche

Philosophie :

- Développement rapide de prototypes
- Code minimal pour des modèles performants
- Focus sur l'architecture, pas sur l'implémentation

Installation et Vérification

Installation de TensorFlow (inclut Keras) :

```
pip install tensorflow
```

Vérification de l'installation :

```
# Dans un terminal Python ou Jupyter
import tensorflow as tf
print(f"TensorFlow version: {tf.__version__}")

# Verifier si GPU est disponible
print(f"GPU disponible: {tf.config.list_physical_devices('GPU')}"
```

Versions recommandées :

- TensorFlow ≥ 2.x (Keras intégré par défaut)
- Python ≥ 3.8

Imports Essentiels

Bibliothèques de base :

```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
# Imports pour construire des modèles
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout, BatchNormaliz
# Imports pour l'optimisation et les metriques
from tensorflow.keras.optimizers import Adam, SGD
from tensorflow.keras.losses import BinaryCrossentropy, Categoric
from tensorflow.keras.metrics import Accuracy, Precision, Recall
```

Note : Le modèle `Sequential` permet de construire des réseaux en empilant des couches séquentiellement (architecture linéaire).

Qu'est-ce qu'une Couche Dense ?

Définition

Une couche **Dense** (ou fully connected) est une couche où **chaque neurone est connecté à tous les neurones de la couche précédente**.

Formule mathématique :

$$\mathbf{y} = f(W \cdot \mathbf{x} + \mathbf{b})$$

Où :

- $\mathbf{x}$: vecteur d'entrée (taille n)
- W : matrice des poids (taille $m \times n$)
- $\mathbf{b}$: vecteur de biais (taille m)
- f : fonction d'activation
- $\mathbf{y}$: vecteur de sortie (taille m)

Paramètres à apprendre :

- Nombre de poids : $n \times m$
- Nombre de biais : m
- **Total** : $n \times m + m$

Exemple :

- Entrée : 100 neurones
- Sortie : 64 neurones
- Paramètres : $100 \times 64 + 64 = 6\,464$

Utilisation d'une Couche Dense

Applications principales :

- **Classification** : multi-classe, binaire
- **Régression** : prédiction de valeurs continues
- **Feature extraction** : apprentissage de représentations
- Couches cachées dans les réseaux profonds

Positionnement typique :

- Couches cachées : extraction de caractéristiques
  - Activation : ReLU, tanh
  - Plusieurs couches empilées
- Couche de sortie : décision finale
  - Classification binaire : 1 neurone, sigmoid
  - Classification multi-classe : C neurones, softmax
  - Régression : k neurones, linéaire

Définir une Couche Dense dans Keras

Syntaxe de base :

```
from tensorflow.keras.layers import Dense

# Creer une couche Dense
layer = Dense(units=64, activation='relu', use_bias=True)
```

Paramètres principaux :

- `units` : **Nombre de neurones** dans la couche
- `activation` : **Fonction d'activation**
  - `'relu'` : couches cachées (standard)
  - `'sigmoid'` : classification binaire (sortie)
  - `'softmax'` : classification multi-classe (sortie)
  - `'linear'` ou `None` : régression (sortie)
- `input_shape` : **Forme des données d'entrée** (première couche uniquement)
- `use_bias` : utiliser un biais (défaut : `True`)
- `kernel_initializer` : initialisation des poids (défaut : 'glorot_uniform')

Construction d'un Modèle Sequential

Approche 1 : Liste de couches

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

model = Sequential([
    Dense(units=64, activation='relu', input_shape=(100,)),
    Dense(units=32, activation='relu'),
    Dense(units=10, activation='softmax')  # Sortie : 10 classes
])
```

Approche 2 : Ajout progressif

```
model = Sequential()
model.add(Dense(64, activation='relu', input_shape=(100,)))
model.add(Dense(32, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Note : `input_shape` est requis uniquement pour la première couche.

