人工智慧概論 HW4

4112064214 侯竣奇

June 22, 2025

1 Summary

本次 HW4 作業的要求是在不使用 PyTorch 及 TensorFlow 等深度學習框架的前提下，使用 Numpy 來訓練一個卷積神經網路 (Convolutional Neural Network, CNN) 來分類 MNIST 資料集。可以另外訓練 CIFAR-10 或 ImageNet 資料集作為加分。

我在訓練過程中遇到過巢狀迴圈執行過於緩慢的問題，改用 img2col 的方式來加速，也使用 Cupy 來進行 GPU 加速。

訓練 MNIST 可以達到 98.60% 的準確率，同時我也完成了各項性能計算（精確率、召回率）、繪制學習曲線、卷積核視覺化（加分項）等任務。

作為加分挑戰，我也將模型應用於更複雜的 CIFAR-10 彩色圖片資料集，在這個資料集上取得了 59.01% 的準確率，並同樣完成了相關的性能評估與視覺化。

其餘詳細的性能分析與解說在檔案最後 Conclusion 的部分有提及。

2 Step 0: 下載資料集

[ ]:
```python
# 延用 HW3 中下載 mnist 的程式
def get_mnist():
    """
    The code to download the mnist data original came from
    https://cntk.ai/pythondocs/CNTK_103A_MNIST_DataLoader.html
    """

    import gzip
    import numpy as np
    import os
    import struct

    from urllib.request import urlretrieve

    def load_data(src, num_samples):
        print("Downloading " + src)
        gzfname, h = urlretrieve(src, "./delete.me")
        print("Done.")
        try:
            with gzip.open(gzfname) as gz:
```

```
                n = struct.unpack("I", gz.read(4))
                # Read magic number.
                if n[0] != 0x3080000:
                    raise Exception("Invalid file: unexpected magic␣
↪number.")
                # Read number of entries.
                n = struct.unpack(">I", gz.read(4))[0]
                if n != num_samples:
                    raise Exception(f"Invalid file: expected␣
↪{num_samples} entries.")
                crow = struct.unpack(">I", gz.read(4))[0]
                ccol = struct.unpack(">I", gz.read(4))[0]
                if crow != 28 or ccol != 28:
                    raise Exception("Invalid file: expected 28 rows/
↪cols per image.")
                # Read data.
                res = np.frombuffer(gz.read(num_samples * crow *␣
↪ccol), dtype=np.uint8)
        finally:
            os.remove(gzfname)
        # 這次我們不在這裡做 normalize，留到 CuPy 陣列處理
        return res.reshape((num_samples, crow, ccol))

    def load_labels(src, num_samples):
        print("Downloading " + src)
        gzfname, h = urlretrieve(src, "./delete.me")
        print("Done.")
        try:
            with gzip.open(gzfname) as gz:
                n = struct.unpack("I", gz.read(4))
                # Read magic number.
                if n[0] != 0x1080000:
                    raise Exception("Invalid file: unexpected magic␣
↪number.")
                # Read number of entries.
                n = struct.unpack(">I", gz.read(4))
                if n[0] != num_samples:
                    raise Exception(f"Invalid file: expected␣
↪{num_samples} rows.")
                # Read labels.
                res = np.frombuffer(gz.read(num_samples), dtype=np.
↪uint8)
        finally:
            os.remove(gzfname)
        return res.reshape(num_samples)
```

```
    def try_download(data_source, label_source, num_samples):
        data = load_data(data_source, num_samples)
        labels = load_labels(label_source, num_samples)
        return data, labels

    server = "https://raw.githubusercontent.com/fgnt/mnist/master"

    # URLs for the train image and label data
    url_train_image = f"{server}/train-images-idx3-ubyte.gz"
    url_train_labels = f"{server}/train-labels-idx1-ubyte.gz"
    num_train_samples = 60000

    print("Downloading train data")
    train_features, train_labels = try_download(url_train_image,␣
 ↪url_train_labels, num_train_samples)

    # URLs for the test image and label data
    url_test_image = f"{server}/t10k-images-idx3-ubyte.gz"
    url_test_labels = f"{server}/t10k-labels-idx1-ubyte.gz"
    num_test_samples = 10000

    print("Downloading test data")
    test_features, test_labels = try_download(url_test_image,␣
 ↪url_test_labels, num_test_samples)

    return train_features, train_labels, test_features, test_labels
```

[ ]:

```
train_features_np, train_labels_np, test_features_np, test_labels_np =␣
 ↪get_mnist()
```

```
Downloading train data
Downloading https://raw.githubusercontent.com/fgnt/mnist/master/
 ↪train-images-
idx3-ubyte.gz
Done.
Downloading https://raw.githubusercontent.com/fgnt/mnist/master/
 ↪train-labels-
idx1-ubyte.gz
Done.
Downloading test data
Downloading https://raw.githubusercontent.com/fgnt/mnist/master/
 ↪t10k-images-
idx3-ubyte.gz
Done.
Downloading https://raw.githubusercontent.com/fgnt/mnist/master/
 ↪t10k-labels-
idx1-ubyte.gz
```

```
Done.
```

