Cours Machine Learning

Chapitre 06

Réseaux de Neurones Fondamentaux

Objectifs d'apprentissage :

— Comprendre le fonctionnement du perceptron et du perceptron multi-couches (MLP)
— Maîtriser le forward pass et le backpropagation
— Découvrir les fonctions d'activation et leur rôle
— Implémenter un réseau de neurones from scratch
— Appliquer les techniques de régularisation (dropout, batch norm)

Prérequis : Chapitres 01 (Math), 02 (Métriques), 03 (Régression)

Durée estimée : 6-8 heures

Notebooks : `06_demo_*.ipynb`

Cours ML - Sandbox-ML
Version 1.0 - 2026

Table des matières

1 Introduction aux Réseaux de Neurones

1.1 Motivation

Les réseaux de neurones artificiels sont inspirés du fonctionnement du cerveau humain. Ils permettent de modéliser des relations non-linéaires complexes entre les données d'entrée et les sorties.

Exemple

Classification d'images manuscrites Reconnaître des chiffres manuscrits (MNIST) : chaque pixel est une entrée, et le réseau doit prédire le chiffre (0-9). Une régression logistique simple obtient 92% de précision, tandis qu'un réseau de neurones atteint 98%.

1.2 Historique

- **1943** : McCulloch-Pitts - premier modèle de neurone formel
- **1958** : Rosenblatt - Perceptron (classification binaire linéaire)
- **1969** : Minsky & Papert - limites du perceptron (XOR)
- **1986** : Rumelhart, Hinton, Williams - Backpropagation
- **2012** : AlexNet - révolution deep learning (ImageNet)

1.3 Analogie biologique

Un neurone artificiel est une simplification extrême d'un neurone biologique :

- **Dendrites** $\rightarrow$ Entrées pondérées ($\mathbf{x} \odot \mathbf{w}$)
- **Corps cellulaire** $\rightarrow$ Sommation ($\sum w_i x_i + b$)
- **Axone** $\rightarrow$ Fonction d'activation ($\sigma(z)$)
- **Synapses** $\rightarrow$ Poids ajustables ($\mathbf{w}$)

2 Le Perceptron

2.1 Définition

Définition

Perceptron Le perceptron est un modèle de classification binaire linéaire. Pour une entrée $\mathbf{x} \in \mathbb{R}^d$, il calcule :

$$y = \begin{cases} 1 & \text{si } \mathbf{w}^T\mathbf{x} + b > 0 \\ 0 & \text{sinon} \end{cases} \tag{1}$$

où $\mathbf{w} \in \mathbb{R}^d$ sont les poids et $b \in \mathbb{R}$ est le biais.

Le perceptron définit un **hyperplan de séparation** dans l'espace des features :

$$\mathbf{w}^T\mathbf{x} + b = 0 \tag{2}$$

2.2 Algorithme d'apprentissage

Algorithm 1 Perceptron Learning Algorithm

Require: Données $\{(\mathbf{x}_i, y_i)\}_{i=1}^{n}$, $y_i \in \{0, 1\}$
Require: Learning rate α, nombre d'epochs T
Ensure: Poids $\mathbf{w}$, biais b

Initialiser $\mathbf{w} = \mathbf{0}$, $b = 0$
for *epoch* $= 1$ **to** T **do**
  for $i = 1$ **to** n **do**
    Prédire : $\hat{y}_i = \mathbb{1}[\mathbf{w}^T \mathbf{x}_i + b > 0]$
    if $\hat{y}_i \neq y_i$ **then**
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_i - \hat{y}_i)\mathbf{x}_i$
      $b \leftarrow b + \alpha(y_i - \hat{y}_i)$
    end if
  end for
end for
return $\mathbf{w}, b$

2.3 Théorème de convergence

Théorème: Convergence du Perceptron

Si les données sont linéairement séparables, l'algorithme du perceptron converge en un nombre fini d'itérations.

/!\ Attention

Le perceptron **ne peut pas** résoudre le problème XOR (non linéairement séparable). C'est une limitation majeure qui a motivé le développement des réseaux multi-couches.

2.4 Implémentation

```python
import numpy as np

class Perceptron:
    def __init__(self, learning_rate=0.01, n_epochs=100):
        self.lr = learning_rate
        self.n_epochs = n_epochs
        self.w = None
        self.b = 0

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.w = np.zeros(n_features)

        for epoch in range(self.n_epochs):
            for i in range(n_samples):
                # Forward pass
```

```python
                z = np.dot(X[i], self.w) + self.b
                y_pred = 1 if z > 0 else 0

                # Update weights si erreur
                if y_pred != y[i]:
                    update = self.lr * (y[i] - y_pred)
                    self.w += update * X[i]
                    self.b += update

        return self

    def predict(self, X):
        z = np.dot(X, self.w) + self.b
        return (z > 0).astype(int)
```

Listing 1 – Perceptron from scratch

3 Fonctions d'Activation

Les fonctions d'activation introduisent la **non-linéarité** dans les réseaux de neurones. Sans elles, un réseau multi-couches serait équivalent à une régression linéaire.

3.1 Sigmoid

Définition

Fonction Sigmoid

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{3}$$

Propriétés :

- Sortie entre 0 et 1 (interprétable comme probabilité)
- Dérivée : $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
- Problème : **vanishing gradient** pour $|z|$ grand

3.2 Tanh (Tangente Hyperbolique)

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1 \tag{4}$$

Propriétés :

- Sortie entre -1 et 1 (centré à 0, meilleur que sigmoid)
- Dérivée : $\tanh'(z) = 1 - \tanh^2(z)$
- Aussi sujet au vanishing gradient

3.3 ReLU (Rectified Linear Unit)

Définition

ReLU

$$\text{ReLU}(z) = \max(0, z) = \begin{cases} z & \text{si } z > 0 \\ 0 & \text{sinon} \end{cases} \tag{5}$$

Propriétés :

- Calcul très rapide
- Pas de vanishing gradient pour $z > 0$
- Dérivée : $\text{ReLU}'(z) = \mathbb{1}[z > 0]$
- Problème : **dying ReLU** (neurones inactifs si $z < 0$)

ReLU est la fonction d'activation par défaut dans les réseaux modernes.

