Broadcasting Avancé & Einstein Summation

Votre Nom

28 novembre 2025

Résumé

Ce document explore deux concepts fondamentaux en calcul numérique avec NumPy : le **Broadcasting Avancé** et la **Einstein Summation**. Ces techniques permettent d'effectuer des opérations complexes sur des tableaux multidimensionnels de manière efficace et élégante.

Table des matières

1 Introduction

1.1 Qu'est-ce que le Broadcasting ?

Le **broadcasting** est un mécanisme puissant de NumPy qui permet d'effectuer des opérations arithmétiques sur des tableaux de formes différentes. Il étend automatiquement les dimensions des tableaux plus petits pour qu'ils correspondent aux dimensions des tableaux plus grands.

1.2 Qu'est-ce que l'Einstein Summation ?

La **notation d'Einstein** est une convention mathématique qui simplifie l'écriture des expressions impliquant des sommes sur des indices répétés. NumPy implémente cette notation via la fonction `np.einsum`.

2 Broadcasting Avancé

2.1 Règles Fondamentales

Le broadcasting suit deux règles principales :

1. **Alignement à droite** : Les dimensions sont alignées à partir de la droite
2. **Dimensions unités** : Les dimensions de taille 1 sont étendues

2.2 Exemples de Base

```python
import numpy as np

# Exemple 1: Broadcasting simple
A = np.array([1, 2, 3])         # Shape: (3,)
B = np.array([[1], [2], [3]])   # Shape: (3, 1)

result = A + B
print(result)
# Output:
# [[2 3 4]
#  [3 4 5]
#  [4 5 6]]
```

2.3 Applications Avancées

2.3.1 Normalisation de Données

```python
# Normalisation d'un dataset de temp ratures
temperatures = np.random.randint(-10, 35, (5, 12))  # 5 villes, 12 mois

# Calcul des statistiques avec broadcasting
mean_by_city = temperatures.mean(axis=1, keepdims=True)
std_by_city = temperatures.std(axis=1, keepdims=True)

# Normalisation
normalized = (temperatures - mean_by_city) / std_by_city
```

2.3.2 Calcul de Distances

```python
def distance_matrix_broadcasting(points1, points2):
    """
    Calcule la matrice des distances entre points
    """
    squared_diff = np.sum(
        (points1[:, np.newaxis, :] - points2[np.newaxis, :, :]) ** 2,
        axis=2
    )
    return np.sqrt(squared_diff)

# Utilisation
points_A = np.array([[0, 0], [1, 1], [2, 2]])
points_B = np.array([[1, 0], [0, 1]])
distances = distance_matrix_broadcasting(points_A, points_B)
```

3 Einstein Summation

3.1 Syntaxe de Base

La syntaxe `np.einsum` suit le format :

```
np.einsum('indices_input -> indices_output', array1, array2, ...)
```

3.2 Opérations Élémentaires

```python
# Transposition
A = np.array([[1, 2], [3, 4]])
transposed = np.einsum('ij->ji', A)

# Somme sur un axe
sum_rows = np.einsum('ij->j', A)  # Somme sur les lignes
sum_cols = np.einsum('ij->i', A)  # Somme sur les colonnes
```

3.3 Applications Avancées

3.3.1 Produit Matriciel

```
A = np.random.rand(3, 4)
B = np.random.rand(4, 5)

# Produit matriciel classique
classic = np.dot(A, B)

# Avec einsum
einsum_result = np.einsum('ik,kj->ij', A, B)

print("Identique:", np.allclose(classic, einsum_result))
```

3.3.2 Similarité Cosinus

```
def cosine_similarity_einsum(vectors):
    norms = np.einsum('ij,ij->i', vectors, vectors)
    normalized = vectors / np.sqrt(norms[:, np.newaxis])
    similarity = np.einsum('ik,jk->ij', normalized, normalized)
    return similarity

vectors = np.random.rand(4, 3)
similarity_matrix = cosine_similarity_einsum(vectors)
```

3.3.3 Contractions Tensorielles

```
def tensor_contractions():
    # Tenseur 3D
    batch_images = np.random.rand(2, 3, 4)
    filters = np.random.rand(3, 3, 4)

    # Contraction complexe
    result = np.einsum('bij,hwij->bhw', batch_images, filters)
    return result
```

4 Combinaison des Deux Techniques

4.1 Exemple Intégré

```
def combined_approach():
    # Donn es: capteurs x mesures x features
    sensor_data = np.random.rand(3, 100, 5)
    weights = np.random.rand(3, 5)

```

```
    # M thode combin e
    result = np.einsum('ijk,ik->ij', sensor_data, weights)
    return result
```

4.2 Avantages de la Combinaison

— **Performance** : Évite les boucles Python
— **Lisibilité** : Code plus concis et expressif
— **Mémoire** : Réduction de l'utilisation mémoire

5 Challenge Pratique

5.1 Énoncé du Challenge

Implémenter une fonction qui calcule la matrice de covariance entre les features d'un dataset en utilisant uniquement du broadcasting et `np.einsum`, sans utiliser `np.cov`.

6 Performance et Bonnes Pratiques

6.1 Comparaison de Performance

Méthode	Temps d'exécution	Utilisation mémoire
Boucles Python	100%	100%
Broadcasting seul	20%	150%
Einsum seul	15%	100%
Combinaison	10%	110%

TABLE 1 – Comparaison des performances (valeurs relatives)

6.2 Recommandations

1. Utiliser le broadcasting pour les opérations simples
2. Préférer einsum pour les contractions complexes
3. Toujours vérifier les shapes intermédiaires
4. Profiler le code pour les applications critiques

7 Conclusion

Le broadcasting avancé et l'Einstein summation sont des outils puissants pour le calcul scientifique avec NumPy. Leur maîtrise permet d'écrire du code à la fois performant et lisible, en évitant les boucles Python coûteuses.

Points Clés à Retenir

- Le broadcasting étend automatiquement les dimensions
- Einsum permet des contractions tensorielles complexes
- La combinaison des deux offre le meilleur des deux mondes
- Toujours privilégier la lisibilité et la maintenabilité

8 Ressources Complémentaires

- Documentation NumPy : `https://numpy.org/doc/`
- Guide du broadcasting : `https://numpy.org/doc/stable/user/basics.broadcasting.html`
- Tutoriel einsum : `https://numpy.org/doc/stable/reference/generated/numpy.einsum.html`