Visualiser l'Architecture du Modèle

Afficher le résumé du modèle :

```
model.summary()
```

Sortie exemple :

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 64)                6464
dense_1 (Dense)              (None, 32)                2080
dense_2 (Dense)              (None, 10)                330
=================================================================
Total params: 8,874
Trainable params: 8,874
Non-trainable params: 0
_________________________________________________________________
```

Calcul des paramètres : Première couche : $100 \times 64 + 64 = 6{,}464$

Compilation du Modèle

Étape de compilation

Avant l'entraînement, il faut **compiler** le modèle : spécifier l'optimiseur, la fonction de perte et les métriques.

```
model.compile(
    optimizer='adam',                    # Algorithme d'optimisation
    loss='categorical_crossentropy',     # Fonction de perte
    metrics=['accuracy']                 # Metriques à suivre)
```

Paramètres :

- **optimizer** : algorithme de mise à jour des poids
  - `'adam'` : adaptatif, recommandé par défaut
  - `'sgd'` : descente de gradient classique
  - `'rmsprop'` : bon pour RNN
- **loss** : fonction de perte à minimiser
- **metrics** : métriques pour évaluer les performances

Choix de la Fonction de Perte

Type de problème	Fonction de perte	Activation sortie
Classification binaire	`binary_crossentropy`	`sigmoid`
Classification multi-classe (labels one-hot)	`categorical_crossentropy`	`softmax`
Classification multi-classe (labels entiers)	`sparse_categorical_crossentropy`	`softmax`
Régression	`mse` ou `mae`	`linear` / None

Exemple :

```
# Classification binaire
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# Regression
model.compile(optimizer='adam',
```

Entraînement du Modèle

Méthode `fit()` :

```
history = model.fit(
    X_train, y_train,              # Donnees d'entraînement
    batch_size=32,                 # Taille du mini-batch
    epochs=50,                     # Nombre d'epoques
    validation_split=0.2,          # 20% pour validation
    verbose=1                      # Affichage progression
)
```

Paramètres importants :

- `batch_size` : nombre d'échantillons par mise à jour (défaut : 32)
- `epochs` : nombre de passages complets sur les données
- `validation_split` : fraction des données pour validation
- `validation_data` : données de validation explicites `(X_val, y_val)`
- `verbose` : niveau de détail (0=muet, 1=barre, 2=ligne par époque)

Retour : L'objet `history` contient l'historique de l'entraînement.

Exemple Pratique : Classification Binaire

```python
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# 1. Generation de donnees fictives
X = np.random.random((1000, 20))       # 1000 echantillons, 20 features
y = np.random.randint(2, size=(1000, 1))  # Labels binaires (0 ou 1)

# 2. Separation train/test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

# 3. Construction du modèle
model = Sequential([
    Dense(32, activation='relu', input_shape=(20,)),  # Couche cachee 1
    Dense(16, activation='relu'),                     # Couche cachee 2
    Dense(1, activation='sigmoid')                    # Sortie binaire
])
```

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

Exemple Pratique : Entraînement et Évaluation

```python
# 4. Compilation
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)

# 5. Entraînement
history = model.fit(
    X_train, y_train,
    batch_size=32,
    epochs=20,
    validation_split=0.2,
    verbose=1
)
# 6. Evaluation sur les donnees de test
test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test accuracy: {test_accuracy:.4f}")
# 7. Predictions
predictions = model.predict(X_test[:5])  # Predire pour 5 echantillons
print(f"Predictions: {predictions.flatten()}")
```

Visualisation de l'Entraînement

```python
import matplotlib.pyplot as plt

# Tracer l'evolution de la perte
plt.figure(figsize=(12, 4))

plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Evolution de la Perte')
plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.title('Evolution de la Precision')
plt.tight_layout()
plt.show()
```

Exemple : Classification Multi-classe

```python
import numpy as np
from tensorflow.keras.utils import to_categorical
# Donnees pour 3 classes
X = np.random.random((1000, 50))
y = np.random.randint(3, size=(1000,))  # Labels : 0, 1 ou 2
# Conversion en one-hot encoding
y_one_hot = to_categorical(y, num_classes=3)
# y=1 devient [0, 1, 0]
# Modèle
model = Sequential([
    Dense(64, activation='relu', input_shape=(50,)),
    Dense(32, activation='relu'),
    Dense(3, activation='softmax')  # 3 classes])
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',  # Pour one-hot
    metrics=['accuracy']
)
model.fit(X, y_one_hot, epochs=10, batch_size=32)
```

Exemple : Régression

```python
import numpy as np

# Donnees de regression
X = np.random.random((1000, 10))
y = np.random.random((1000, 1))  # Valeurs continues

# Modèle de regression
model = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),
    Dense(32, activation='relu'),
    Dense(1)  # Sortie : 1 valeur, pas d'activation (lineaire)])
model.compile(
    optimizer='adam',
    loss='mse',           # Mean Squared Error
    metrics=['mae']       # Mean Absolute Error)
history = model.fit(X, y, epochs=20, batch_size=32, validation_split=0.2)
# Prediction
predictions = model.predict(X[:5])
print(f"Predictions : {predictions.flatten()}")
print(f"Valeurs reelles : {y[:5].flatten()}")
```

Bonnes Pratiques avec Keras

1. **Architecture du modèle :**
   - Commencer simple : 1-2 couches cachées
   - Augmenter progressivement la complexité si nécessaire
   - Nombre de neurones : décroissant vers la sortie (ex : $128 \rightarrow 64 \rightarrow 32$)
2. **Choix des hyperparamètres :**
   - **Activation** : ReLU pour couches cachées (standard)
   - **Batch size** : 32 par défaut, ajuster selon mémoire
   - **Epochs** : 20-50 pour commencer, surveiller validation
   - **Optimizer** : Adam recommandé, learning_rate=0.001 (défaut)
3. **Prévention du surapprentissage :**
   - Utiliser `validation_split` ou `validation_data`
   - Ajouter des couches `Dropout` si besoin
   - Early stopping pour arrêter automatiquement

Techniques Avancées

1. Dropout pour régularisation :

```
from tensorflow.keras.layers import Dropout
model = Sequential([
    Dense(64, activation='relu', input_shape=(100,)),
    Dropout(0.5),  # Desactive aleatoirement 50% des neurones
    Dense(32, activation='relu'),
    Dropout(0.3),
    Dense(10, activation='softmax')])
```

2. Early Stopping :

```
from tensorflow.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(
    monitor='val_loss',  # Surveiller la validation loss
    patience=5,          # Arrêt si pas d'amelioration après 5 epoques
    restore_best_weights=True)
model.fit(X, y, epochs=100, validation_split=0.2,
          callbacks=[early_stop])
```

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

Résumé

Points clés à retenir :

1. **Keras** : API simple et puissante pour le deep learning
2. **Couches Dense** : couches fully connected, base des réseaux
   - Paramètres : $n \times m + m$ (poids + biais)
3. **Workflow** : Construction $\rightarrow$ Compilation $\rightarrow$ Entraînement $\rightarrow$ Évaluation
4. **Compilation** : définir optimizer, loss, metrics
5. **Choix fonction de perte** selon le type de problème :
   - Binary crossentropy (classification binaire)
   - Categorical crossentropy (multi-classe)
   - MSE/MAE (régression)
6. **Validation** : toujours surveiller val_loss pour éviter le surapprentissage

Prochaines Étapes

Concepts à explorer :

- **Rétropropagation** : comment le gradient est calculé
- **Régularisation** : Dropout, L1/L2, Batch Normalization
- **Autres types de couches** : Conv2D (CNN), LSTM (RNN)
- **Transfer learning** : utiliser des modèles pré-entraînés
- **Hyperparameter tuning** : optimiser les performances

Ressources :

- Documentation officielle : `https://keras.io`
- TensorFlow tutorials : `https://www.tensorflow.org/tutorials`
- Cours en ligne : Coursera, Deep Learning Specialization