3.4 Variantes de ReLU

Leaky ReLU :

$$\text{LeakyReLU}(z) = \begin{cases} z & \text{si } z > 0 \\ \alpha z & \text{sinon} \end{cases} \quad (\alpha \approx 0.01) \tag{6}$$

ELU (Exponential Linear Unit) :

$$\text{ELU}(z) = \begin{cases} z & \text{si } z > 0 \\ \alpha(e^z - 1) & \text{sinon} \end{cases} \tag{7}$$

Swish / SiLU :

$$\text{Swish}(z) = z \cdot \sigma(z) \tag{8}$$

3.5 Softmax (pour classification multi-classe)

Définition

Softmax Pour un vecteur $\mathbf{z} \in \mathbb{R}^K$, la fonction softmax renvoie un vecteur de probabilités :

$$\text{softmax}(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{pour } j = 1, \ldots, K \tag{9}$$

Propriété : $\sum_{j=1}^{K} \text{softmax}(\mathbf{z})_j = 1$ (distribution de probabilité)

TABLE 1 – Comparaison des fonctions d'activation

Fonction	**Plage**	**Vanishing Gradient**	**Usage**
Sigmoid	$(0, 1)$	× Oui	Output binaire
Tanh	$(-1, 1)$	× Oui	RNN (historique)
ReLU	$[0, \infty)$	✓ Non	Hidden layers (défaut)
Leaky ReLU	$(-\infty, \infty)$	✓ Non	Alternative ReLU
Softmax	$[0, 1]^K$	-	Output multi-classe

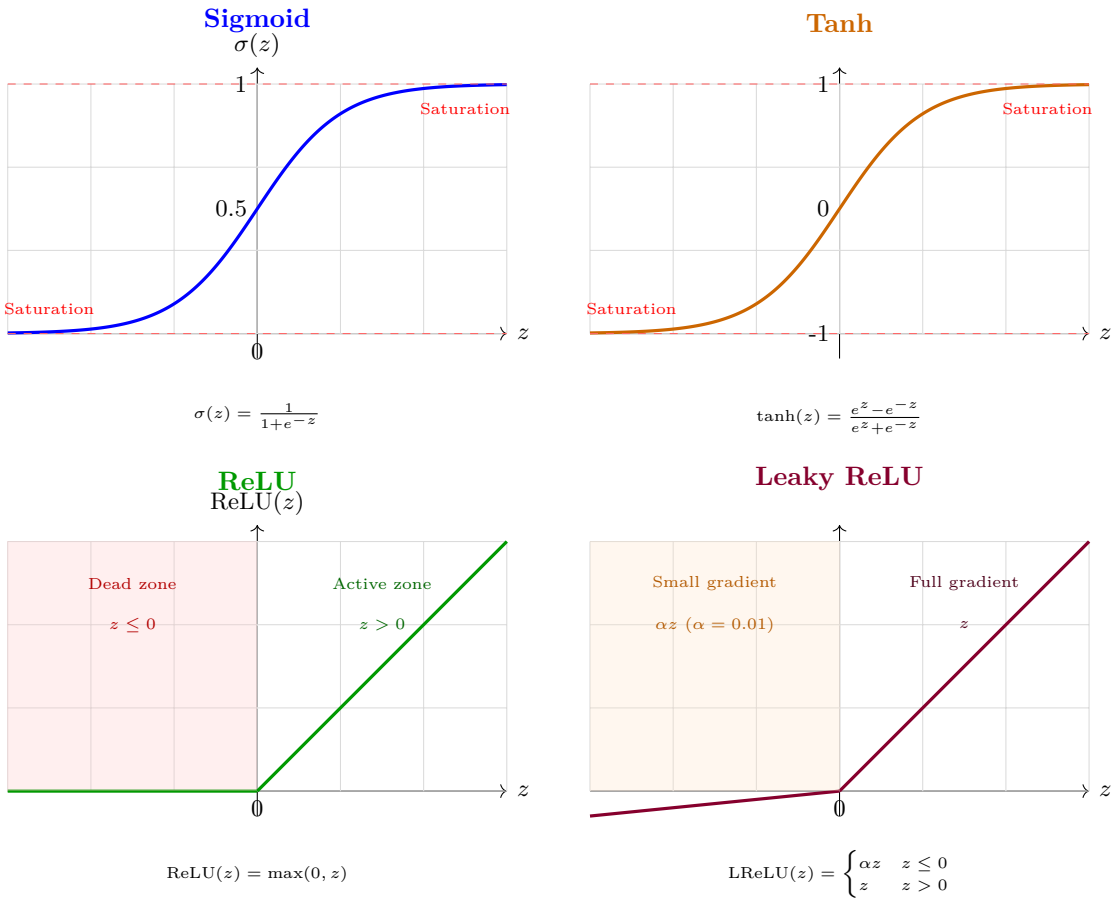

FIGURE 1 – Comparaison des principales fonctions d'activation. **Sigmoid et Tanh** souffrent de saturation (gradients ≈ 0 pour $|z|$ grand), causant le *vanishing gradient*. **ReLU** résout ce problème mais peut mourir (dead neurons) si $z \leq 0$ toujours. **Leaky ReLU** évite la mort neuronale avec un petit gradient négatif ($\alpha = 0.01$). En pratique, ReLU est le choix par défaut pour les couches cachées.

4 Perceptron Multi-Couches (MLP)

4.1 Architecture

Un MLP est composé de plusieurs couches de neurones :

- **Couche d'entrée (input layer)** : reçoit les features $\mathbf{x}$
- **Couches cachées (hidden layers)** : transformations non-linéaires
- **Couche de sortie (output layer)** : prédictions $\hat{y}$

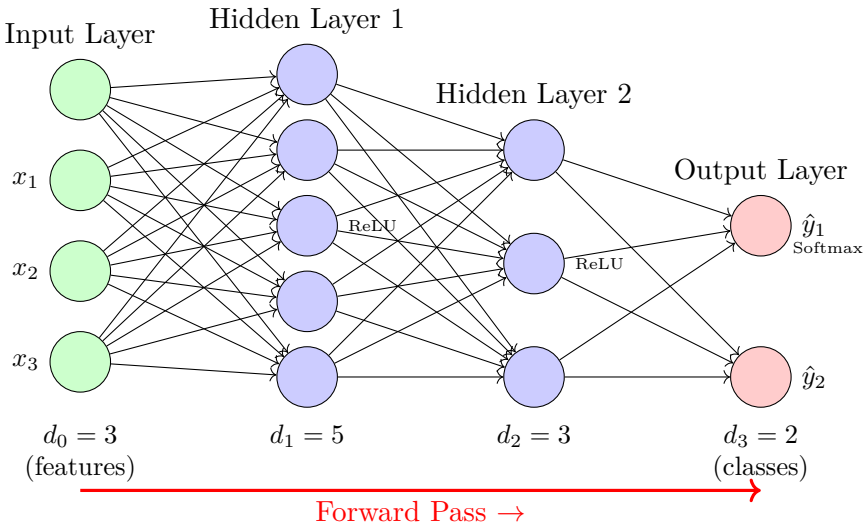

FIGURE 2 – Architecture MLP (Multi-Layer Perceptron) : les neurones d'une couche sont connectés à tous les neurones de la couche suivante (fully-connected). Les activations se propagent de gauche à droite via transformations linéaires ($\mathbf{Wa} + \mathbf{b}$) suivies de fonctions d'activation non-linéaires (ReLU, Softmax).

Définition

MLP à L couches Pour une entrée $\mathbf{x} \in \mathbb{R}^{d_0}$, un MLP calcule :

$$\mathbf{a}^{[0]} = \mathbf{x} \tag{10}$$

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \quad \text{pour } l = 1, \ldots, L \tag{11}$$

$$\mathbf{a}^{[l]} = \sigma^{[l]}(\mathbf{z}^{[l]}) \tag{12}$$

$$\hat{y} = \mathbf{a}^{[L]} \tag{13}$$

où :

- $\mathbf{W}^{[l]} \in \mathbb{R}^{d_l \times d_{l-1}}$: matrice de poids de la couche l
- $\mathbf{b}^{[l]} \in \mathbb{R}^{d_l}$: vecteur de biais de la couche l
- $\sigma^{[l]}$: fonction d'activation de la couche l
- $\mathbf{a}^{[l]}$: activations de la couche l
- $\mathbf{z}^{[l]}$: pré-activations (avant fonction d'activation)

4.2 Théorème d'approximation universelle

Théorème: Approximation Universelle

Un MLP avec une seule couche cachée de taille suffisante et une fonction d'activation non-linéaire peut approximer n'importe quelle fonction continue sur un compact de $\mathbb{R}^d$ avec une précision arbitraire.

/!\ Attention

Ce théorème est **théorique** : en pratique, les réseaux profonds (deep learning) avec plusieurs couches sont plus efficaces et nécessitent moins de neurones.

4.3 Exemple : MLP 3 couches

Architecture : 784 (input) → 128 (hidden) → 64 (hidden) → 10 (output)

$$\mathbf{x} \in \mathbb{R}^{784} \quad \text{(image } 28\times28\text{)} \tag{14}$$
$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]}\mathbf{x} + \mathbf{b}^{[1]} \quad (\mathbf{W}^{[1]} \in \mathbb{R}^{128\times784}) \tag{15}$$
$$\mathbf{a}^{[1]} = \text{ReLU}(\mathbf{z}^{[1]}) \quad \in \mathbb{R}^{128} \tag{16}$$
$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]}\mathbf{a}^{[1]} + \mathbf{b}^{[2]} \quad (\mathbf{W}^{[2]} \in \mathbb{R}^{64\times128}) \tag{17}$$
$$\mathbf{a}^{[2]} = \text{ReLU}(\mathbf{z}^{[2]}) \quad \in \mathbb{R}^{64} \tag{18}$$
$$\mathbf{z}^{[3]} = \mathbf{W}^{[3]}\mathbf{a}^{[2]} + \mathbf{b}^{[3]} \quad (\mathbf{W}^{[3]} \in \mathbb{R}^{10\times64}) \tag{19}$$
$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{[3]}) \quad \in \mathbb{R}^{10} \tag{20}$$

Nombre total de paramètres :

$$(784 \times 128 + 128) + (128 \times 64 + 64) + (64 \times 10 + 10) = 109{,}386 \tag{21}$$

5 Forward Pass (Propagation Avant)

5.1 Principe

Le forward pass consiste à calculer la sortie du réseau pour une entrée donnée en propageant les activations couche par couche.

Algorithm 2 Forward Pass

Require: Entrée $\mathbf{x}$, poids $\{\mathbf{W}^{[l]}, \mathbf{b}^{[l]}\}_{l=1}^{L}$
Ensure: Prédiction $\hat{y}$, cache des activations $\{\mathbf{a}^{[l]}, \mathbf{z}^{[l]}\}$

$\mathbf{a}^{[0]} = \mathbf{x}$
for $l = 1$ **to** L **do**
  $\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$ {Combinaison linéaire}
  $\mathbf{a}^{[l]} = \sigma^{[l]}(\mathbf{z}^{[l]})$ {Activation non-linéaire}
  Stocker $\mathbf{z}^{[l]}, \mathbf{a}^{[l]}$ dans le cache {Pour backprop}
end for
$\hat{y} = \mathbf{a}^{[L]}$
return $\hat{y}$, cache

5.2 Implémentation vectorisée

Pour un batch de m exemples $\mathbf{X} \in \mathbb{R}^{m \times d_0}$:

$$\mathbf{A}^{[0]} = \mathbf{X} \quad \in \mathbb{R}^{m \times d_0} \tag{22}$$

$$\mathbf{Z}^{[l]} = \mathbf{A}^{[l-1]}(\mathbf{W}^{[l]})^T + \mathbf{b}^{[l]} \quad \in \mathbb{R}^{m \times d_l} \tag{23}$$

$$\mathbf{A}^{[l]} = \sigma^{[l]}(\mathbf{Z}^{[l]}) \tag{24}$$

```python
def forward_pass(X, parameters):
    """
    X : (m, d0) - batch de m exemples
    parameters : dict avec W[l], b[l] pour chaque couche l
    """
    cache = {}
    A = X
    L = len(parameters) // 2  # Nombre de couches

    for l in range(1, L + 1):
        A_prev = A
        W = parameters[f'W{l}']
        b = parameters[f'b{l}']

        # Linear forward
        Z = np.dot(A_prev, W.T) + b

        # Activation
        if l < L:  # Hidden layers : ReLU
            A = np.maximum(0, Z)
        else:  # Output layer : softmax
            A = softmax(Z)

        # Cache pour backprop
        cache[f'Z{l}'] = Z
        cache[f'A{l}'] = A
```

```
27          cache[f'A{l-1}'] = A_prev
28
29      return A, cache
```

Listing 2 – Forward pass vectorisé

6 Fonctions de Coût

6.1 Pour la régression : MSE

$$L(\theta) = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2 \tag{25}$$

6.2 Pour classification binaire : Binary Cross-Entropy

$$L(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\right] \tag{26}$$

6.3 Pour classification multi-classe : Categorical Cross-Entropy

Définition

Cross-Entropy Loss Pour K classes, avec labels one-hot $\mathbf{y} \in \{0,1\}^K$:

$$L(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{i,k}\log(\hat{y}_{i,k}) \tag{27}$$

où $\hat{\mathbf{y}}_i = \text{softmax}(\mathbf{z}_i^{[L]})$.

En pratique, on utilise souvent la formulation :

$$L = -\frac{1}{m}\sum_{i=1}^{m}\log(\hat{y}_{i,c_i}) \tag{28}$$

où c_i est la classe correcte pour l'exemple i.

7 Backpropagation

7.1 Principe

La **backpropagation** (rétropropagation) est l'algorithme qui permet de calculer efficacement les gradients de la fonction de coût par rapport à tous les paramètres du réseau.

Idée clé : Utiliser la **règle de la chaîne (chain rule)** pour propager les gradients de la sortie vers l'entrée.

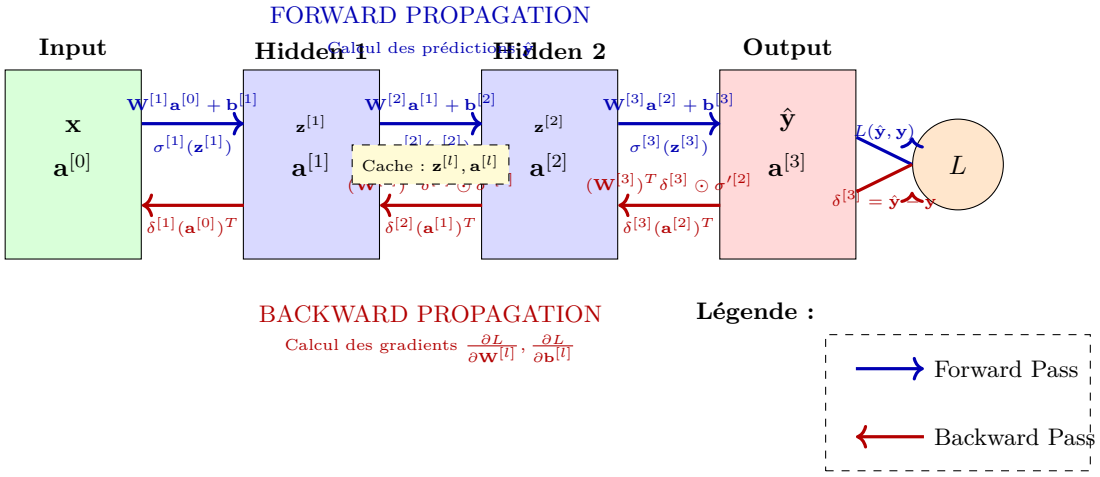

FIGURE 3 – Flux bidirectionnel Forward et Backward Propagation dans un MLP à 2 couches cachées. **Forward Pass (bleu) :** les activations se propagent de gauche à droite pour calculer $\hat{\mathbf{y}}$, puis la loss $L(\hat{\mathbf{y}}, \mathbf{y})$. Les valeurs intermédiaires $(\mathbf{z}^{[l]}, \mathbf{a}^{[l]})$ sont stockées dans un cache. **Backward Pass (rouge) :** les gradients se propagent de droite à gauche via la règle de la chaîne, calculant $\frac{\partial L}{\partial \mathbf{W}^{[l]}}$ et $\frac{\partial L}{\partial \mathbf{b}^{[l]}}$ pour chaque couche. Le gradient initial $\delta^{[L]} = \hat{\mathbf{y}} - \mathbf{y}$ (pour cross-entropy + softmax).

7.2 Notations

Définissons :

$$\delta^{[l]} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \quad \text{(gradient par rapport aux pré-activations)} \tag{29}$$

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} \quad \text{(gradient par rapport aux poids)} \tag{30}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} \quad \text{(gradient par rapport aux biais)} \tag{31}$$

7.3 Dérivation des équations

Couche de sortie ($l = L$) :

Pour cross-entropy avec softmax :

$$\delta^{[L]} = \hat{\mathbf{y}} - \mathbf{y} \quad \text{(formule simplifiée !)} \tag{32}$$

Couches cachées ($l < L$) :

Par la règle de la chaîne :

$$\delta^{[l]} = \frac{\partial L}{\partial \mathbf{z}^{[l]}} \tag{33}$$

$$= \frac{\partial L}{\partial \mathbf{a}^{[l]}} \odot \frac{\partial \mathbf{a}^{[l]}}{\partial \mathbf{z}^{[l]}} \tag{34}$$

$$= \left[(\mathbf{W}^{[l+1]})^T \delta^{[l+1]} \right] \odot \sigma'^{[l]}(\mathbf{z}^{[l]}) \tag{35}$$

où $\odot$ est le produit élément par élément (Hadamard).

Gradients des paramètres :

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \delta^{[l]} (\mathbf{a}^{[l-1]})^T \tag{36}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \delta^{[l]} \tag{37}$$

Pour un batch de m exemples :

$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} \mathbf{\Delta}^{[l]} (\mathbf{A}^{[l-1]})^T \tag{38}$$

$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \delta_i^{[l]} \tag{39}$$

7.4 Algorithme complet

Algorithm 3 Backpropagation

Require: Cache du forward pass $\{\mathbf{a}^{[l]}, \mathbf{z}^{[l]}\}$
Require: Gradient de la loss $\frac{\partial L}{\partial \mathbf{a}^{[L]}}$
Ensure: Gradients $\{\frac{\partial L}{\partial \mathbf{W}^{[l]}}, \frac{\partial L}{\partial \mathbf{b}^{[l]}}\}$

Calculer $\delta^{[L]} = \hat{\mathbf{y}} - \mathbf{y}$ {Output layer}
for $l = L$ **to** 1 **do**
  {Parcourir en sens inverse}
  $\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m}\delta^{[l]}(\mathbf{a}^{[l-1]})^T$
  $\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m}\sum_i \delta_i^{[l]}$
  if $l > 1$ **then**
    $\delta^{[l-1]} = [(\mathbf{W}^{[l]})^T\delta^{[l]}] \odot \sigma'^{[l-1]}(\mathbf{z}^{[l-1]})$
  end if
end for
return Gradients

7.5 Dérivées des fonctions d'activation

— **Sigmoid :** $\sigma'(z) = \sigma(z)(1 - \sigma(z))$
— **Tanh :** $\tanh'(z) = 1 - \tanh^2(z)$
— **ReLU :** $\text{ReLU}'(z) = \mathbb{1}[z > 0]$
— **Leaky ReLU :** $\text{LReLU}'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z \leq 0 \end{cases}$

7.6 Implémentation

```python
def backward_pass(y_true, cache, parameters):
    """
    y_true : (m, K) - labels one-hot
    cache : dict des activations du forward pass
    parameters : dict des poids W[l], b[l]
    """
    m = y_true.shape[0]
    grads = {}
    L = len(parameters) // 2

    # Output layer gradient (cross-entropy + softmax)
    y_pred = cache[f'A{L}']
    dZ = y_pred - y_true  # Shape: (m, K)

    # Backprop travers les couches
    for l in reversed(range(1, L + 1)):
        A_prev = cache[f'A{l-1}']

        # Gradients des param tres
        grads[f'dW{l}'] = (1/m) * np.dot(dZ.T, A_prev)
        grads[f'db{l}'] = (1/m) * np.sum(dZ, axis=0, keepdims=True)
```

```
22
23          if l > 1:
24              # Gradient pour la couche pr c dente
25              W = parameters[f'W{l}']
26              dA_prev = np.dot(dZ, W)
27
28              # Gradient    travers l'activation (ReLU)
29              Z_prev = cache[f'Z{l-1}']
30              dZ = dA_prev * (Z_prev > 0)  # ReLU derivative
31
32      return grads
```

Listing 3 – Backpropagation

8 Entraînement d'un MLP

8.1 Optimiseurs

8.1.1 Gradient Descent

$$\theta_{t+1} = \theta_t - \alpha \nabla L(\theta_t) \tag{40}$$

8.1.2 Stochastic Gradient Descent (SGD)

Mise à jour sur un seul exemple (ou un mini-batch) :

$$\theta_{t+1} = \theta_t - \alpha \nabla L_i(\theta_t) \tag{41}$$

Avec momentum :

$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + (1-\beta)\nabla L(\theta_t) \tag{42}$$

$$\theta_{t+1} = \theta_t - \alpha \mathbf{v}_t \tag{43}$$

Typiquement $\beta = 0.9$.

8.1.3 Adam (Adaptive Moment Estimation)

Combine momentum et RMSprop :

$$\mathbf{m}_t = \beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\nabla L(\theta_t) \quad \text{(1st moment)} \tag{44}$$

$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)(\nabla L(\theta_t))^2 \quad \text{(2nd moment)} \tag{45}$$

$$\hat{\mathbf{m}}_t = \frac{\mathbf{m}_t}{1-\beta_1^t} \quad \text{(bias correction)} \tag{46}$$

$$\hat{\mathbf{v}}_t = \frac{\mathbf{v}_t}{1-\beta_2^t} \tag{47}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon} \tag{48}$$

Hyperparamètres par défaut : $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.

(i) Astuce

Adam est l'optimiseur par défaut pour la plupart des problèmes deep learning. Il est robuste et nécessite peu de tuning.

8.2 Algorithme d'entraînement complet

Algorithm 4 Entraînement MLP

Require: Dataset (X, y), architecture $\{d_0, d_1, \ldots, d_L\}$
Require: Learning rate α, batch size B, nombre d'epochs T
Ensure: Paramètres entraînés θ^*

```
Initialiser aléatoirement W^[l], b^[l] pour l = 1, ..., L
for epoch = 1 to T do
    Mélanger les données
    for chaque mini-batch (X_batch, y_batch) do
        {Forward pass}
        ŷ_batch, cache = forward_pass(X_batch, θ)
        L_batch = compute_loss(ŷ_batch, y_batch)
        {Backward pass}
        grads = backward_pass(y_batch, cache, θ)
        {Update parameters}
        for chaque paramètre θ do
            θ ← θ − α · ∂L/∂θ
        end for
    end for
    Évaluer sur validation set
end for
return θ*
```

8.3 Initialisation des poids

/!\ Attention

Ne JAMAIS initialiser tous les poids à 0 ! Les neurones seraient tous identiques (symétrie).

Xavier/Glorot initialization (pour Sigmoid/Tanh) :

$$W_{ij}^{[l]} \sim \mathcal{N}\left(0, \frac{2}{d_{l-1} + d_l}\right) \tag{49}$$

He initialization (pour ReLU) :

$$W_{ij}^{[l]} \sim \mathcal{N}\left(0, \frac{2}{d_{l-1}}\right) \tag{50}$$

```python
def initialize_parameters(layer_dims):
    """
    layer_dims : [d0, d1, d2, ..., dL]
    """
    parameters = {}
    L = len(layer_dims) - 1

    for l in range(1, L + 1):
        # He initialization
        parameters[f'W{l}'] = np.random.randn(
            layer_dims[l], layer_dims[l-1]
        ) * np.sqrt(2 / layer_dims[l-1])

        parameters[f'b{l}'] = np.zeros((1, layer_dims[l]))

    return parameters
```

Listing 4 – Initialisation He

9 Régularisation

9.1 L2 Regularization (Weight Decay)

Ajouter un terme de pénalité sur les poids :

$$L_{reg}(\theta) = L(\theta) + \frac{\lambda}{2m} \sum_{l=1}^{L} \|\mathbf{W}^{[l]}\|_F^2 \tag{51}$$

où $\|\mathbf{W}\|_F^2 = \sum_{i,j} W_{ij}^2$ est la norme de Frobenius.

Le gradient devient :

$$\frac{\partial L_{reg}}{\partial \mathbf{W}^{[l]}} = \frac{\partial L}{\partial \mathbf{W}^{[l]}} + \frac{\lambda}{m} \mathbf{W}^{[l]} \tag{52}$$

9.2 Dropout

Définition

Dropout Pendant l'entraînement, chaque neurone est désactivé avec probabilité p (typiquement $p = 0.5$). Au test, on utilise tous les neurones mais on multiplie les sorties par $(1 - p)$.

Implémentation (inverted dropout) :

```python
def dropout_forward(A, keep_prob=0.5, training=True):
    if training:
        mask = np.random.rand(*A.shape) < keep_prob
        A = A * mask / keep_prob  # Inverted dropout
        return A, mask
    else:
```

```
        return A, None
```

(i) Astuce

Le dropout agit comme un **ensemble de réseaux** : à chaque itération, on entraîne un sous-réseau différent.

9.3 Batch Normalization

Normaliser les activations à chaque couche :

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} z_i^{[l]} \tag{53}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(z_i^{[l]} - \mu_B)^2 \tag{54}$$

$$\hat{z}_i^{[l]} = \frac{z_i^{[l]} - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{55}$$

$$\tilde{z}_i^{[l]} = \gamma \hat{z}_i^{[l]} + \beta \tag{56}$$

où γ, β sont des paramètres apprenables.

Avantages :

- Accélère l'entraînement (learning rates plus élevés)
- Réduit la sensibilité à l'initialisation
- Effet régularisant (similaire au dropout)

9.4 Early Stopping

Surveiller la loss sur le validation set et arrêter l'entraînement quand elle commence à augmenter.

```
best_val_loss = float('inf')
patience = 10
counter = 0

for epoch in range(n_epochs):
    train_model()
    val_loss = evaluate_validation()

    if val_loss < best_val_loss:
        best_val_loss = val_loss
        save_checkpoint()
        counter = 0
    else:
        counter += 1
        if counter >= patience:
            print("Early stopping")
            break
```

10 Implémentation Complète avec NumPy

```python
import numpy as np

class MLP:
    def __init__(self, layer_dims, learning_rate=0.01):
        self.layer_dims = layer_dims
        self.lr = learning_rate
        self.parameters = self._initialize_parameters()
        self.L = len(layer_dims) - 1

    def _initialize_parameters(self):
        """He initialization"""
        params = {}
        for l in range(1, len(self.layer_dims)):
            params[f'W{l}'] = np.random.randn(
                self.layer_dims[l], self.layer_dims[l-1]
            ) * np.sqrt(2 / self.layer_dims[l-1])
            params[f'b{l}'] = np.zeros((1, self.layer_dims[l]))
        return params

    def _relu(self, Z):
        return np.maximum(0, Z)

    def _softmax(self, Z):
        expZ = np.exp(Z - np.max(Z, axis=1, keepdims=True))
        return expZ / np.sum(expZ, axis=1, keepdims=True)

    def forward(self, X):
        """Forward pass"""
        cache = {'A0': X}
        A = X

        for l in range(1, self.L + 1):
            Z = np.dot(A, self.parameters[f'W{l}'].T) + \
                self.parameters[f'b{l}']

            if l < self.L:
                A = self._relu(Z)
            else:
                A = self._softmax(Z)

            cache[f'Z{l}'] = Z
            cache[f'A{l}'] = A

        return A, cache

    def backward(self, y_true, cache):
        """Backpropagation"""
        m = y_true.shape[0]
        grads = {}
```

```
        # Output layer
        dZ = cache[f'A{self.L}'] - y_true

        for l in reversed(range(1, self.L + 1)):
            A_prev = cache[f'A{l-1}']
            grads[f'dW{l}'] = (1/m) * np.dot(dZ.T, A_prev)
            grads[f'db{l}'] = (1/m) * np.sum(dZ, axis=0, keepdims=True)

            if l > 1:
                W = self.parameters[f'W{l}']
                dA_prev = np.dot(dZ, W)
                dZ = dA_prev * (cache[f'Z{l-1}'] > 0)

        return grads

    def update_parameters(self, grads):
        """Gradient descent update"""
        for l in range(1, self.L + 1):
            self.parameters[f'W{l}'] -= self.lr * grads[f'dW{l}']
            self.parameters[f'b{l}'] -= self.lr * grads[f'db{l}']

    def compute_loss(self, y_pred, y_true):
        """Cross-entropy loss"""
        m = y_true.shape[0]
        loss = -np.sum(y_true * np.log(y_pred + 1e-8)) / m
        return loss

    def fit(self, X, y, epochs=100, batch_size=32, verbose=True):
        """Training loop"""
        m = X.shape[0]
        history = {'loss': []}

        for epoch in range(epochs):
            # Shuffle
            indices = np.random.permutation(m)
            X_shuffled = X[indices]
            y_shuffled = y[indices]

            epoch_loss = 0
            n_batches = m // batch_size

            for i in range(n_batches):
                start = i * batch_size
                end = start + batch_size
                X_batch = X_shuffled[start:end]
                y_batch = y_shuffled[start:end]

                # Forward
                y_pred, cache = self.forward(X_batch)
                loss = self.compute_loss(y_pred, y_batch)
```

```
                epoch_loss += loss

                # Backward
                grads = self.backward(y_batch, cache)

                # Update
                self.update_parameters(grads)

            avg_loss = epoch_loss / n_batches
            history['loss'].append(avg_loss)

            if verbose and (epoch + 1) % 10 == 0:
                print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}")

        return history

    def predict(self, X):
        """Predictions"""
        y_pred, _ = self.forward(X)
        return np.argmax(y_pred, axis=1)

# Exemple d'utilisation
if __name__ == "__main__":
    from sklearn.datasets import load_digits
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import OneHotEncoder

    # Charger donn es
    digits = load_digits()
    X, y = digits.data, digits.target

    # Normaliser
    X = X / 16.0

    # One-hot encoding
    y_onehot = np.zeros((len(y), 10))
    y_onehot[np.arange(len(y)), y] = 1

    # Split
    X_train, X_test, y_train, y_test = train_test_split(
        X, y_onehot, test_size=0.2, random_state=42
    )

    # Entra ner
    mlp = MLP(layer_dims=[64, 128, 64, 10], learning_rate=0.1)
    history = mlp.fit(X_train, y_train, epochs=100, batch_size=32)

    #   valuer
    y_pred = mlp.predict(X_test)
    y_test_labels = np.argmax(y_test, axis=1)
    accuracy = np.mean(y_pred == y_test_labels)
```

```
    print(f"\nTest Accuracy: {accuracy:.4f}")
```

Listing 5 – MLP complet from scratch

11 Implémentation avec PyTorch

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, TensorDataset

class MLP_PyTorch(nn.Module):
    def __init__(self, input_dim, hidden_dims, output_dim):
        super(MLP_PyTorch, self).__init__()

        layers = []
        prev_dim = input_dim

        # Hidden layers
        for hidden_dim in hidden_dims:
            layers.append(nn.Linear(prev_dim, hidden_dim))
            layers.append(nn.ReLU())
            layers.append(nn.Dropout(0.2))
            prev_dim = hidden_dim

        # Output layer
        layers.append(nn.Linear(prev_dim, output_dim))

        self.network = nn.Sequential(*layers)

    def forward(self, x):
        return self.network(x)

# Exemple d'utilisation
model = MLP_PyTorch(input_dim=64, hidden_dims=[128, 64], output_dim=10)

# Loss et optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Training loop
def train_pytorch(model, train_loader, epochs=50):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for X_batch, y_batch in train_loader:
            # Forward
            outputs = model(X_batch)
            loss = criterion(outputs, y_batch)

```

```
            # Backward
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

        if (epoch + 1) % 10 == 0:
            print(f"Epoch {epoch+1}, Loss: {total_loss/len(train_loader)
    :.4f}")

# Utilisation
X_train_tensor = torch.FloatTensor(X_train)
y_train_tensor = torch.LongTensor(np.argmax(y_train, axis=1))

train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)

train_pytorch(model, train_loader, epochs=50)
```

Listing 6 – MLP avec PyTorch

12 Diagnostic et Debugging

12.1 Problèmes courants

TABLE 2 – Diagnostic des problèmes d'entraînement

Symptôme	**Cause probable**	**Solution**
Loss = NaN	Explosion gradients	Réduire learning rate Gradient clipping Vérifier normalisation
Loss ne diminue pas	Learning rate trop faible Mauvaise init Architecture inadaptée	Augmenter LR He/Xavier init Changer nb couches
Overfitting	Trop de paramètres Pas assez de données	Dropout, L2 reg Data augmentation Early stopping
Underfitting	Modèle trop simple Régularisation trop forte	Ajouter couches/neurones Réduire λ, dropout
Vanishing gradients	Sigmoid/Tanh profond	Utiliser ReLU Batch normalization Residual connections

12.2 Gradient checking

Vérifier l'implémentation de backprop en comparant avec gradient numérique :

$$\frac{\partial L}{\partial \theta} \approx \frac{L(\theta + \epsilon) - L(\theta - \epsilon)}{2\epsilon} \tag{57}$$

```python
def gradient_check(model, X, y, epsilon=1e-7):
    """V rifie backprop avec gradients num riques"""
    # Calculer gradients analytiques (backprop)
    y_pred, cache = model.forward(X)
    grads_analytic = model.backward(y, cache)

    # Calculer gradients num riques
    for param_name in model.parameters:
        param = model.parameters[param_name]
        grad_numeric = np.zeros_like(param)

        it = np.nditer(param, flags=['multi_index'])
        while not it.finished:
            idx = it.multi_index
            old_value = param[idx]

            # f(theta + epsilon)
            param[idx] = old_value + epsilon
            y_pred_plus, _ = model.forward(X)
            loss_plus = model.compute_loss(y_pred_plus, y)

            # f(theta - epsilon)
            param[idx] = old_value - epsilon
            y_pred_minus, _ = model.forward(X)
            loss_minus = model.compute_loss(y_pred_minus, y)

            # Gradient num rique
            grad_numeric[idx] = (loss_plus - loss_minus) / (2 * epsilon)

            param[idx] = old_value
            it.iternext()

        # Comparer
        grad_analytic = grads_analytic[f'd{param_name}']
        diff = np.linalg.norm(grad_numeric - grad_analytic) / \
               (np.linalg.norm(grad_numeric) + np.linalg.norm(
   grad_analytic))

        print(f"{param_name}: diff = {diff:.2e}")
        if diff < 1e-7:
            print("      Gradient correct")
        else:
            print("      Erreur dans backprop!")
```

Listing 7 – Gradient checking

13 Avantages et Limites

13.1 Avantages

— ✓ Modèle les relations non-linéaires complexes
— ✓ Approximation universelle (théoriquement)
— ✓ Flexible : régression, classification, séries temporelles
— ✓ Apprentissage de représentations (features automatiques)
— ✓ Parallélisation sur GPU

13.2 Limites

— × Boîte noire (difficile à interpréter)
— × Nécessite beaucoup de données
— × Sensible aux hyperparamètres
— × Risque d'overfitting
— × Temps d'entraînement long
— × Optima locaux (pas de garantie de convergence globale)

13.3 Quand utiliser un MLP ?

(i) Astuce

Un MLP est particulièrement adapté quand :
— Les données sont tabulaires (non structurées spatialement)
— Relations non-linéaires complexes
— Beaucoup de données disponibles
— Performance > interprétabilité

/!\ Attention

Préférer d'autres modèles dans les cas suivants :
— Petites données (< 1000 exemples) → Random Forest, SVM
— Images → CNN (Chapitre 07)
— Séquences/texte → RNN, Transformers (Chapitre 08)
— Besoin d'interprétabilité → Arbres de décision, régression linéaire

14 Hyperparamètres et Tuning

14.1 Stratégies de tuning

1. **Grid Search** : Tester toutes les combinaisons (coûteux)
2. **Random Search** : Échantillonner aléatoirement (souvent meilleur)
3. **Bayesian Optimization** : Optimisation intelligente (Optuna, Hyperopt)

```
from sklearn.model_selection import RandomizedSearchCV
from sklearn.neural_network import MLPClassifier

```

TABLE 3 – Hyperparamètres principaux d'un MLP

Paramètre	Valeurs typiques	Impact
Architecture		
Nombre de couches	2-5	Capacité du modèle
Neurones par couche	64, 128, 256, 512	Capacité, overfitting
Optimisation		
Learning rate	10^{-4} à 10^{-1}	Vitesse/stabilité
Batch size	32, 64, 128, 256	Vitesse, généralisation
Optimizer	SGD, Adam	Convergence
Régularisation		
Dropout	0.2, 0.5	Overfitting
L2 weight decay	10^{-5} à 10^{-3}	Overfitting
Autres		
Activation	ReLU, Leaky ReLU	Gradient flow
Initialisation	He, Xavier	Convergence initiale

```python
param_distributions = {
    'hidden_layer_sizes': [(64,), (128,), (64, 32), (128, 64)],
    'activation': ['relu', 'tanh'],
    'alpha': [0.0001, 0.001, 0.01],  # L2 regularization
    'learning_rate_init': [0.001, 0.01, 0.1],
    'batch_size': [32, 64, 128]
}

mlp = MLPClassifier(max_iter=100)
random_search = RandomizedSearchCV(
    mlp, param_distributions, n_iter=20, cv=3, n_jobs=-1
)

random_search.fit(X_train, y_train)
print("Best params:", random_search.best_params_)
```

Listing 8 – Random search avec scikit-learn

15 Applications Pratiques

1. **Classification d'images** : MNIST, CIFAR-10 (avant CNN)
2. **Reconnaissance vocale** : Phonèmes, commandes vocales
3. **Prédiction de séries temporelles** : Finance, météo
4. **Systèmes de recommandation** : Collaborative filtering
5. **Détection de fraude** : Transactions bancaires
6. **Diagnostic médical** : Prédiction de maladies à partir de biomarqueurs
7. **NLP** : Sentiment analysis, classification de textes

16 Résumé du Chapitre

16.1 Points Clés

— **Perceptron** : Classification binaire linéaire (limité au linéairement séparable)
— **MLP** : Empilage de couches avec activations non-linéaires (approximation universelle)
— **Forward pass** : Propagation des activations de l'entrée à la sortie
— **Backpropagation** : Calcul efficace des gradients via la chaîne rule
— **Fonctions d'activation** : ReLU (défaut), sigmoid (output binaire), softmax (multi-classe)
— **Optimiseurs** : SGD, Adam (défaut), momentum
— **Régularisation** : Dropout, L2, batch norm, early stopping
— **Initialisation** : He (ReLU), Xavier (sigmoid/tanh)

16.2 Formules Essentielles

Formules à retenir

Forward pass (couche l) :

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]}\mathbf{a}^{[l-1]} + \mathbf{b}^{[l]}$$
$$\mathbf{a}^{[l]} = \sigma^{[l]}(\mathbf{z}^{[l]})$$

Backpropagation :

$$\delta^{[l]} = [(\mathbf{W}^{[l+1]})^T \delta^{[l+1]}] \odot \sigma'^{[l]}(\mathbf{z}^{[l]})$$
$$\frac{\partial L}{\partial \mathbf{W}^{[l]}} = \delta^{[l]}(\mathbf{a}^{[l-1]})^T$$
$$\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \delta^{[l]}$$

Cross-Entropy + Softmax :

$$L = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K} y_{i,k}\log(\hat{y}_{i,k})$$
$$\delta^{[L]} = \hat{\mathbf{y}} - \mathbf{y}$$

Gradient Descent :

$$\theta \leftarrow \theta - \alpha\nabla L(\theta)$$

17 Exercices

17.1 Questions de compréhension

1. Pourquoi le perceptron ne peut-il pas résoudre le problème XOR ?
2. Que se passe-t-il si on initialise tous les poids à 0 ?
3. Expliquer le problème du vanishing gradient avec la fonction sigmoid.
4. Pourquoi ReLU est-il préféré à sigmoid dans les couches cachées ?

5. Quelle est la différence entre batch, epoch et iteration ?
6. Pourquoi utilise-t-on softmax + cross-entropy pour la classification multi-classe ?

17.2 Exercices pratiques

1. **Perceptron from scratch**
   — Implémenter le perceptron en NumPy
   — Tester sur un dataset linéairement séparable (make_classification)
   — Visualiser la frontière de décision
2. **MLP pour MNIST**
   — Charger le dataset MNIST (sklearn.datasets.load_digits)
   — Entraîner un MLP from scratch (architecture libre)
   — Comparer avec MLPClassifier de scikit-learn
   — Objectif : $> 95\%$ accuracy
3. **Gradient checking**
   — Implémenter le gradient checking
   — Vérifier votre implémentation de backprop
4. **Régularisation**
   — Comparer MLP avec/sans dropout
   — Tester différentes valeurs de L2 regularization
   — Tracer les courbes d'apprentissage
5. **Hyperparameter tuning**
   — Utiliser RandomizedSearchCV pour trouver les meilleurs hyperparamètres
   — Comparer GridSearch vs RandomSearch

Solutions disponibles dans `06_exercices.ipynb` *(solutions intégrées dans le notebook)*

18 Pour Aller Plus Loin

18.1 Lectures Recommandées

— **Deep Learning Book** (Goodfellow et al., 2016) - Chapitres 6-8
— **Neural Networks and Deep Learning** (Michael Nielsen) - En ligne gratuit
— Rumelhart et al. (1986) - "Learning representations by back-propagating errors"
— Glorot & Bengio (2010) - "Understanding the difficulty of training deep feedforward neural networks"

18.2 Ressources en Ligne

— Playground TensorFlow : https://playground.tensorflow.org/
— 3Blue1Brown - Neural Networks : https://www.youtube.com/watch?v=aircAruvnKk
— Documentation scikit-learn MLP : https://scikit-learn.org/stable/modules/neural_networks_supervised.html
— PyTorch Tutorials : https://pytorch.org/tutorials/beginner/basics/buildmodel_tutorial.html

18.3 Extensions

— **Residual Networks (ResNet)** : Connexions résiduelles pour réseaux très profonds
— **Batch Normalization** : Normalisation des activations
— **Autoencoders** : Apprentissage de représentations non supervisé
— **Transfer Learning** : Réutilisation de réseaux pré-entraînés

18.4 Prochaines Étapes

Chapitre suivant recommandé : **Chapitre 07 - Deep Learning : Réseaux de Neurones Convolutifs (CNN)**

Les CNN sont une architecture spécialisée pour les données spatiales (images) qui exploite la localité et l'invariance par translation.

Références

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
2. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). "Learning representations by back-propagating errors". *Nature*, 323(6088), 533-536.
3. LeCun, Y., Bengio, Y., & Hinton, G. (2015). "Deep learning". *Nature*, 521(7553), 436-444.
4. Kingma, D. P., & Ba, J. (2014). "Adam : A method for stochastic optimization". *arXiv preprint arXiv :1412.6980*.
5. Srivastava, N., et al. (2014). "Dropout : A simple way to prevent neural networks from overfitting". *JMLR*, 15(1), 1929-1958.
6. He, K., et al. (2015). "Delving deep into rectifiers : Surpassing human-level performance on ImageNet classification". *ICCV*.