3 Step 1: 資料前處理 (Data Preprocessing)

資料下載完成後，接下來我們要進行一系列的處理，才能將其輸入到 CNN 模型中。

3.1 Step 1-1: 讀取資料並轉換為 CuPy 陣列

在 HW4 中我們改用 CuPy 來加速原先 Numpy 的運算速度

[ ]:
```python
import cupy as cp
import numpy as np

# 使用 cp.asarray() 將資料從 CPU memory 移至 GPU memory
train_x_orig = cp.asarray(train_features_np)
train_y_orig = cp.asarray(train_labels_np)
test_x_orig = cp.asarray(test_features_np)
test_y_orig = cp.asarray(test_labels_np)

# --- 資料塑形與正規化 (Reshape & Normalize) ---
# CNN 需要的輸入格式為 (樣本數, 色彩頻道, 高度, 寬度)
# MNIST 是灰階圖片，所以 Channel 為 1
# 我們也在此將像素值從 0-255 正規化到 0-1 之間
train_x = train_x_orig.reshape(train_x_orig.shape[0], 1, 28, 28) / 255.
↪0
test_x = test_x_orig.reshape(test_x_orig.shape[0], 1, 28, 28) / 255.0
```

[ ]:
```python
# 驗證一下 shape 和 type
print("--- Shapes of CuPy arrays ---")
print("train_x.shape:", train_x.shape)
print("train_y_orig.shape:", train_y_orig.shape)
print("test_x.shape:", test_x.shape)
print("test_y_orig.shape:", test_y_orig.shape)
print("\n--- Data types ---")
print("type of train_x:", type(train_x))
print("type of train_y_orig:", type(train_y_orig))
```

```
--- Shapes of CuPy arrays ---
train_x.shape: (60000, 1, 28, 28)
train_y_orig.shape: (60000,)
test_x.shape: (10000, 1, 28, 28)
test_y_orig.shape: (10000,)

--- Data types ---
type of train_x: <class 'cupy.ndarray'>
type of train_y_orig: <class 'cupy.ndarray'>
```

3.2 Step 1-2: 對 Label 進行 One-Hot Encode

```
[ ]: def one_hot_encode(labels, num_classes):
         # 創建一個全為 0 的矩陣，shape 為 (樣本數, 類別數)
         one_hot = cp.zeros((labels.size, num_classes))
         # 在對應的類別位置上填上 1
         one_hot[cp.arange(labels.size), labels] = 1
         return one_hot
```

```
[ ]: # 我們的類別是數字 0-9，所以有 10 個類別
     num_classes = 10
     train_y = one_hot_encode(train_y_orig, num_classes)
     test_y = one_hot_encode(test_y_orig, num_classes)

     print("One-Hot encode 前的 train_y.shape:", train_y_orig.shape)
     print("One-Hot 後的 train_y.shape:", train_y.shape)

     print(f"一個 One-Hot 編碼範例 (原始 label 是 {train_y_orig[0]}):␣
       ↪{train_y[0]}")
```

```
One-Hot encode 前的 train_y.shape: (60000,)
One-Hot 後的 train_y.shape: (60000, 10)
一個 One-Hot 編碼範例 (原始 label 是 5): [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

4 Step 2: 實作 CNN

4.1 Step 2-2: 初始化權重

初始化 CNN 的權重與偏置

參數:

- `input_dim` – 輸入圖片的維度 (C, H, W)
- `n_filters`, `filter_size`, `conv_stride`, `conv_padding` – 卷積層的 hyperparameter
- `pool_size`, `pool_stride` – 池化層的超參數
- `n_h`, `n_y` – 全連接層與輸出層的神經元數量

Return:

- `parameters` – 一個包含所有權重與 bias 的 Python dict

```
[ ]: def initialize_parameters(input_dim=(1, 28, 28),
                               n_filters=8, filter_size=3, conv_stride=1,␣
       ↪conv_padding=1,
                               pool_size=2, pool_stride=2,
                               n_h=128, n_y=10):

         cp.random.seed(42)

         C_in, H_in, W_in = input_dim
```

```python
    # 1. 初始化卷積層權重
    W_conv = cp.random.randn(n_filters, C_in, filter_size,␣
↪filter_size) * 0.01
    b_conv = cp.zeros((n_filters, 1))

    # 2. 動態計算全連接層的輸入維度
    # 計算卷積層的輸出維度
    H_conv = int((H_in - filter_size + 2 * conv_padding) /␣
↪conv_stride) + 1
    W_conv_out = int((W_in - filter_size + 2 * conv_padding) /␣
↪conv_stride) + 1

    # 計算池化層的輸出維度（池化層不補零）
    H_pool = int((H_conv - pool_size) / pool_stride) + 1
    W_pool_out = int((W_conv_out - pool_size) / pool_stride) + 1

    # 扁平化後的維度
    fc_input_dim = n_filters * H_pool * W_pool_out

    # print(f" 根據設定計算，全連接層的輸入維度為: {fc_input_dim}")

    # 3. 初始化全連接層與輸出層權重
    W_fc = cp.random.randn(fc_input_dim, n_h) * 0.01
    b_fc = cp.ones((1, n_h)) * 0.01
    W_out = cp.random.randn(n_h, n_y) * 0.01
    b_out = cp.zeros((1, n_y))

    parameters = {
        "W_conv": W_conv, "b_conv": b_conv,
        "W_fc": W_fc, "b_fc": b_fc,
        "W_out": W_out, "b_out": b_out
    }

    return parameters
```

[ ]:

```python
# MNIST
# 輸入維度: (1, 28, 28), 卷積設定: 3x3 filter, stride 1, padding 1 -> 輸
出 (28, 28)
# 池化設定: 2x2, stride 2 -> 輸出 (14, 14)

params = initialize_parameters(input_dim=(1, 28, 28), n_filters=8,␣
↪filter_size=3)

print("\n--- Parameter Shapes ---")
print("W_conv Shape: " + str(params["W_conv"].shape))
print("W_fc Shape: " + str(params["W_fc"].shape))
```

```
--- Parameter Shapes ---
W_conv Shape: (8, 1, 3, 3)
W_fc Shape: (1568, 128)
```

4.2 Step 2-2: 實作卷積層 Forward Propagation

zero_pad: 對 CuPy 陣列進行補零

參數:

- `X` – CuPy 陣列, shape (N, C, H, W)
- `pad` – 整數，代表要補的圈數

Return:

- `X_pad` – 補零後的陣列, shape (N, C, H + 2 * pad, W + 2 * pad)

---

conv_forward: CNN 的前向傳播

參數:

- `A_prev` – 上一層的輸出 (輸入圖片), shape (N, C_prev, H_prev, W_prev)
- `W` – Filter 權重, shape (n_filters, C_prev, f, f)
- `b` – bias, shape (n_filters, 1)
- `stride` – 滑動步長
- `padding` – 補零圈數

Return:

- `Z` – 卷積層的輸出, shape (N, n_filters, H_out, W_out)
- `cache` – 用於反向傳播的快取

[ ]:
```python
def zero_pad(X, pad):
    # cp.pad 的用法和 np.pad 相同
    # ((0, 0), (0, 0), (pad, pad), (pad, pad)) 分別對應 4 個維度的補零設定
    X_pad = cp.pad(X, ((0, 0), (0, 0), (pad, pad), (pad, pad)), mode='constant', constant_values=0)
    return X_pad
```

[ ]:
```python
def conv_forward(A_prev, W, b, stride=1, padding=1):
    # 1. 從 W 的維度取得濾波器資訊
    (n_filters, C_prev, f, f) = W.shape
    # 從 A_prev 的維度取得輸入資訊
    (N, C_prev, H_prev, W_prev) = A_prev.shape

    # 2. 計算輸出特徵圖的維度
    H_out = int((H_prev - f + 2 * padding) / stride) + 1
    W_out = int((W_prev - f + 2 * padding) / stride) + 1
```

```python
    # 3. 初始化輸出 Z
    Z = cp.zeros((N, n_filters, H_out, W_out))

    # 4. 對輸入 A_prev 進行補零
    A_prev_pad = zero_pad(A_prev, padding)

    # 5. 進行卷積運算 (使用巢狀迴圈)
    for n in range(N): # 遍歷每一張圖片樣本
        a_prev_pad = A_prev_pad[n, :, :, :]
        for c in range(n_filters): # 遍歷每一個 Filter
            for h in range(H_out): # 遍歷輸出的垂直方向
                for w in range(W_out): # 遍歷輸出的水平方向
                    # 定位當前滑動窗口的垂直和水平起始位置
                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f

                    # 選取當前的滑動窗口
                    a_slice_prev = a_prev_pad[:, vert_start:vert_end,␣
↪horiz_start:horiz_end]

                    # 進行對應元素相乘後相加，再加上 bias
                    Z[n, c, h, w] = cp.sum(a_slice_prev * W[c, :, :, :
↪]) + b[c]

    # 儲存快取以供反向傳播使用
    cache = (A_prev, W, b, stride, padding)

    return Z, cache
```

4.2.1 Step 2-2-2: 優化前向傳播

在執行過程我發現這種四層 for 迴圈的寫法效能實在太低，所以我們改用 im2col 的寫法來改進

[ ]:
```python
def get_im2col_indices(x_shape, field_height, field_width, padding=1,␣
↪stride=1):
    # First figure out what the output shape is
    N, C, H, W = x_shape
    assert (H + 2 * padding - field_height) % stride == 0
    assert (W + 2 * padding - field_width) % stride == 0
    out_height = (H + 2 * padding - field_height) // stride + 1
    out_width = (W + 2 * padding - field_width) // stride + 1

    i0 = cp.repeat(cp.arange(field_height), field_width)
    i0 = cp.tile(i0, C)
```

```python
    i1 = stride * cp.repeat(cp.arange(out_height), out_width)
    j0 = cp.tile(cp.arange(field_width), field_height * C)
    j1 = stride * cp.tile(cp.arange(out_width), out_height)
    i = i0.reshape(-1, 1) + i1.reshape(1, -1)
    j = j0.reshape(-1, 1) + j1.reshape(1, -1)

    k = cp.repeat(cp.arange(C), field_height * field_width).reshape(-1, 1)

    return (k, i, j)

def im2col_gpu(x, field_height, field_width, padding=1, stride=1):
    """ An implementation of im2col on GPU """
    # Zero-pad the input
    p = padding
    x_padded = cp.pad(x, ((0, 0), (0, 0), (p, p), (p, p)), mode='constant')

    k, i, j = get_im2col_indices(x.shape, field_height, field_width, padding, stride)

    cols = x_padded[:, k, i, j]
    C = x.shape[1]
    cols = cols.transpose(1, 2, 0).reshape(field_height * field_width * C, -1)
    return cols
```

[ ]:

```python
def conv_forward_fast(A_prev, W, b, stride=1, padding=1):

    # 從 A_prev 的維度取得輸入資訊
    N, C_prev, H_prev, W_prev = A_prev.shape

    # 從 W (權重) 的維度取得 Filter 資訊
    n_filters, _, f, _ = W.shape

    # 計算輸出維度
    H_out = (H_prev - f + 2 * padding) // stride + 1
    W_out = (W_prev - f + 2 * padding) // stride + 1

    # 使用 im2col 將輸入圖片轉換為矩陣
    A_col = im2col_gpu(A_prev, f, f, padding=padding, stride=stride)

    # 將 Filter 權重攤平成矩陣
    W_row = W.reshape(n_filters, -1)

    # 核心運算：一次矩陣乘法 + bias
    Z = W_row @ A_col + b
```

```python
    # 將結果塑形成最終的輸出維度
    Z = Z.reshape(n_filters, H_out, W_out, N)
    Z = Z.transpose(3, 0, 1, 2) # 調整維度順序為 (N, n_filters, H_out,␣
↪W_out)

    # 快取 A_col 供反向傳播使用
    cache = (A_prev, W, b, stride, padding, A_col)

    return Z, cache
```

4.3 Step 2-3: 實作 Activation Function (ReLU)

ReLU 的反向傳播

參數:

- `dA` – 流回來的梯度
- `cache_Z` – 前向傳播時的 Z 值

Return:

- `dZ` – 對 Z 的梯度

[ ]:
```python
def relu(Z):
    return cp.maximum(0, Z)

def relu_backward(dA, cache_Z):
    dZ = cp.array(dA, copy=True)
    # 當 Z <= 0 時，梯度也為 0
    dZ[cache_Z <= 0] = 0
    return dZ
```

4.4 Step 2-4: 實作池化層 Forward Propagation

實現 Max Pooling 的前向傳播

參數:

- `A_prev` – 上一層的輸出 (ReLU), shape (N, C, H_prev, W_prev)
- `pool_size` – 池化窗口的大小
- `stride` – 滑動步長

Return:

- `A_out` – 池化層的輸出, shape (N, C, H_out, W_out)
- `cache` – 用於反向傳播的快取，包含 A_prev 和超參數

[ ]:
```python
def max_pool_forward(A_prev, pool_size=2, stride=2):
    """
    """
```

```
    # 取得輸入維度
    (N, C, H_prev, W_prev) = A_prev.shape

    # 計算輸出維度
    H_out = int((H_prev - pool_size) / stride) + 1
    W_out = int((W_prev - pool_size) / stride) + 1

    # 初始化輸出
    A_out = cp.zeros((N, C, H_out, W_out))

    # 進行最大池化運算
    for n in range(N):
        for c in range(C):
            for h in range(H_out):
                for w in range(W_out):
                    # 定位當前滑動窗口
                    vert_start = h * stride
                    vert_end = vert_start + pool_size
                    horiz_start = w * stride
                    horiz_end = horiz_start + pool_size

                    # 選取滑動窗口
                    a_slice = A_prev[n, c, vert_start:vert_end,␣
↪horiz_start:horiz_end]

                    # 在窗口中找到最大值並賦給輸出
                    A_out[n, c, h, w] = cp.max(a_slice)

    cache = (A_prev, (pool_size, stride))
    return A_out, cache
```

4.4.1 Step 2-4-2: 優化池化層前向傳播

[ ]:
```
def max_pool_forward_fast(A_prev, pool_size=2, stride=2):

    N, C, H_prev, W_prev = A_prev.shape

    # 計算輸出維度
    H_out = 1 + (H_prev - pool_size) // stride
    W_out = 1 + (W_prev - pool_size) // stride

    # 透過 reshape 和 transpose 將每個池化窗口的元素分組
    A_reshaped = A_prev.reshape(N, C, H_out, pool_size, W_out,␣
↪pool_size)
    A_transposed = A_reshaped.transpose(0, 1, 2, 4, 3, 5)
```

```
    # 在每個窗口內取最大值
    A_out = cp.max(A_transposed, axis=(4, 5))

    cache = (A_prev, A_out, (pool_size, stride))
    return A_out, cache
```

4.5 Step 2-5: 完整的 Forward Propagation

流程如下：

1. CONV → RELU：對圖片進行卷積運算，然後傳給 ReLU
2. POOL: 對 ReLU 的輸出進行 Max Pooling
3. FLATTEN: 將池化後的特徵圖攤平成一維向量
4. FC→RELU: 將攤平的向量輸入全連接層，再通過一次 ReLU
5. FC→SOFTMAX: 最後通過輸出層，並使用 Softmax 函數得到最終的分類機率

Softmax 激活函數 (CuPy 版本)

為了數值穩定性，先減去 Z 中的最大值

```
[ ]: def softmax(Z):
         # Z.shape: (N, 10), N 是樣本數
         exp_scores = cp.exp(Z - cp.max(Z, axis=1, keepdims=True))
         return exp_scores / cp.sum(exp_scores, axis=1, keepdims=True)
```

實現 CNN 的完整前向傳播

參數:

- X – 輸入資料, shape (N, C, H, W)
- parameters – 包含所有權重的字典

Return:

- A_out – 模型的最終輸出 (Softmax 的結果)
- cache – 包含所有中間層計算結果的字典，供反向傳播使用

```
[ ]: def forward_propagation(X, parameters):
         # 1. 取出權重
         W_conv = parameters['W_conv']
         b_conv = parameters['b_conv']
         W_fc = parameters['W_fc']
         b_fc = parameters['b_fc']
         W_out = parameters['W_out']
         b_out = parameters['b_out']

         # 2. CONV -> RELU -> POOL
         # 卷積層
         # 這裡我們使用固定的 stride=1, padding=1, pool_size=2, pool_stride=2
         Z_conv, cache_conv = conv_forward_fast(X, W_conv, b_conv,
     ↪stride=1, padding=1)
```

```
    # ReLU 激活
    A_conv = relu(Z_conv)

    # 池化層
    A_pool, cache_pool = max_pool_forward_fast(A_conv, pool_size=2,␣
↪stride=2)

    # 3. FLATTEN -> FC -> RELU
    # 扁平化
    N = A_pool.shape[0]
    A_flat = A_pool.reshape(N, -1) # -1 會讓 cupy 自動計算維度

    # 全連接層
    Z_fc = cp.dot(A_flat, W_fc) + b_fc

    # ReLU 激活
    A_fc = relu(Z_fc)

    # 4. OUTPUT LAYER -> SOFTMAX
    Z_out = cp.dot(A_fc, W_out) + b_out
    A_out = softmax(Z_out)

    # 5. 儲存所有快取
    # 這是反向傳播時的關鍵
    cache = {
        "cache_conv": cache_conv, # (A_prev, W, b, stride, padding)
        "Z_conv": Z_conv, # Conv 輸出
        "A_conv": A_conv, # Conv 的 ReLU 輸出
        "cache_pool": cache_pool, # (A_conv, (pool_size, stride))
        "A_pool": A_pool, # Pool 輸出
        "A_flat": A_flat, # Flatten 輸出
        "Z_fc": Z_fc, # FC 輸出
        "A_fc": A_fc, # FC 的 ReLU 輸出
        "W_fc": W_fc, # FC 權重
        "W_out": W_out # Out 權重
    }

    return A_out, cache
```

4.6 Step 2-6: 損失函數 Loss Function

跟 HW3 一樣，使用 Cross Entropy

參數:

- `A_out` – 模型的預測機率輸出, shape (N, num_classes)
- `Y` – 真實的 One-Hot 標籤, shape (N, num_classes)

Return:

- `loss` – 交叉熵損失值 (一個純量)

```
[ ]: def compute_loss(A_out, Y):

         N = Y.shape[0]

         # 加上一個極小值 1e-9 是為了避免 log(0) 的情況
         loss = -cp.sum(Y * cp.log(A_out + 1e-9)) / N

         return loss
```

4.7 Step 2-7: Backward Propagation

路徑如下：

`dLoss/dZ_out` -> `dLoss/dA_fc` -> `dLoss/dZ_fc` -> `dLoss/dA_pool` -> `dLoss/dA_conv` -> `dLoss/dZ_conv`

每一步，我們都會計算出對應層的權重梯度 (`dW`, `db`) 以及傳遞給前一層的梯度 (`dA`)。

對於「Softmax + Cross Entropy」這個經典組合，起始梯度有個很簡潔的結果：

$$\frac{\partial Loss}{\partial_{out}} = A_{out} - Y$$

其中 A_{out} 是 Softmax 的預測機率，Y 是正解的 One-Hot 標籤。這大大簡化了我們的計算。我們就從這個梯度 `dZ_out` 開始。

參數:

- `A_out` – 模型的預測輸出
- `Y` – 真實的 One-Hot 標籤
- `cache` – 前向傳播儲存的所有中間值
- `parameters` – 包含所有權重的字典

Return:

- `grads` – 包含所有權重與 bias 梯度的字典

```
[ ]: def backward_propagation(A_out, Y, cache, parameters):
         grads = {}
         N = Y.shape[0] # 批次大小

         # 1. 輸出層的反向傳播
         # 起始梯度 dZ_out
         dZ_out = A_out - Y

         # 對 W_out, b_out 的梯度
         A_fc = cache['A_fc']
         grads['dW_out'] = (1/N) * cp.dot(A_fc.T, dZ_out)
```

```python
    grads['db_out'] = (1/N) * cp.sum(dZ_out, axis=0, keepdims=True)

    # 傳遞給前一層 (FC 層) 的梯度 dA_fc
    W_out = parameters['W_out']
    dA_fc = cp.dot(dZ_out, W_out.T)

    # 2. 全連接層 (FC) 的反向傳播 ---
    # 首先通過 ReLU 的反向傳播
    Z_fc = cache['Z_fc']
    dZ_fc = relu_backward(dA_fc, Z_fc) # 我們之前寫好的 relu_backward

    # 對 W_fc, b_fc 的梯度
    A_flat = cache['A_flat']
    grads['dW_fc'] = (1/N) * cp.dot(A_flat.T, dZ_fc)
    grads['db_fc'] = (1/N) * cp.sum(dZ_fc, axis=0, keepdims=True)

    # 傳遞給前一層 (扁平層) 的梯度 dA_flat
    W_fc = parameters['W_fc']
    dA_flat = cp.dot(dZ_fc, W_fc.T)

    # 3. 扁平層 (Flatten) 的反向傳播
    # 這一步只是將梯度從 1D 向量恢復成 4D 陣列
    A_pool = cache['A_pool']
    dA_pool = dA_flat.reshape(A_pool.shape)

    # 接下來的兩步 (池化層和卷積層) 比較複雜，我們先把函式框架寫好

    # 4. 池化層 (Pooling) 的反向傳播
    cache_pool = cache['cache_pool']
    dA_conv = max_pool_backward_fast(dA_pool, cache_pool)

    # 5. 卷積層 (Conv) 的反向傳播
    # 首先通過 ReLU 的反向傳播
    Z_conv = cache['Z_conv']
    dZ_conv = relu_backward(dA_conv, Z_conv)

    # 傳入卷積層反向傳播函式
    cache_conv = cache['cache_conv']
    dA_prev, dW_conv, db_conv = conv_backward_fast(dZ_conv, cache_conv)

    grads['dW_conv'] = dW_conv
    grads['db_conv'] = db_conv

    return grads
```

上面的程式碼還無法執行，因為我們還缺少 max_pool_backward 和 conv_backward 這兩個最關鍵的函式。

4.7.1 Step 2-7-2: 池化層的反向傳播

實現最大池化層的反向傳播

參數:

- dA – 池化層輸出的梯度, shape (N, C, H_out, W_out)
- cache – 前向傳播的快取 (A_prev, (pool_size, stride))

Return:

- dA_prev – 傳遞給前一層的梯度

```
[ ]: def max_pool_backward(dA, cache):

    A_prev, (pool_size, stride) = cache
    (N, C, H_prev, W_prev) = A_prev.shape
    (N, C, H_out, W_out) = dA.shape

    dA_prev = cp.zeros_like(A_prev)

    for n in range(N):
        for c in range(C):
            for h in range(H_out):
                for w in range(W_out):
                    vert_start = h * stride
                    vert_end = vert_start + pool_size
                    horiz_start = w * stride
                    horiz_end = horiz_start + pool_size

                    # 選取前向傳播時的同一個窗口
                    a_slice = A_prev[n, c, vert_start:vert_end,␣
 ↪horiz_start:horiz_end]

                    # 找到最大值的位置 (mask)
                    mask = (a_slice == cp.max(a_slice))

                    # 將 dA 的梯度加到這個最大值的位置上
                    dA_prev[n, c, vert_start:vert_end, horiz_start:
 ↪horiz_end] += mask * dA[n, c, h, w]

    return dA_prev
```

4.7.2 Step 2-7-3: 卷積層的反向傳播

實現卷積層的反向傳播

參數:

- dZ – 卷積層輸出的梯度, shape (N, n_filters, H_out, W_out)
- cache – 前向傳播的快取 (A_prev, W, b, stride, padding)

Return:

- dA_prev, dW, db

[ ]:
```
def conv_backward(dZ, cache):
    A_prev, W, b, stride, padding = cache
    (N, C_prev, H_prev, W_prev) = A_prev.shape
    (n_filters, C_prev, f, f) = W.shape
    (N, n_filters, H_out, W_out) = dZ.shape

    # 初始化梯度
    dA_prev = cp.zeros_like(A_prev)
    dW = cp.zeros_like(W)
    db = cp.zeros((n_filters, 1))

    # 對輸入進行補零 (為了計算 dA_prev)
    A_prev_pad = zero_pad(A_prev, padding)
    dA_prev_pad = zero_pad(dA_prev, padding)

    for n in range(N):
        # 選取單一樣本
        a_prev_pad = A_prev_pad[n]
        da_prev_pad = dA_prev_pad[n]

        for c in range(n_filters):
            for h in range(H_out):
                for w in range(W_out):
                    vert_start = h * stride
                    vert_end = vert_start + f
                    horiz_start = w * stride
                    horiz_end = horiz_start + f

                    # 定位滑動窗口
                    a_slice = a_prev_pad[:, vert_start:vert_end, 
 ↪horiz_start:horiz_end]

                    # 計算梯度
                    # 1. 更新傳遞給前一層的梯度 dA_prev
                    da_prev_pad[:, vert_start:vert_end, horiz_start:
 ↪horiz_end] += W[c, :, :, :] * dZ[n, c, h, w]
                    # 2. 更新對濾波器的梯度 dW
                    dW[c, :, :, :] += a_slice * dZ[n, c, h, w]
                    # 3. 更新對偏置的梯度 db (此處先累加)
                    db[c] += dZ[n, c, h, w]

        # 去掉 dA_prev_pad 的補零部分
        if padding > 0:
```

```python
            dA_prev[n, :, :, :] = da_prev_pad[:, padding:-padding,␣
↪padding:-padding]
        else:
            dA_prev[n, :, :, :] = da_prev_pad

    # 對 db 取平均
    db = db / N

    return dA_prev, dW, db
```

4.7.3 Step 2-7-4: 優化卷積層反向傳播

跟前向傳播時一樣，執行時發現巢狀迴圈效率過差

[ ]:
```python
from cupy.lib.stride_tricks import as_strided

def col2im_gpu(cols, x_shape, field_height=3, field_width=3,␣
↪padding=1, stride=1):
    """ An implementation of col2im on GPU """
    N, C, H, W = x_shape
    H_padded, W_padded = H + 2 * padding, W + 2 * padding
    x_padded = cp.zeros((N, C, H_padded, W_padded), dtype=cols.dtype)
    k, i, j = get_im2col_indices(x_shape, field_height, field_width,␣
↪padding, stride)
    cols_reshaped = cols.reshape(C * field_height * field_width, -1, N)
    cols_reshaped = cols_reshaped.transpose(2, 0, 1)

    # 使用 cp.add.at 這個原子操作來高效地將梯度加總回去
    cp.add.at(x_padded, (slice(None), k, i, j), cols_reshaped)

    if padding == 0:
        return x_padded
    return x_padded[:, :, padding:-padding, padding:-padding]
```

[ ]:
```python
def conv_backward_fast(dZ, cache):
    """
    使用 col2im 實現的高效版卷積反向傳播
    """
    # 解開 cache
    A_prev, W, b, stride, padding, A_col = cache
    N, C, H, W_shape = A_prev.shape
    n_filters, _, _, _ = W.shape

    # 計算 db 的梯度
    db = cp.sum(dZ, axis=(0, 2, 3))
    db = db.reshape(n_filters, -1)
```

```
    # 重新塑形 dZ 以便進行矩陣乘法
    dZ_reshaped = dZ.transpose(1, 2, 3, 0).reshape(n_filters, -1)

    # 計算 dW 的梯度
    dW = dZ_reshaped @ A_col.T
    dW = dW.reshape(W.shape)

    # 計算傳遞給前一層的梯度 dA_prev
    W_row = W.reshape(n_filters, -1)
    dA_col = W_row.T @ dZ_reshaped
    dA_prev = col2im_gpu(dA_col, A_prev.shape, W.shape[2], W.shape[3],␣
↪padding, stride)

    return dA_prev, dW, db
```

4.7.4 Step 2-7-5: 優化池化層的反向傳播

[ ]:
```
def max_pool_backward_fast(dA, cache):
    # 從快取中取出前向傳播的輸入與輸出
    A_prev, A_out, (pool_size, stride) = cache

    # 1. 將池化層輸出的梯度 dA 上採樣
    dA_upsampled = dA.repeat(pool_size, axis=2).repeat(pool_size,␣
↪axis=3)

    # 2. 同樣地，將前向傳播的輸出 A_out 上採樣
    A_out_upsampled = A_out.repeat(pool_size, axis=2).
↪repeat(pool_size, axis=3)

    # 3. 建立遮罩：A_prev 中等於上採樣後 A_out 的位置，就是當時的最大值
    # 這些位置的值為 True (1)，其餘為 False (0)
    mask = (A_prev == A_out_upsampled)

    # 4. 將上採樣的梯度應用於遮罩，梯度只會流向最大值所在的位置
    dA_prev = dA_upsampled * mask

    return dA_prev
```

4.8 Step 2-8: 更新權重

使用梯度下降法更新模型的參數

參數:

- `parameters` – 包含當前權重的字典
- `grads` – 包含梯度的字典
- `learning_rate` – 學習率 (alpha)

Return:

- parameters – 更新後的參數字典

```
[ ]: def update_parameters(parameters, grads, learning_rate):

    # 根據梯度下降規則更新每一個參數
    parameters['W_conv'] -= learning_rate * grads['dW_conv']
    parameters['b_conv'] -= learning_rate * grads['db_conv']
    parameters['W_fc'] -= learning_rate * grads['dW_fc']
    parameters['b_fc'] -= learning_rate * grads['db_fc']
    parameters['W_out'] -= learning_rate * grads['dW_out']
    parameters['b_out'] -= learning_rate * grads['db_out']

    return parameters
```

5 Step 3: 整合出 CNN Model

5.1 Step 3-1: Create the CNN Model

完整的 CNN 訓練模型

Return:

- trained_parameters – 訓練完成後的模型參數
- history – 包含每個 epoch 的 cost 和 accuracy 的字典

```
[ ]: import time # 引入 time 模組來計算訓練時間
import copy # 我們需要 copy 模組來 deepcopy 權重

def cnn_model(X_train, Y_train, Y_train_orig, X_test, Y_test_orig,
              learning_rate=0.05, num_epochs=10, batch_size=64,
              n_filters=8, filter_size=3,
              use_early_stopping=True, early_stopping_patience=5):

    (N_train, C_train, H_train, W_train) = X_train.shape
    num_classes = Y_train.shape[1]

    # 1. 初始化權重
    parameters = initialize_parameters(input_dim=(C_train, H_train,␣
↪W_train),
                                       n_filters=n_filters,␣
↪filter_size=filter_size,
                                       n_y=num_classes)

    # 儲存每個 epoch 的 loss 和 accuracy，用於後續繪圖
    costs = []
    accuracies = []

    # === 早停機制初始化 ===
```

```
    if use_early_stopping:
        patience = early_stopping_patience
        epochs_no_improve = 0
        best_accuracy = 0.0
        # 用來儲存性能最好時的權重
        best_parameters = {}

    # 2. 訓練迴圈
    for i in range(num_epochs):
        epoch_start_time = time.time()
        epoch_cost = 0.0

        # 在每個 epoch 開始前，將資料隨機打亂
        permutation = cp.random.permutation(N_train)
        shuffled_X = X_train[permutation, :, :, :]
        shuffled_Y = Y_train[permutation, :]

        # Mini-batch 處理
        num_minibatches = N_train // batch_size

        for j in range(num_minibatches):
            start = j * batch_size
            end = start + batch_size
            minibatch_X = shuffled_X[start:end, :, :, :]
            minibatch_Y = shuffled_Y[start:end, :]

            # a. 前向傳播
            A_out, cache = forward_propagation(minibatch_X, parameters)

            # b. 計算損失
            cost = compute_loss(A_out, minibatch_Y)
            epoch_cost += cost

            # c. 反向傳播
            grads = backward_propagation(A_out, minibatch_Y, cache,␣
↪parameters)

            # d. 更新參數
            parameters = update_parameters(parameters, grads,␣
↪learning_rate)

        # 計算並記錄這個 epoch 的平均損失
        avg_epoch_cost = epoch_cost / num_minibatches
        costs.append(avg_epoch_cost)

        # 在每個 epoch 結束後，用測試集評估準確率
```

```
        predictions = predict(X_test, parameters)
        current_accuracy = calculate_accuracy(predictions, Y_test_orig)
        accuracies.append(current_accuracy)

        epoch_end_time = time.time()

        print(f"Epoch {i + 1}/{num_epochs} - Cost: {avg_epoch_cost:.
↪6f} - Accuracy: {current_accuracy:.4f} - Time: {epoch_end_time -␣
↪epoch_start_time:.2f}s")

        # 早停機制
        if use_early_stopping:
            if current_accuracy > best_accuracy:
                best_accuracy = current_accuracy
                epochs_no_improve = 0
                # deepcopy 目前最好的權重
                best_parameters = copy.deepcopy(parameters)
                print(f"  -> Accuracy improved to {best_accuracy:.4f}!␣
↪Saving model.")
            else:
                epochs_no_improve += 1
                print(f"  -> Accuracy did not improve for␣
↪{epochs_no_improve} epoch(s).")

            if epochs_no_improve >= patience:
                print(f"\nEarly stopping triggered after {patience}␣
↪epochs with no improvement.")
                # 訓練結束，回傳歷史最佳的權重
                parameters = best_parameters
                break

    history = {"costs": costs, "accuracies": accuracies}
    return parameters, history
```

[ ]:
```
# --- 預測與準確率計算的輔助函式 ---
def predict(X, parameters):
    A_out, _ = forward_propagation(X, parameters)
    # 找出每個樣本中，機率最高的那個類別的索引 (0-9)
    predictions = cp.argmax(A_out, axis=1)
    return predictions

def calculate_accuracy(predictions, labels):
    # 比較預測結果和正解標籤
    return cp.mean(predictions == labels)
```

5.2 Step 3-2: 訓練模型

[ ]:
```
# 設定超參數
LEARNING_RATE = 0.05
NUM_EPOCHS = 100
BATCH_SIZE = 64
N_FILTERS = 32
FILTER_SIZE = 3
PATIENCE = 10

# 開始訓練
trained_parameters, history = cnn_model(
    train_x, train_y, train_y_orig,
    test_x, test_y_orig,
    learning_rate=LEARNING_RATE,
    num_epochs=NUM_EPOCHS,
    batch_size=BATCH_SIZE,
    n_filters=N_FILTERS,
    filter_size=FILTER_SIZE,
    use_early_stopping=True,
    # 如果連續 PATIENCE 次 accuracy 沒有提升就停止
    early_stopping_patience=PATIENCE
)
```

```
Epoch 1/100 - Cost: 0.402958 - Accuracy: 0.9508 - Time: 19.36s
  -> Accuracy improved to 0.9508! Saving model.
Epoch 2/100 - Cost: 0.134227 - Accuracy: 0.9645 - Time: 8.23s
  -> Accuracy improved to 0.9645! Saving model.
Epoch 3/100 - Cost: 0.098766 - Accuracy: 0.9617 - Time: 8.31s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 4/100 - Cost: 0.076742 - Accuracy: 0.9744 - Time: 8.39s
  -> Accuracy improved to 0.9744! Saving model.
Epoch 5/100 - Cost: 0.058125 - Accuracy: 0.9736 - Time: 8.42s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 6/100 - Cost: 0.046284 - Accuracy: 0.9814 - Time: 8.45s
  -> Accuracy improved to 0.9814! Saving model.
Epoch 7/100 - Cost: 0.038609 - Accuracy: 0.9812 - Time: 8.55s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 8/100 - Cost: 0.031190 - Accuracy: 0.9822 - Time: 8.56s
  -> Accuracy improved to 0.9822! Saving model.
Epoch 9/100 - Cost: 0.024794 - Accuracy: 0.9818 - Time: 8.52s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 10/100 - Cost: 0.020114 - Accuracy: 0.9840 - Time: 8.47s
  -> Accuracy improved to 0.9840! Saving model.
Epoch 11/100 - Cost: 0.016269 - Accuracy: 0.9814 - Time: 8.94s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 12/100 - Cost: 0.014241 - Accuracy: 0.9848 - Time: 8.44s
  -> Accuracy improved to 0.9848! Saving model.
```

```
Epoch 13/100 - Cost: 0.010336 - Accuracy: 0.9839 - Time: 8.39s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 14/100 - Cost: 0.008706 - Accuracy: 0.9843 - Time: 8.44s
  -> Accuracy did not improve for 2 epoch(s).
Epoch 15/100 - Cost: 0.005845 - Accuracy: 0.9854 - Time: 8.45s
  -> Accuracy improved to 0.9854! Saving model.
Epoch 16/100 - Cost: 0.004464 - Accuracy: 0.9826 - Time: 8.45s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 17/100 - Cost: 0.006639 - Accuracy: 0.9852 - Time: 8.43s
  -> Accuracy did not improve for 2 epoch(s).
Epoch 18/100 - Cost: 0.004916 - Accuracy: 0.9826 - Time: 8.48s
  -> