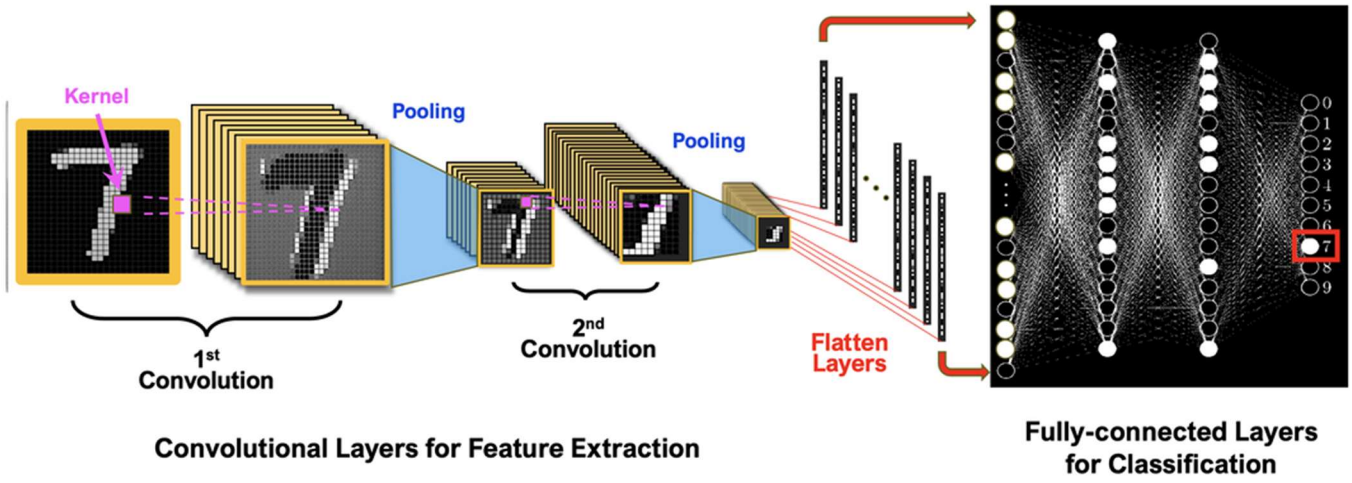
جامعة سيدي محمد بن عبد الله بفاس
UNIVERSITE SIDI MOHAMED BEN ABDELLAH DE FES
كلية العلوم ظهر المهراز فاس
FACULTE DES SCIENCES DHAR EL MAHRAZ FES

BDSaS

Master Big Data Analytics & Smart Systems

CNN From Scrach



Réaliser Par :

- **BENELFQIH AYA**
- **ERRAZOUKI AYA**
- **OULAJA SAFAE**

Encadré Par:

- **Pr. RIFFI Jamal**

Année universitaire 2024/2025

Sommaire :

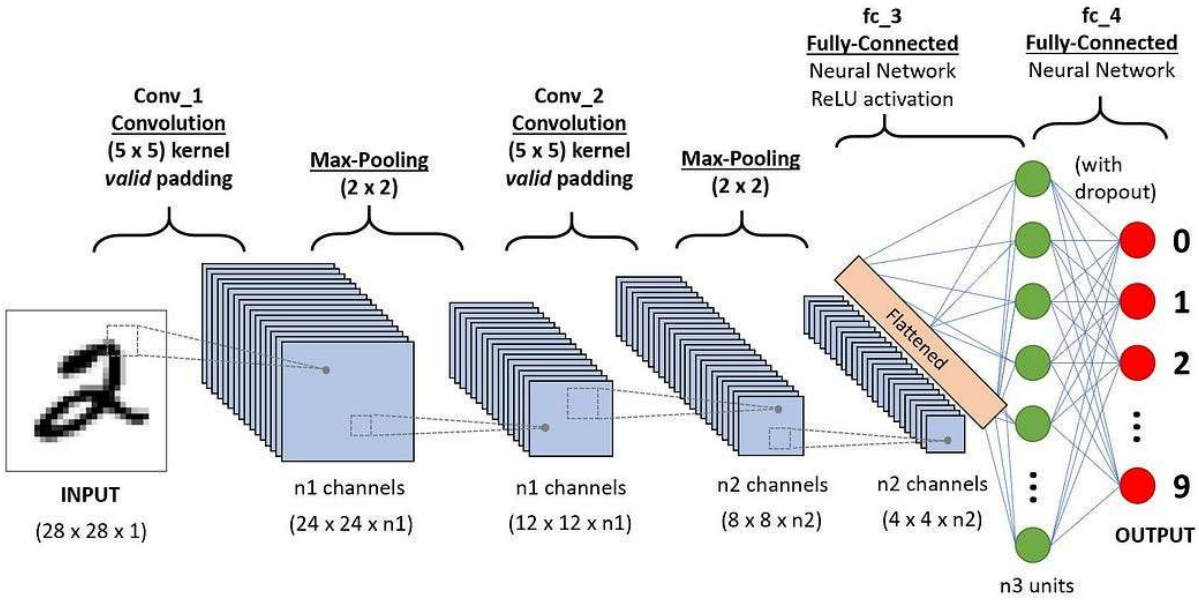
I. Introduction :

Un réseau neuronal convolutif (CNN, pour Convolutional Neural Network) est un type de réseau de neurones profond largement utilisé dans le domaine de l'apprentissage profond pour des tâches telles que la reconnaissance d'images, la vision par ordinateur, la reconnaissance vocale, et plus encore. Il est particulièrement efficace pour le traitement de données structurées sous forme de grilles, comme les images.

Il est conçu pour traiter des données visuelles sous forme de matrices de pixels, telles que des images ou des vidéos. Contrairement aux réseaux de neurones classiques (fully connected), un CNN utilise des opérations de convolution pour extraire des caractéristiques locales importantes de l'image.

II. Architecture d'un CNN :

L'architecture de base d'un CNN est composée de plusieurs couches qui jouent des rôles spécifiques.



Calcul de la taille de sortie de convolution

L'entrée d'un CNN est généralement une image de taille H×W×C, où H est la hauteur, W est la largeur et c'est le nombre de canaux (par exemple, 3 pour une image RGB).
La formule générale pour calculer la taille de sortie de convolution est la suivante :

Taille de sortie = ((H - F + 2P) / S) + 1 , ((W - F + 2P) / S) + 1

où :

- H = hauteur de l'image d'entrée,
- W = largeur de l'image d'entrée,
- F = taille du filtre (noyau),
- P = padding (quantité de pixels ajoutés autour de l'image),
- S = stride (pas de déplacement du filtre).

Soit une image d'entrée de taille 28x28x1 (image en niveaux de gris) de l'architecture, un filtre de taille 5x5, un padding de 0 et un stride de 1.

Taille de sortie (hauteur) = (28 - 5 + 2(0)) / 1 + 1 = 23 / 1 + 1 = 24

Taille de sortie (largeur) = (28 - 5 + 2(0)) / 1 + 1 = 23 / 1 + 1 = 24

Donc, la taille de la sortie de convolution sera de taille **24x24.**

Pour mieux comprendre le fonctionnement du CNN, nous illustrons ci-dessous les étapes du passage Forward et Backward :

1. **Forward Propagation**

La propagation avant est le processus où l'image d'entrée traverse les différentes couches du réseau, et une prédiction est faite :

- **Entrée** : L'image (ex. 28x28 pixels pour MNIST) est donnée au réseau.
- **Convolution** : Des filtres extraits des caractéristiques locales de l'image (comme les bords).
- **Activation (ReLU)** : Applique une fonction ReLU pour ajouter de la non-linéarité.
- **Pooling** : Réduit la taille des cartes de caractéristiques, tout en conservant les informations essentielles.
- **Couches entièrement connectées** : Combine les caractéristiques extraites pour la classification.
- **Softmax** : Convertit les valeurs de sortie en probabilités de classification.
- **Prédiction** : La classe avec la probabilité la plus élevée est choisie comme prédiction.

2. **Backward Propagation**

La rétropropagation (Backward Propagation) est le processus par lequel le réseau ajuste ses poids pour minimiser l'erreur entre la prédiction.

- **Calcul de la perte** : Compare la prédiction aux valeurs réelles pour calculer l'erreur (fonction de perte, souvent Cross-Entropy).
- **Rétropropagation des gradients** : Le gradient de l'erreur est calculé à partir de la sortie et propagé vers les couches précédentes.
- **Mise à jour des poids** : Les poids sont ajustés à l'aide de la descente de gradient pour réduire l'erreur.

III. Forward Propagation :

1) Couche de Convolution :

Cette couche applique un filtre sur l'image d'entrée. Le filtre glisse sur l'image pour produire une carte de caractéristiques. Cela permet d'extraire des motifs locaux tels que des bords, des textures ou des formes simples.

Formule de convolution : $F = X * K$

- X : Matrice de l'entrée (image ou carte de caractéristiques précédente),
- K : Noyau de convolution
- $*$: Opération de convolution.

Exemple :

1	0	3
2	5	1
4	0	2

X

*

1	0
2	-1

f1

-1	2
0	1

f2

Soit une image d'entrée de taille 3x3x1, un filtre de taille 2x2, un padding de 0 et un stride de 1.

Taille de sortie (hauteur) = (3 - 2 + 2(0)) / 1 + 1 = 1/ 1 + 1 = 2

Taille de sortie (largeur) = (3- 2+ 2(0)) / 1 + 1 = 1 / 1 + 1 = 2

Donc, la taille de la sortie de convolution sera de taille **2x2.**

On va appliquer la convolution entre l'entrée X et les deux filtre f1 et f2 :

$O_1 =$

0	9
10	3

$O_2 =$

4	7
8	-1

2) Fonction d'activation (ReLU) :

Après chaque opération de convolution, une fonction d'activation est généralement appliquée pour introduire de la non-linéarité dans le modèle. Fonctions d'activation comme ReLU (Rectified Linear Unit) introduisent de la non-linéarité, ce qui permet au réseau d'apprendre des modèles plus complexes.

Formule de ReLU : ***ReLU(F)=max(0,F)***

$Relu(O_1) =$

0	9
10	3

$Relu(O_2) =$

4	7
8	0

3) Couche de Pooling :

Le pooling est une opération qui permet de réduire la taille de la carte de caractéristiques tout en conservant les informations les plus importantes. L'opération la plus courante est le max pooling, qui sélectionne la valeur maximale d'une fenêtre glissante .

Formule du pooling (max pooling) : *P' = max_pool(P)*

max pooling 1 = [10] max pooling 2 = [8]

4) Flattening :

Après plusieurs couches de convolution et de pooling, la sortie est généralement aplatie (flatten) en un vecteur unidimensionnel pour être envoyée à une couche entièrement connectée (fully connected layer). Cela transforme une matrice 2D en un vecteur 1D.

Formule du flattening : ***x=Flatten(P')***

$x = \begin{bmatrix} 10 \\ 8 \end{bmatrix}$

5) Full Connected :

Dans une **couche entièrement connectée**, chaque neurone est connecté à tous les neurones de la couche précédente. La sortie de la couche précédente (le vecteur aplati) est multipliée par une matrice de poids W_f et un biais b_f est ajouté.

Formule de la couche entièrement connectée : $z=W_f x+b_f$

- W_f est la matrice de poids
- x est le vecteur aplati provenant de la couche précédente,
- b_f est le vecteur de biais,
- z est le vecteur de sortie avant activation.

Initialisation des poids et des biais :

$W_1 = \begin{bmatrix} 0.5 \\ -0.3 \end{bmatrix}$ $W2 = \begin{bmatrix} -0.2 \\ 0.7 \end{bmatrix}$ $b = \begin{bmatrix} 1.2 \\ -0.5 \end{bmatrix}$

$Z_1 = (10 \times 0.5) + (8 \times 0.5) + 1.2 = 3.8$

$Z_2 = (10 \times -0.2) + (8 \times 0.7) - 1.2 = 3.1$

6) Fonction d' activation Softmax :

La fonction **Softmax** est appliquée à la sortie de la couche entièrement connectée pour obtenir des probabilités de classe. La somme des probabilités est égale à 1.

Formule de Softmax : $\text{Softmax}(Z_i)=\frac{e^{Zi}}{\sum e^{Zi}}$

$\hat{y}_1 = \frac{e^{3.8}}{e^{3.8}+e^{3.1}} = 0.67$

$\hat{y}_2 = \frac{e^{3.1}}{e^{3.8}+e^{3.1}} = 0.33$

Donc :

$Z = \begin{bmatrix} 3.8 \\ 3.1 \end{bmatrix}$ $\hat{y} = \begin{bmatrix} 0.67 \\ 0.33 \end{bmatrix}$

On a choisi comme valeur réelle : $y = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

IV. Backward Propagation :

1) Calcul de Loss :

On commence par le calcul de l'erreur entre les prédictions du modèle et les valeurs réelles. Une fonction de perte commune pour les tâches de classification est la perte d'entropie croisée.

Formule de la perte (cross-entropy loss) :

$$L = -\sum_{i=1}^{c} yi \log(\hat{y}\, i)$$

L = - (0 × log(0.67) + 1 × log(0.33)) = **0.48**

2) Calcul du Gradient :

Une fois que la perte est calculée, la prochaine étape est de propager cette erreur dans le réseau. Le gradient de la perte par rapport aux sorties du réseau est calculé en appliquant la règle de la chaîne.

Gradient de la perte par rapport aux logits:

$$\frac{\partial L}{\partial Zi} = \hat{yi} - yi$$

$\frac{\partial L}{\partial Z1} = 0.67 - 0 = 0.67$

$\frac{\partial L}{\partial Z} = 0.33 - 1 = -0.67$

Gradient de la perte par rapport aux poids et biais de la couche FC :

- **Gradient par rapport aux poids :**

$$\frac{\partial L}{\partial Wf} = \frac{\partial L}{\partial Zi}\frac{\partial Zi}{\partial Wf} \quad \text{avec } \frac{\partial Zi}{\partial Wf} = x_i$$

$\frac{\partial L}{\partial W1} = 0.67 \begin{bmatrix}10\\8\end{bmatrix} = \begin{bmatrix}6.7\\5.36\end{bmatrix}$

$\frac{\partial L}{\partial W2} = -0.67 \begin{bmatrix}10\\8\end{bmatrix} = \begin{bmatrix}-6.7\\-5.36\end{bmatrix}$

- **Gradient par rapport aux biais :**

$$\frac{\partial L}{\partial bf} = \frac{\partial L}{\partial Zi}\frac{\partial Zi}{\partial bf} \quad \text{avec } \frac{\partial Zi}{\partial bf} = 1$$

$\frac{\partial L}{\partial b1} = 0.67$

$\frac{\partial L}{\partial b2}$ = -0.67

3) Backward propagation à travers la Couche de Pooling

Gradient de la perte par rapport à la carte de caractéristiques activée (ReLU) :

$$\frac{\partial L}{\partial xi} = \frac{\partial L}{\partial Zi} \frac{\partial Zi}{\partial xi} \text{ avec } \frac{\partial Zi}{\partial xi} = w_i$$

$$\frac{\partial L}{\partial xi} = \frac{\partial L}{\partial Zi} w1 + \frac{\partial L}{\partial Zi} w2$$

$$\frac{\partial L}{\partial x} = (0.67 \times 0.5) + (-0.67 \times -0.2) = 0.469$$

$$\frac{\partial L}{\partial x} = (0.67 \times -0.3) + (-0.67 \times 0.7) = -0.67$$

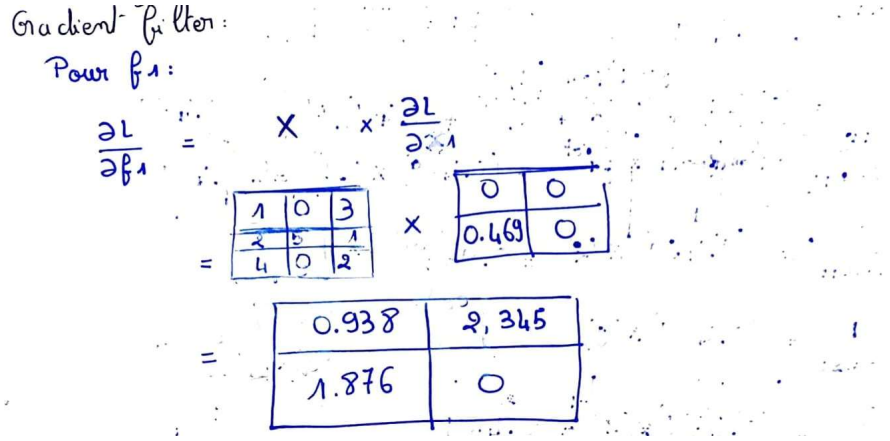
$$\frac{\partial L}{\partial x} = \begin{bmatrix} 0.469 \\ -0.67 \end{bmatrix}$$

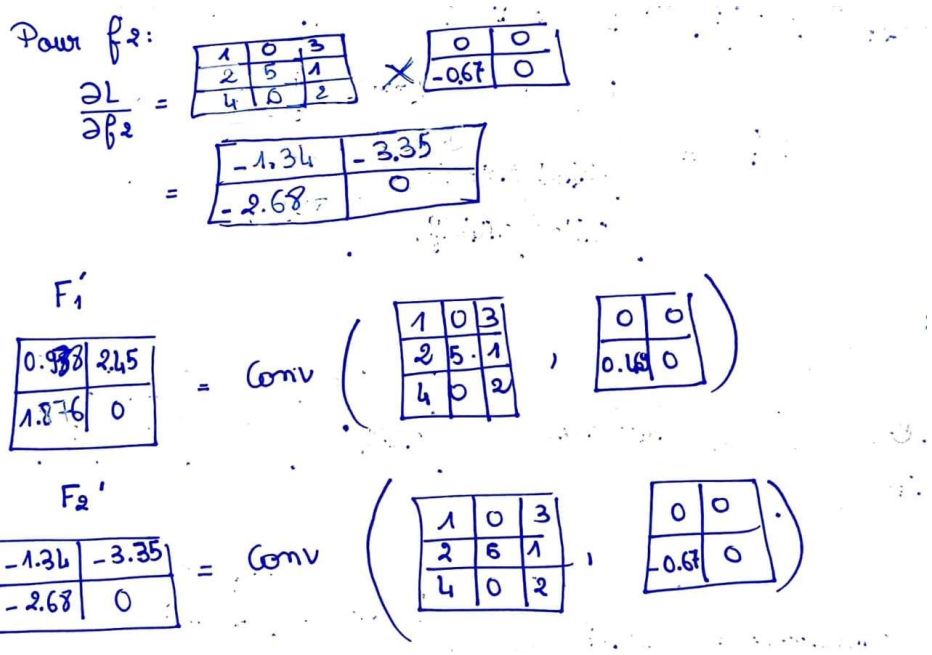
dmax pooling 1 = $\begin{matrix} 0 & 0 \\ 0.469 & 0 \end{matrix}$

dmax pooling 2 = $\begin{matrix} 0 & 0 \\ -0.67 & 0 \end{matrix}$

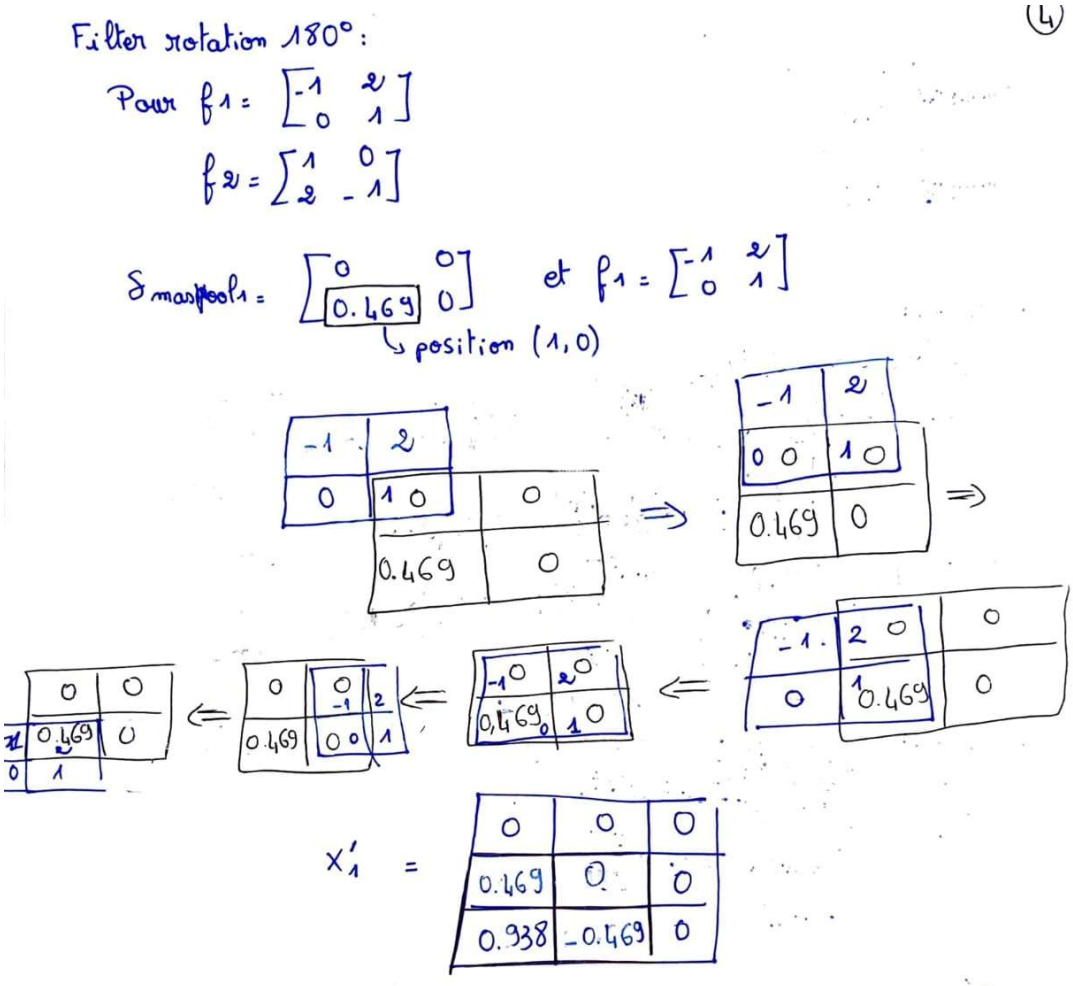
4) Backward propagation à travers la Convolution

Gradient de la perte par rapport au noyau de convolution (filtre) :





Gradient de la perte par rapport à l'entrée de la couche précédente (image) :



X'_1

0	0	0
+0.469	0	0
0.938	0.469	0

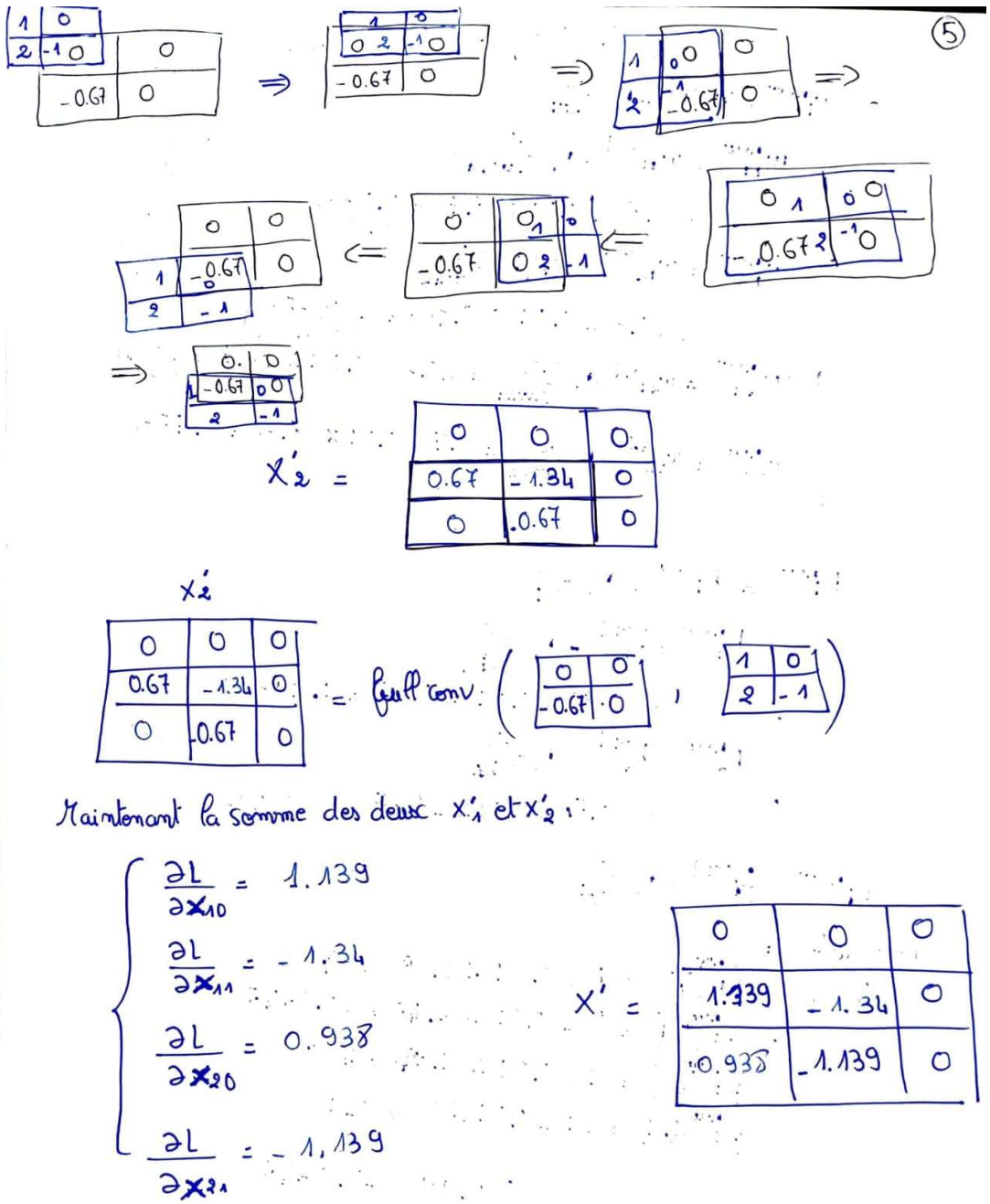
= full conv (| 0 | 0 |
|---|---|
| 0.469 | 0 |

, | -1 | 2 |
|---|---|
| 0 | 1 |

)

$\delta_{maxpool2}$ = | 0 | 0 |
|---|---|
| -0.67 | 0 |

(position (1, 0)) et f_2 = | 1 | 0 |
|---|---|
| 2 | -1 |



X'_2 =

0	0	0
0.67	-1.34	0
0	-0.67	0

X'_2

0	0	0
0.67	-1.34	0
0	-0.67	0

= full conv (| 0 | 0 |
|---|---|
| -0.67 | 0 |

, | 1 | 0 |
|---|---|
| 2 | -1 |

)

Maintenant la somme des deux X'_1 et X'_2 :

$$\begin{cases} \frac{\partial L}{\partial X_{10}} = 1.139 \\ \frac{\partial L}{\partial X_{11}} = -1.34 \\ \frac{\partial L}{\partial X_{20}} = 0.938 \\ \frac{\partial L}{\partial X_{21}} = -1.139 \end{cases}$$

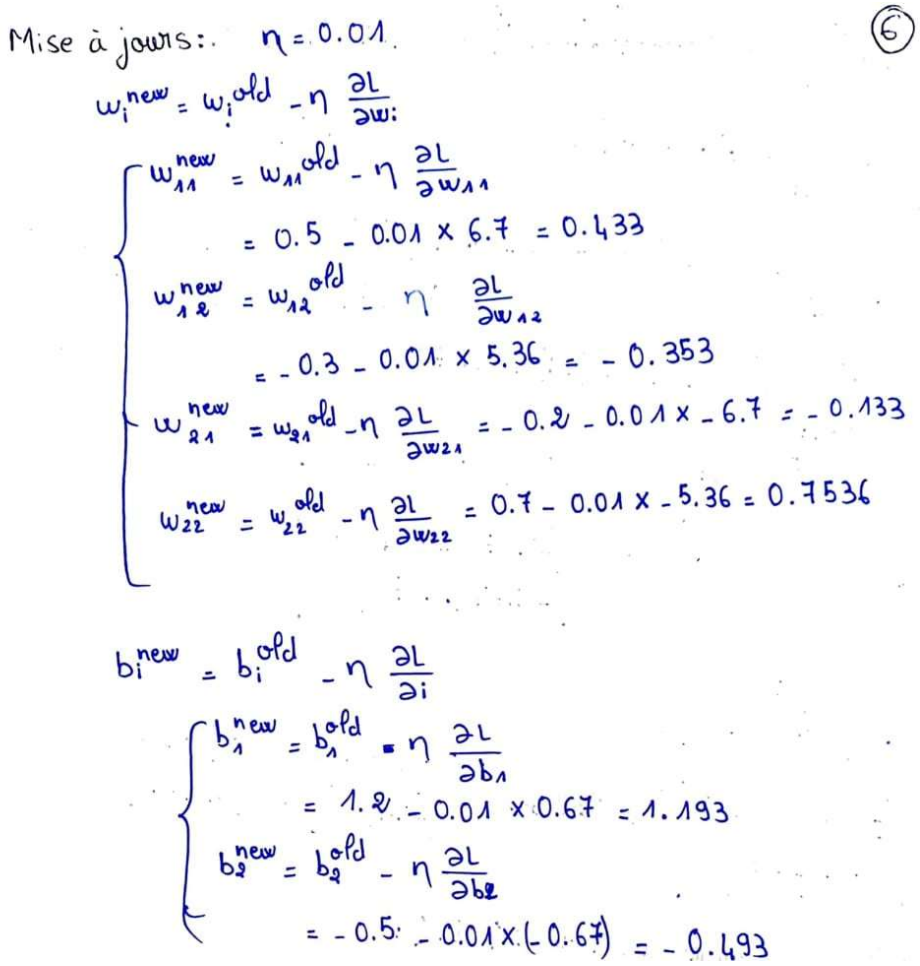
X' =

0	0	0
1.139	-1.34	0
0.938	-1.139	0

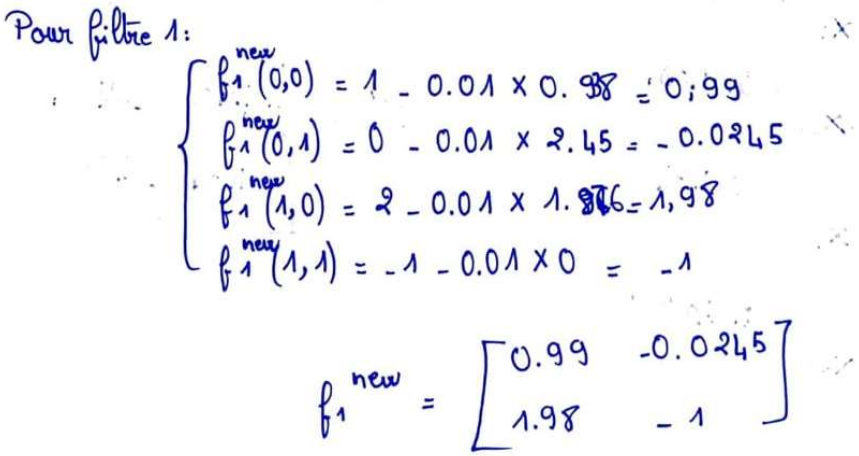
5) Mise à Jour des Poids (Gradient Descent)

Les gradients calculés sont ensuite utilisés pour mettre à jour les poids et les biais avec l'algorithme du gradient descent (descente de gradient) :

- **Mise à jour des poids de la couche entièrement connectée :**



- **Mise à jour des filtres de convolution :**



Pour filtre 2 :

$$\begin{cases} f_2^{new}(0,0) = -1 - 0.01 \times (-1.34) = -0.99 \\ f_2^{new}(0,1) = 2 - 0.01 \times (-3.35) = 2.033 \\ f_2^{new}(1,0) = 0 - 0.01 \times (-2.68) = 0.0268 \\ f_2^{new}(1,1) = 1 - 0.01 \times (0) = 1 \end{cases}$$

$$f_2^{new} = \begin{bmatrix} -0.99 & 2.033 \\ 0.0268 & 1 \end{bmatrix}$$

V. Implementation du code from scrach de CNN :

1) Ajout de Padding

La fonction add_padding ajoute des zéros autour de l'image pour conserver la taille des cartes de caractéristiques après la convolution, ou pour assurer que les dimensions de l'image sont multiples de la taille du noyau.

```
def add_padding(input_matrix, padding):
    if padding > 0:
        padded_matrix = np.pad(input_matrix, pad_width=padding, mode='constant', constant_values=0)
        return padded_matrix
    else:
        return input_matrix
```

2) Convolution

La fonction convolution applique un filtre à l'image d'entrée, en déplaçant le noyau sur l'image et en effectuant un produit scalaire entre le noyau et matrice correspondante de l'image.

```
def convolution(input_matrix, kernel, stride=1, padding=0):
    input_matrix = add_padding(input_matrix, padding)
    input_height, input_width = input_matrix.shape
    kernel_height, kernel_width = kernel.shape

    output_height = int((input_height - kernel_height) / stride) + 1
    output_width = int((input_width - kernel_width) / stride) + 1

    feature_map = np.zeros((output_height, output_width))

    for i in range(0, output_height):
        for j in range(0, output_width):
            start_i = i * stride
            start_j = j * stride
            end_i = start_i + kernel_height
            end_j = start_j + kernel_width

            feature_map[i, j] = np.sum(input_matrix[start_i:end_i, start_j:end_j] * kernel)

    return feature_map
```

Le noyau est appliqué sur l'image en se déplaçant dans la matrice d'entrée. Pour chaque position (i,j) de la sortie , le filtre est appliqué à la fenêtre correspondante de l'image d'entrée.

- **start_i et start_j** : Ce sont les indices de début de la fenêtre sur l'image d'entrée.
- **end_i et end_j** : Ce sont les indices de fin de la fenêtre.
- La fenêtre correspondante de l'image d'entrée est extraite et multipliée élément par élément avec le noyau. Ensuite, on effectue la somme de ces produits et on la place dans la carte de caractéristiques à la position (i,j).

3) ReLU

La fonction relu applique une activation ReLU (Rectified Linear Unit), qui remplace les valeurs négatives par zéro, introduisant ainsi de la non-linéarité dans le réseau.

```
def relu(feature_map):
    activated_map = np.maximum(0, feature_map)
    return activated_map
```

4) Max Pooling

La fonction pooling applique une opération de pooling, qui réduit la taille des cartes de caractéristiques en conservant la valeur maximale d'une fenêtre de taille donnée.

```
def pooling(activated_map, pooling_size=2, stride=2):
    output_height = (activated_map.shape[0] - pooling_size) // stride + 1
    output_width = (activated_map.shape[1] - pooling_size) // stride + 1
    pooled_map = np.zeros((output_height, output_width))

    for i in range(0, output_height):
        for j in range(0, output_width):
            start_i = i * stride
            start_j = j * stride
            end_i = start_i + pooling_size
            end_j = start_j + pooling_size

            window = activated_map[start_i:end_i, start_j:end_j]
            pooled_map[i, j] = np.max(window)

    return pooled_map
```

5) Flattening

La fonction flatten convertit la carte de caractéristiques en un vecteur unidimensionnel afin qu'il puisse être passé à une couche entièrement connectée.

```
def flatten(pooled_map):
    pooled_shape = pooled_map.shape

    total_elements = pooled_shape[0] * pooled_shape[1]

    flattened_vector = np.zeros(total_elements)

    index = 0
    for i in range(pooled_shape[0]):
        for j in range(pooled_shape[1]):
            flattened_vector[index] = pooled_map[i, j]
            index += 1

    return flattened_vector
```

À chaque itération, la valeur pooled_map[i, j] est stockée dans le vecteur flattened_vector à la position correspondante donnée par index. Une fois que toutes les valeurs de la carte de caractéristiques ont été insérées dans le vecteur, la fonction retourne ce vecteur **aplati**.

6) Fully Connected Layer

La fonction fully_connected calcule la sortie d'une couche entièrement connectée en effectuant un produit matriciel entre le vecteur aplati et les poids de la couche, puis en ajoutant un biais.

```
def fully_connected(flattened_vector, weights, biases):
    return np.dot(weights, flattened_vector) + biases
```

7) Softmax

La fonction softmax convertit les scores bruts de la couche entièrement connectée en probabilités, qui peuvent être interprétées comme des prédictions de classe.

```
def softmax(logits):
    exp_values = np.exp(logits - np.max(logits))
    return exp_values / np.sum(exp_values)
```

8) Perte de Cross-Entropy

La fonction cross_entropy_loss calcule la perte de *cross-entropy* entre les étiquettes réelles et les prédictions, une mesure courante utilisée pour les problèmes de classification multi-classe.

```
def cross_entropy_loss(y_true, y_pred):
    return -np.sum(y_true * np.log(y_pred + 1e-9))
```

9) Backward propagation et Mise à jour des Poids

Les gradients sont calculés à l'aide de la rétropropagation et utilisés pour mettre à jour les poids et les biais du modèle, afin de réduire la perte à chaque itération de l'entraînement.

- **Gradient :**

La fonction compute_gradients calcule les gradients des poids et des biais dans la couche fully connected layer d'un réseau neuronal, en utilisant la méthode de la **Backpropagation**.

```python
def compute_gradients(flattened_output, y_true, logits, weights):
    m = y_true.shape[0]
    y_pred = softmax(logits)
    dL_dz = y_pred - y_true
    dL_dw = np.dot(dL_dz.reshape(-1, 1), flattened_output.reshape(1, -1)) / m
    dL_db = dL_dz / m
    dL_dflattened = np.dot(weights.T, dL_dz)
    return dL_dw, dL_db, dL_dflattened
```

Calcul des gradients :

- **y_pred** : La sortie de la fonction softmax appliquée aux logits.
- **dL_dz** : Le gradient de la fonction de perte par rapport aux logits (z), c'est-à-dire la différence entre les prédictions y_pred et les étiquettes réelles y_true.
- **dL_dw** : Le gradient de la fonction de perte par rapport aux poids. Il est calculé en effectuant un produit matriciel entre dL_dz et flattened_output.
- **dL_db** : Le gradient de la fonction de perte par rapport aux biais.
- **dL_dflattened** : Le gradient de la fonction de perte par rapport à l'entrée aplatie de la couche précédente, calculé en multipliant les poids transposés par dL_dz.

❖ Backward propagation à travers la Couche de Pooling

La fonction calcule le gradient de la **Backpropagation** pour une opération de **max pooling**. Max pooling est une opération qui sélectionne la valeur maximale dans une fenêtre de la carte d'activation.

```python
def max_pooling_backward(dL_dpool, activated_map, pooling_size=2, stride=2):
    dL_dmap = np.zeros_like(activated_map)
    output_height, output_width = dL_dpool.shape
    for i in range(output_height):
        for j in range(output_width):
            start_i = i * stride
            start_j = j * stride
            window = activated_map[start_i:start_i + pooling_size, start_j:start_j + pooling_size]
            max_value = np.max(window)
            for m in range(pooling_size):
                for n in range(pooling_size):
                    if window[m, n] == max_value:
                        dL_dmap[start_i + m, start_j + n] = dL_dpool[i, j]
                        break
    return dL_dmap
```

Calcul du gradient :

- Pour chaque position dans dL_dpool, la fonction identifie la fenêtre de pooling correspondante dans activated_map.
- Puis, elle trouve la valeur maximale dans cette fenêtre et place le gradient de la sortie (dL_dpool[i, j]) uniquement sur l'élément maximal.
- Pour tous les autres éléments de la fenêtre, le gradient est nul.

❖ Backward propagation à travers la Convolution

La fonction calcule le gradient de la rétropropagation pour une opération de convolution. La rétropropagation dans les CNN consiste à calculer les gradients par rapport à l'entrée (dx) et aux filtres (dfilter).

```python
def conv_backward(dL_dconv, x, kernel, stride=1, padding=0):
    kernel_rotated = np.rot90(kernel, 2)
    h_x, w_x = x.shape
    h_k, w_k = kernel.shape
    h_out, w_out = dL_dconv.shape

    # Pad x and dx
    x_padded = np.pad(x, ((padding, padding), (padding, padding)), mode='constant', constant_values=0)
    dx_padded = np.zeros_like(x_padded)
    dfilter = np.zeros_like(kernel)

    # Compute dfilter
    for i in range(h_out):
        for j in range(w_out):
            start_i = i * stride
            start_j = j * stride
            x_window = x_padded[start_i:start_i + h_k, start_j:start_j + w_k]
            dfilter += dL_dconv[i, j] * x_window

    # Compute dx
    for i in range(h_x + 2 * padding):
        for j in range(w_x + 2 * padding):
            for m in range(h_out):
                for n in range(w_out):
                    i_out = i - m * stride
                    j_out = j - n * stride
                    if 0 <= i_out < h_k and 0 <= j_out < w_k:
                        dx_padded[i, j] += dL_dconv[m, n] * kernel_rotated[i_out, j_out]

    if padding > 0:
        dx = dx_padded[padding:-padding, padding:-padding]
    else:
        dx = dx_padded

    return dx, dfilter
```

Calcul des gradients :

- **dfilter** : Le gradient de la fonction de perte par rapport au filtre de convolution est calculé en effectuant un produit de dL_dconv avec les fenêtres correspondantes de x.
- **dx** : Le gradient de la fonction de perte par rapport à l'entrée est calculé en faisant une rétropropagation à travers le noyau. On fait une opération de convolution inverse en utilisant le filtre retourné (rotation de 180 degrés).

VI. Entraînement et Test

Le modèle est entraîné sur le jeu de données MNIST pendant plusieurs époques, puis testé sur le jeu de test pour évaluer sa précision.

```python
# Load and normalize data
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train = x_train.astype(np.float32) / 255.0
x_test = x_test.astype(np.float32) / 255.0

# Convert labels to one-hot encoding
num_classes = 10
y_train_one_hot = np.zeros((y_train.size, num_classes))
y_train_one_hot[np.arange(y_train.size), y_train] = 1

# Initialize parameters
np.random.seed(42)  # For reproducibility
kernel = np.random.randn(3, 3) * 0.01
epochs = 3
learning_rate = 0.01

# Fully connected layer weights and biases
weights = np.random.randn(num_classes, 196) * 0.01
biases = np.zeros(num_classes)
```

Les images de MNIST sont extraites de la base de données et divisées en ensembles d'entraînement (x_train, y_train) et de test (x_test, y_test). Les valeurs des pixels des images sont initialement comprises entre 0 et

255. Chaque label est ensuite transformé en un vecteur de longueur 10 (puisque MNIST comprend 10 classes), où l'élément correspondant à la classe est égal à 1, et tous les autres éléments sont égaux à 0. Les poids de la couche complètement connectée sont également initialisés de manière aléatoire, avec une forme adaptée pour correspondre à la taille de l'entrée après l'étape de pooling.

```
# Training loop with full backward propagation
for epoch in range(epochs):
    total_loss = 0
    correct_predictions = 0
    print(f"Epoch {epoch + 1}/{epochs}:")
    for i in tqdm(range(len(x_train)), desc="Training"):
        test_image = x_train[i]
        if test_image.ndim == 3:
            test_image = np.mean(test_image, axis=-1)
        y_true = y_train_one_hot[i]
        conv_output = convolution(test_image, kernel, stride=1, padding=1)
        # Activation (ReLU)
        relu_output = relu(conv_output)
        # Max Pooling
        pooled_output = pooling(relu_output, pooling_size=2, stride=2)
        # Flatten
        flattened_output = flatten(pooled_output)
        # Fully Connected Layer
        logits = fully_connected(flattened_output, weights, biases)
        # Softmax
        output_probabilities = softmax(logits)
        # Loss
        loss = cross_entropy_loss(y_true, output_probabilities)
        total_loss += loss
        # Prediction
        predicted_label = np.argmax(output_probabilities)
        true_label = np.argmax(y_true)
        if predicted_label == true_label:
            correct_predictions += 1
        # Gradients for fully connected layer
        dL_dw, dL_db, dL_dflattened = compute_gradients(flattened_output, y_true, logits, weights)
        # Reshape gradient to match pooled output shape
        dL_dpool = dL_dflattened.reshape(pooled_output.shape)
        # Backpropagate through max pooling
        dL_drelu = max_pooling_backward(dL_dpool, relu_output, pooling_size=2, stride=2)
        # Backpropagate through ReLU directly
        dL_dconv = dL_drelu.copy()
        dL_dconv[conv_output <= 0] = 0
        # Backpropagate through convolution
        dx, dfilter = conv_backward(dL_dconv, test_image, kernel, stride=1, padding=1)
        weights -= learning_rate * dL_dw
        biases -= learning_rate * dL_db
        kernel -= learning_rate * dfilter  # Update convolution filter

    average_loss = total_loss / len(x_train)
    accuracy = correct_predictions / len(x_train)
    print(f'Average Loss: {average_loss:.4f}, Accuracy: {accuracy:.4f}')
```

L'entraînement s'effectue sur plusieurs époques , où chaque époque représente un passage sur l'ensemble des données d'entraînement.

- **Forward propagation :**
  1. **Convolution** : L'image est passée dans une couche de convolution avec un filtre de 3x3.
  2. **Activation (ReLU)** : Une fonction d'activation ReLU est appliquée pour ajouter de la non-linéarité.
  3. **Pooling (Max Pooling)** : Le résultat de ReLU passe par une couche de max pooling pour réduire la taille de la carte de caractéristiques.
  4. **Flattening** : La sortie après pooling est aplatie pour être utilisée dans une couche complètement connectée.
  5. **Fully connected**: Le vecteur aplati est passé à travers une fully connected avec les poids et biais.
  6. **Softmax** : La sortie de la couche complètement connectée est passée par une fonction softmax pour obtenir des probabilités pour chaque classe.

```
conv_output = convolution(test_image, kernel, stride=1, padding=1)
# Activation (ReLU)
relu_output = relu(conv_output)
# Max Pooling
pooled_output = pooling(relu_output, pooling_size=2, stride=2)
# Flatten
flattened_output = flatten(pooled_output)
# Fully Connected Layer
logits = fully_connected(flattened_output, weights, biases)
# Softmax
output_probabilities = softmax(logits)
```

- **Calcul de la Perte et Mise à Jour des Paramètres**

La perte est calculée à l'aide de la cross-entropy loss, qui est une fonction couramment utilisée pour des problèmes de classification. Les prédictions sont comparées aux labels réels et la précision est calculée.

```
# Loss
loss = cross_entropy_loss(y_true, output_probabilities)
total_loss += loss
# Prediction
predicted_label = np.argmax(output_probabilities)
true_label = np.argmax(y_true)
if predicted_label == true_label:
    correct_predictions += 1
```

- **Backward propagation :**

Calcul des gradients : Les gradients de la fonction de perte par rapport aux paramètres du modèle sont calculés en utilisant la Backpropagation :

- **Gradients pour la fully connected**: Calcul des gradients des poids et des biais.
- **Max Pooling** : Propagation des gradients à travers la couche de pooling.
- **ReLU** : Propagation des gradients à travers la fonction d'activation ReLU.
- **Convolution** : Propagation des gradients à travers la couche de convolution pour obtenir les gradients des filtres.

```
# Gradients for fully connected layer
dL_dw, dL_db, dL_dflattened = compute_gradients(flattened_output, y_true, logits, weights)
# Reshape gradient to match pooled output shape
dL_dpool = dL_dflattened.reshape(pooled_output.shape)
# Backpropagate through max pooling
dL_drelu = max_pooling_backward(dL_dpool, relu_output, pooling_size=2, stride=2)
# Backpropagate through ReLU directly
dL_dconv = dL_drelu.copy()
dL_dconv[conv_output <= 0] = 0
# Backpropagate through convolution
dx, dfilter = conv_backward(dL_dconv, test_image, kernel, stride=1, padding=1)
```

- **Mise à Jour des Paramètres**

Les paramètres du modèle (poids, biais, noyau de convolution) sont mis à jour en utilisant le gradient descent . Le taux d'apprentissage (learning rate) contrôle la taille de ces ajustements.

```
weights -= learning_rate * dL_dw
biases -= learning_rate * dL_db
kernel -= learning_rate * dfilter
```

- **Affichage de la Perte et de la Précision**

La **perte moyenne** et la **précision** sur les données d'entraînement sont calculées et affichées à la fin de chaque époque.

```
average_loss = total_loss / len(x_train)
accuracy = correct_predictions / len(x_train)
print(f'Average Loss: {average_loss:.4f}, Accuracy: {accuracy:.4f}')
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
Epoch 1/2:
Training: 100%|██████████| 60000/60000 [2:17:26<00:00,  7.28it/s]
Average Loss: 0.4149, Accuracy: 0.8735
Epoch 2/2:
Training: 100%|██████████| 60000/60000 [2:14:05<00:00,  7.46it/s]Average Loss: 0.3125, Accuracy: 0.9085
```

Le teste d'un modèle CNN effectuant les opérations suivantes pour chaque image du jeu de test : convolution, activation ReLU, max pooling, flatten, propagation à travers une fully connected, application de la fonction softmax pour obtenir des probabilités, puis comparaison avec l'étiquette réelle pour compter les prédictions correctes. La précision du modèle est calculée à la fin du test.

```
# Test the model
test_image = x_test[400]
if test_image.ndim == 3:
    test_image = np.mean(test_image, axis=-1)

# Forward pass for prediction
conv_output = convolution(test_image, kernel, stride=1, padding=1)
relu_output = relu(conv_output)
pooled_output = pooling(relu_output, pooling_size=2, stride=2)
flattened_output = flatten(pooled_output)

# Perform CNN forward pass
logits = fully_connected(flattened_output, weights, biases)
output_probabilities = softmax(logits)

# Print results
predicted_class = np.argmax(output_probabilities)
print("Output probabilities:", output_probabilities)
print("Predicted class:", predicted_class)

# Show the predicted image
plt.imshow(test_image, cmap='gray')
plt.title(f'Predicted Class: {predicted_class}')
plt.axis('off')
plt.show()
```

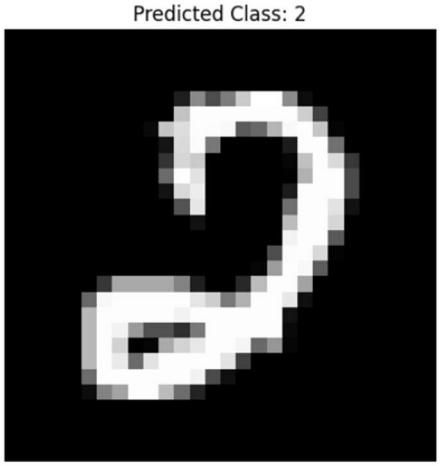
```
correct_predictions = 0
for i in tqdm(range(len(x_test)), desc="Testing"):
    test_image = x_test[i]
    if test_image.ndim == 3:
        test_image = np.mean(test_image, axis=-1)
    y_true = y_test[i]

    # Forward Pass
    conv_output = convolution(test_image, kernel, stride=1, padding=1)
    relu_output = relu(conv_output)
    pooled_output = pooling(relu_output, pooling_size=2, stride=2)
    flattened_output = flatten(pooled_output)
    logits = fully_connected(flattened_output, weights, biases)
    output_probabilities = softmax(logits)
    predicted_label = np.argmax(output_probabilities)

    if predicted_label == y_true:
        correct_predictions += 1

accuracy = correct_predictions / len(x_test)
print(f'Test Accuracy: {accuracy:.4f}')
```

```
Output probabilities: [6.02304814e-02 1.61433925e-10 9.37145076e-01 1.84385776e-05
 1.72390508e-07 3.40384689e-05 3.23415347e-05 4.09386545e-08
 2.52854752e-03 1.08625973e-05]
Predicted class: 2
```




```
Test Accuracy: 0.9051
```

VII. Conclusion :

Ce rapport présente l'implémentation d'un CNN from scratch, en construisant manuellement chaque couche fondamentale : convolution, activation, pooling et fully connected, sans recourir à des bibliothèques de deep learning préexistantes. Cette approche nous a permis d'acquérir une compréhension approfondie des principes mathématiques et algorithmiques des CNN, en suivant le processus complet, de la propagation avant à la rétropropagation (backward propagation) du gradient.

Les résultats obtenus montrent que notre modèle parvient à classer les images avec une précision satisfaisante, bien que des améliorations soient possibles. L'ajustement des hyperparamètres, l'ajout de régularisation, ainsi que l'introduction de couches plus profondes pourraient permettre de mieux capturer les caractéristiques complexes des données.

Enfin, ce travail ouvre la voie à des applications futures dans des domaines tels que la détection d'objets, la segmentation d'images, ainsi qu'à l'exploration de modèles plus complexes, comme les architectures ResNet ou VGG ou Unet … , pour améliorer les performances.