

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

Table des matières

| | | |
|----------|--|-----------|
| 1 | Introduction aux Réseaux de Neurones | 2 |
| 1.1 | Motivation | 2 |
| 1.2 | Historique | 2 |
| 1.3 | Analogie biologique | 2 |
| 2 | Le Perceptron | 2 |
| 2.1 | Définition | 2 |
| 2.2 | Algorithme d'apprentissage | 3 |
| 2.3 | Théorème de convergence | 3 |
| 2.4 | Implémentation | 3 |
| 3 | Fonctions d'Activation | 4 |
| 3.1 | Sigmoid | 4 |
| 3.2 | Tanh (Tangente Hyperbolique) | 4 |
| 3.3 | ReLU (Rectified Linear Unit) | 5 |
| 3.4 | Variantes de ReLU | 5 |
| 3.5 | Softmax (pour classification multi-classe) | 5 |
| 4 | Perceptron Multi-Couches (MLP) | 6 |
| 4.1 | Architecture | 6 |
| 4.2 | Théorème d'approximation universelle | 6 |
| 4.3 | Exemple : MLP 3 couches | 6 |
| 5 | Forward Pass (Propagation Avant) | 7 |
| 5.1 | Principe | 7 |
| 5.2 | Implémentation vectorisée | 7 |
| 6 | Fonctions de Coût | 8 |
| 6.1 | Pour la régression : MSE | 8 |
| 6.2 | Pour classification binaire : Binary Cross-Entropy | 8 |
| 6.3 | Pour classification multi-classe : Categorical Cross-Entropy | 9 |
| 7 | Backpropagation | 9 |
| 7.1 | Principe | 9 |
| 7.2 | Notations | 9 |
| 7.3 | Dérivation des équations | 9 |
| 7.4 | Algorithme complet | 10 |
| 7.5 | Dérivées des fonctions d'activation | 10 |
| 7.6 | Implémentation | 11 |
| 8 | Entraînement d'un MLP | 11 |
| 8.1 | Optimiseurs | 11 |
| 8.1.1 | Gradient Descent | 11 |
| 8.1.2 | Stochastic Gradient Descent (SGD) | 12 |
| 8.1.3 | Adam (Adaptive Moment Estimation) | 12 |

| | | |
|-----------|---|-----------|
| 8.2 | Algorithme d'entraînement complet | 13 |
| 8.3 | Initialisation des poids | 13 |
| 9 | Régularisation | 14 |
| 9.1 | L2 Regularization (Weight Decay) | 14 |
| 9.2 | Dropout | 14 |
| 9.3 | Batch Normalization | 15 |
| 9.4 | Early Stopping | 15 |
| 10 | Implémentation Complète avec NumPy | 15 |
| 11 | Implémentation avec PyTorch | 18 |
| 12 | Diagnostic et Debugging | 20 |
| 12.1 | Problèmes courants | 20 |
| 12.2 | Gradient checking | 20 |
| 13 | Avantages et Limites | 21 |
| 13.1 | Avantages | 21 |
| 13.2 | Limites | 22 |
| 13.3 | Quand utiliser un MLP ? | 22 |
| 14 | Hyperparamètres et Tuning | 22 |
| 14.1 | Stratégies de tuning | 23 |
| 15 | Applications Pratiques | 23 |
| 16 | Résumé du Chapitre | 23 |
| 16.1 | Points Clés | 23 |
| 16.2 | Formules Essentielles | 24 |
| 17 | Exercices | 24 |
| 17.1 | Questions de compréhension | 24 |
| 17.2 | Exercices pratiques | 24 |
| 18 | Pour Aller Plus Loin | 25 |
| 18.1 | Lectures Recommandées | 25 |
| 18.2 | Ressources en Ligne | 25 |
| 18.3 | Extensions | 25 |
| 18.4 | Prochaines Étapes | 25 |

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

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

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 :

$$\mathbf{w}^T \mathbf{x} + b = 0 \quad (2)$$

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

```

1 : Initialiser  $\mathbf{w} = \mathbf{0}$ ,  $b = 0$ 
2 : for  $epoch = 1$  to  $T$  do
3 :   for  $i = 1$  to  $n$  do
4 :     Prédire :  $\hat{y}_i = \mathbb{I}[\mathbf{w}^T \mathbf{x}_i + b > 0]$ 
5 :     if  $\hat{y}_i \neq y_i$  then
6 :        $\mathbf{w} \leftarrow \mathbf{w} + \alpha(y_i - \hat{y}_i)\mathbf{x}_i$ 
7 :        $b \leftarrow b + \alpha(y_i - \hat{y}_i)$ 
8 :     end if
9 :   end for
10 : end for
11 : 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

```

1 import numpy as np
2
3 class Perceptron:
4     def __init__(self, learning_rate=0.01, n_epochs=100):
5         self.lr = learning_rate
6         self.n_epochs = n_epochs
7         self.w = None
8         self.b = 0
9
10    def fit(self, X, y):
11        n_samples, n_features = X.shape
12        self.w = np.zeros(n_features)
13
14        for epoch in range(self.n_epochs):
15            for i in range(n_samples):
16                # Forward pass
17                z = np.dot(X[i], self.w) + self.b
```

```

18         y_pred = 1 if z > 0 else 0
19
20         # Update weights si erreur
21         if y_pred != y[i]:
22             update = self.lr * (y[i] - y_pred)
23             self.w += update * X[i]
24             self.b += update
25
26         return self
27
28     def predict(self, X):
29         z = np.dot(X, self.w) + self.b
30         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

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

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)

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

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

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

Propriétés :

- Calcul très rapide
- Pas de vanishing gradient pour $z > 0$
- Dérivée : $\text{ReLU}'(z) = \mathbb{I}[z > 0]$
- 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 :

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

ELU (Exponential Linear Unit) :

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

Swish / SiLU :

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

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 :

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

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 |

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}

Définition : MLP à L couches

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

$$\mathbf{a}^{[0]} = \mathbf{x} \quad (10)$$

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

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

$$\hat{y} = \mathbf{a}^{[L]} \quad (13)$$

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)

$$\mathbf{x} \in \mathbb{R}^{784} \quad (\text{image } 28 \times 28) \quad (14)$$

$$\mathbf{z}^{[1]} = \mathbf{W}^{[1]} \mathbf{x} + \mathbf{b}^{[1]} \quad (\mathbf{W}^{[1]} \in \mathbb{R}^{128 \times 784}) \quad (15)$$

$$\mathbf{a}^{[1]} = \text{ReLU}(\mathbf{z}^{[1]}) \in \mathbb{R}^{128} \quad (16)$$

$$\mathbf{z}^{[2]} = \mathbf{W}^{[2]} \mathbf{a}^{[1]} + \mathbf{b}^{[2]} \quad (\mathbf{W}^{[2]} \in \mathbb{R}^{64 \times 128}) \quad (17)$$

$$\mathbf{a}^{[2]} = \text{ReLU}(\mathbf{z}^{[2]}) \in \mathbb{R}^{64} \quad (18)$$

$$\mathbf{z}^{[3]} = \mathbf{W}^{[3]} \mathbf{a}^{[2]} + \mathbf{b}^{[3]} \quad (\mathbf{W}^{[3]} \in \mathbb{R}^{10 \times 64}) \quad (19)$$

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{z}^{[3]}) \in \mathbb{R}^{10} \quad (20)$$

Nombre total de paramètres :

$$(784 \times 128 + 128) + (128 \times 64 + 64) + (64 \times 10 + 10) = 109,386 \quad (21)$$

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 $\{\mathbf{a}^{[l]}, \mathbf{z}^{[l]}\}$

```

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

```

5.2 Implémentation vectorisée

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

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

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

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

```

1 def forward_pass(X, parameters):
2     """

```

```

3  X : (m, d0) - batch de m exemples
4  parameters : dict avec W[l], b[l] pour chaque couche l
5  """
6  cache = {}
7  A = X
8  L = len(parameters) // 2 # Nombre de couches
9
10 for l in range(1, L + 1):
11     A_prev = A
12     W = parameters[f'W{l}']
13     b = parameters[f'b{l}']
14
15     # Linear forward
16     Z = np.dot(A_prev, W.T) + b
17
18     # Activation
19     if l < L: # Hidden layers : ReLU
20         A = np.maximum(0, Z)
21     else: # Output layer : softmax
22         A = softmax(Z)
23
24     # Cache pour backprop
25     cache[f'Z{l}'] = Z
26     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

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

6.2 Pour classification binaire : Binary Cross-Entropy

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

6.3 Pour classification multi-classe : Categorical Cross-Entropy

Définition : Cross-Entropy Loss

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

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

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

En pratique, on utilise souvent la formulation :

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

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.

7.2 Notations

Définissons :

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

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

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

7.3 Dérivation des équations

Couche de sortie ($l = L$) :

Pour cross-entropy avec softmax :

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

Couches cachées ($l < L$) :

Par la règle de la chaîne :

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

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

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

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

Gradients des paramètres :

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

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

Pour un batch de m exemples :

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

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

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 $\{\frac{\partial L}{\partial \mathbf{W}^{[l]}}, \frac{\partial L}{\partial \mathbf{b}^{[l]}}\}$

1 : Calculer $\delta^{[L]} = \hat{\mathbf{y}} - \mathbf{y}$ {Output layer}

2 : **for** $l = L$ **to** 1 **do**

{Parcourir en sens inverse}

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

4 : $\frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \sum_i \delta_i^{[l]}$

5 : **if** $l > 1$ **then**

6 : $\delta^{[l-1]} = [(\mathbf{W}^{[l]})^T \delta^{[l]}] \odot \sigma'^{[l-1]}(\mathbf{z}^{[l-1]})$

7 : **end if**

8 : **end for**

9 : **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** : $\text{ReLU}'(z) = \mathbb{K}[z > 0]$

— **Leaky ReLU** : $\text{LReLU}'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z \leq 0 \end{cases}$

7.6 Implémentation

```

1 def backward_pass(y_true, cache, parameters):
2     """
3     y_true : (m, K) - labels one-hot
4     cache : dict des activations du forward pass
5     parameters : dict des poids W[l], b[l]
6     """
7     m = y_true.shape[0]
8     grads = {}
9     L = len(parameters) // 2
10
11     # Output layer gradient (cross-entropy + softmax)
12     y_pred = cache[f'A{L}']
13     dZ = y_pred - y_true # Shape: (m, K)
14
15     # Backprop à travers les couches
16     for l in reversed(range(1, L + 1)):
17         A_prev = cache[f'A{l-1}']
18
19         # Gradients des paramètres
20         grads[f'dW{l}'] = (1/m) * np.dot(dZ.T, A_prev)
21         grads[f'db{l}'] = (1/m) * np.sum(dZ, axis=0, keepdims=True)
22
23         if l > 1:
24             # Gradient pour la couche précédente
25             W = parameters[f'W{l}']
26             dA_prev = np.dot(dZ, W)
27
28             # Gradient à 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

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

8.1.2 Stochastic Gradient Descent (SGD)

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

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

Avec momentum :

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

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

Typiquement $\beta = 0.9$.

8.1.3 Adam (Adaptive Moment Estimation)

Combine momentum et RMSprop :

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

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

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

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

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

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

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 $\{d_0, d_1, \dots, d_L\}$

Require : Learning rate α , batch size B , nombre d'epochs T

Ensure : Paramètres entraînés θ^*

```

1 : Initialiser aléatoirement  $\mathbf{W}^{[l]}, \mathbf{b}^{[l]}$  pour  $l = 1, \dots, L$ 
2 : for  $epoch = 1$  to  $T$  do
3 :   Mélanger les données
4 :   for chaque mini-batch  $(X_{batch}, y_{batch})$  do
5 :     {Forward pass}
6 :      $\hat{y}_{batch}, cache = \text{forward\_pass}(X_{batch}, \theta)$ 
7 :      $L_{batch} = \text{compute\_loss}(\hat{y}_{batch}, y_{batch})$ 
8 :     {Backward pass}
9 :      $grads = \text{backward\_pass}(y_{batch}, cache, \theta)$ 
10 :    {Update parameters}
11 :    for chaque paramètre  $\theta$  do
12 :       $\theta \leftarrow \theta - \alpha \cdot \frac{\partial L}{\partial \theta}$ 
13 :    end for
14 :  end for
15 :  Évaluer sur validation set
16 : end for
17 : return  $\theta^*$ 

```

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) :

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

He initialization (pour ReLU) :

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

```

1 def initialize_parameters(layer_dims):
2     """
3     layer_dims : [d0, d1, d2, ..., dL]
4     """
5     parameters = {}
6     L = len(layer_dims) - 1
7
8     for l in range(1, L + 1):

```

```

9      # He initialization
10     parameters[f'W{1}'] = np.random.randn(
11         layer_dims[1], layer_dims[1-1]
12     ) * np.sqrt(2 / layer_dims[1-1])
13
14     parameters[f'b{1}'] = np.zeros((1, layer_dims[1]))
15
16     return parameters

```

Listing 4 – Initialisation He

9 Régularisation

9.1 L2 Regularization (Weight Decay)

Ajouter un terme de pénalité sur les poids :

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

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

Le gradient devient :

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

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) :

```

1 def dropout_forward(A, keep_prob=0.5, training=True):
2     if training:
3         mask = np.random.rand(*A.shape) < keep_prob
4         A = A * mask / keep_prob # Inverted dropout
5         return A, mask
6     else:
7         return A, None

```

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 :

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

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

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

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

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.

```

1 best_val_loss = float('inf')
2 patience = 10
3 counter = 0
4
5 for epoch in range(n_epochs):
6     train_model()
7     val_loss = evaluate_validation()
8
9     if val_loss < best_val_loss:
10         best_val_loss = val_loss
11         save_checkpoint()
12         counter = 0
13     else:
14         counter += 1
15         if counter >= patience:
16             print("Early stopping")
17             break

```

10 Implémentation Complète avec NumPy

```

1 import numpy as np
2
3 class MLP:
4     def __init__(self, layer_dims, learning_rate=0.01):
5         self.layer_dims = layer_dims

```

```

6         self.lr = learning_rate
7         self.parameters = self._initialize_parameters()
8         self.L = len(layer_dims) - 1
9
10        def _initialize_parameters(self):
11            """He initialization"""
12            params = {}
13            for l in range(1, len(self.layer_dims)):
14                params[f'W{l}'] = np.random.randn(
15                    self.layer_dims[l], self.layer_dims[l-1]
16                ) * np.sqrt(2 / self.layer_dims[l-1])
17                params[f'b{l}'] = np.zeros((1, self.layer_dims[l]))
18            return params
19
20        def _relu(self, Z):
21            return np.maximum(0, Z)
22
23        def _softmax(self, Z):
24            expZ = np.exp(Z - np.max(Z, axis=1, keepdims=True))
25            return expZ / np.sum(expZ, axis=1, keepdims=True)
26
27        def forward(self, X):
28            """Forward pass"""
29            cache = {'A0': X}
30            A = X
31
32            for l in range(1, self.L + 1):
33                Z = np.dot(A, self.parameters[f'W{l}'].T) + \
34                    self.parameters[f'b{l}']
35
36                if l < self.L:
37                    A = self._relu(Z)
38                else:
39                    A = self._softmax(Z)
40
41                cache[f'Z{l}'] = Z
42                cache[f'A{l}'] = A
43
44            return A, cache
45
46        def backward(self, y_true, cache):
47            """Backpropagation"""
48            m = y_true.shape[0]
49            grads = {}
50
51            # Output layer
52            dZ = cache[f'A{self.L}'] - y_true
53
54            for l in reversed(range(1, self.L + 1)):
55                A_prev = cache[f'A{l-1}']
56                grads[f'dW{l}'] = (1/m) * np.dot(dZ.T, A_prev)

```

```

57         grads[f'db{1}'] = (1/m) * np.sum(dZ, axis=0, keepdims=True)
58
59         if l > 1:
60             W = self.parameters[f'W{1}']
61             dA_prev = np.dot(dZ, W)
62             dZ = dA_prev * (cache[f'Z{1-1}'] > 0)
63
64         return grads
65
66     def update_parameters(self, grads):
67         """Gradient descent update"""
68         for l in range(1, self.L + 1):
69             self.parameters[f'W{1}'] -= self.lr * grads[f'dW{1}']
70             self.parameters[f'b{1}'] -= self.lr * grads[f'db{1}']
71
72     def compute_loss(self, y_pred, y_true):
73         """Cross-entropy loss"""
74         m = y_true.shape[0]
75         loss = -np.sum(y_true * np.log(y_pred + 1e-8)) / m
76         return loss
77
78     def fit(self, X, y, epochs=100, batch_size=32, verbose=True):
79         """Training loop"""
80         m = X.shape[0]
81         history = {'loss': []}
82
83         for epoch in range(epochs):
84             # Shuffle
85             indices = np.random.permutation(m)
86             X_shuffled = X[indices]
87             y_shuffled = y[indices]
88
89             epoch_loss = 0
90             n_batches = m // batch_size
91
92             for i in range(n_batches):
93                 start = i * batch_size
94                 end = start + batch_size
95                 X_batch = X_shuffled[start:end]
96                 y_batch = y_shuffled[start:end]
97
98                 # Forward
99                 y_pred, cache = self.forward(X_batch)
100                 loss = self.compute_loss(y_pred, y_batch)
101                 epoch_loss += loss
102
103                 # Backward
104                 grads = self.backward(y_batch, cache)
105
106                 # Update
107                 self.update_parameters(grads)

```

```

108         avg_loss = epoch_loss / n_batches
109         history['loss'].append(avg_loss)
110
111
112         if verbose and (epoch + 1) % 10 == 0:
113             print(f"Epoch {epoch+1}/{epochs}, Loss: {avg_loss:.4f}")
114
115         return history
116
117     def predict(self, X):
118         """Predictions"""
119         y_pred, _ = self.forward(X)
120         return np.argmax(y_pred, axis=1)
121
122 # Exemple d'utilisation
123 if __name__ == "__main__":
124     from sklearn.datasets import load_digits
125     from sklearn.model_selection import train_test_split
126     from sklearn.preprocessing import OneHotEncoder
127
128     # Charger données
129     digits = load_digits()
130     X, y = digits.data, digits.target
131
132     # Normaliser
133     X = X / 16.0
134
135     # One-hot encoding
136     y_onehot = np.zeros((len(y), 10))
137     y_onehot[np.arange(len(y)), y] = 1
138
139     # Split
140     X_train, X_test, y_train, y_test = train_test_split(
141         X, y_onehot, test_size=0.2, random_state=42
142     )
143
144     # Entraîner
145     mlp = MLP(layer_dims=[64, 128, 64, 10], learning_rate=0.1)
146     history = mlp.fit(X_train, y_train, epochs=100, batch_size=32)
147
148     # Évaluer
149     y_pred = mlp.predict(X_test)
150     y_test_labels = np.argmax(y_test, axis=1)
151     accuracy = np.mean(y_pred == y_test_labels)
152     print(f"\nTest Accuracy: {accuracy:.4f}")

```

Listing 5 – MLP complet from scratch

11 Implémentation avec PyTorch

```
1 import torch
2 import torch.nn as nn
3 import torch.optim as optim
4 from torch.utils.data import DataLoader, TensorDataset
5
6 class MLP_PyTorch(nn.Module):
7     def __init__(self, input_dim, hidden_dims, output_dim):
8         super(MLP_PyTorch, self).__init__()
9
10        layers = []
11        prev_dim = input_dim
12
13        # Hidden layers
14        for hidden_dim in hidden_dims:
15            layers.append(nn.Linear(prev_dim, hidden_dim))
16            layers.append(nn.ReLU())
17            layers.append(nn.Dropout(0.2))
18            prev_dim = hidden_dim
19
20        # Output layer
21        layers.append(nn.Linear(prev_dim, output_dim))
22
23        self.network = nn.Sequential(*layers)
24
25        def forward(self, x):
26            return self.network(x)
27
28 # Exemple d'utilisation
29 model = MLP_PyTorch(input_dim=64, hidden_dims=[128, 64], output_dim=10)
30
31 # Loss et optimizer
32 criterion = nn.CrossEntropyLoss()
33 optimizer = optim.Adam(model.parameters(), lr=0.001)
34
35 # Training loop
36 def train_pytorch(model, train_loader, epochs=50):
37     model.train()
38     for epoch in range(epochs):
39         total_loss = 0
40         for X_batch, y_batch in train_loader:
41             # Forward
42             outputs = model(X_batch)
43             loss = criterion(outputs, y_batch)
44
45             # Backward
46             optimizer.zero_grad()
47             loss.backward()
48             optimizer.step()
49
50         total_loss += loss.item()
51
```

```

52     if (epoch + 1) % 10 == 0:
53         print(f"Epoch {epoch+1}, Loss: {total_loss/len(train_loader)
54             :.4f}")
55 # Utilisation
56 X_train_tensor = torch.FloatTensor(X_train)
57 y_train_tensor = torch.LongTensor(np.argmax(y_train, axis=1))
58
59 train_dataset = TensorDataset(X_train_tensor, y_train_tensor)
60 train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
61
62 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 :

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

```

1 def gradient_check(model, X, y, epsilon=1e-7):
2     """Vérifie backprop avec gradients numériques"""
3     # Calculer gradients analytiques (backprop)
4     y_pred, cache = model.forward(X)

```

```

5     grads_analytic = model.backward(y, cache)
6
7     # Calculer gradients numériques
8     for param_name in model.parameters:
9         param = model.parameters[param_name]
10        grad_numeric = np.zeros_like(param)
11
12        it = np.nditer(param, flags=['multi_index'])
13        while not it.finished:
14            idx = it.multi_index
15            old_value = param[idx]
16
17            # f(theta + epsilon)
18            param[idx] = old_value + epsilon
19            y_pred_plus, _ = model.forward(X)
20            loss_plus = model.compute_loss(y_pred_plus, y)
21
22            # f(theta - epsilon)
23            param[idx] = old_value - epsilon
24            y_pred_minus, _ = model.forward(X)
25            loss_minus = model.compute_loss(y_pred_minus, y)
26
27            # Gradient numérique
28            grad_numeric[idx] = (loss_plus - loss_minus) / (2 * epsilon)
29
30            param[idx] = old_value
31            it.iternext()
32
33        # Comparer
34        grad_analytic = grads_analytic[f'd{param_name}']
35        diff = np.linalg.norm(grad_numeric - grad_analytic) / \
36            (np.linalg.norm(grad_numeric) + np.linalg.norm(
grad_analytic))
37
38        print(f"{param_name}: diff = {diff:.2e}")
39        if diff < 1e-7:
40            print("    Gradient correct")
41        else:
42            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 ?

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

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 |

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)

```
1 from sklearn.model_selection import RandomizedSearchCV
2 from sklearn.neural_network import MLPClassifier
3
4 param_distributions = {
5     'hidden_layer_sizes': [(64,), (128,), (64, 32), (128, 64)],
6     'activation': ['relu', 'tanh'],
7     'alpha': [0.0001, 0.001, 0.01], # L2 regularization
8     'learning_rate_init': [0.001, 0.01, 0.1],
9     'batch_size': [32, 64, 128]
10 }
11
12 mlp = MLPClassifier(max_iter=100)
13 random_search = RandomizedSearchCV(
14     mlp, param_distributions, n_iter=20, cv=3, n_jobs=-1
15 )
16
17 random_search.fit(X_train, y_train)
18 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=aircArvnKk>
- 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*.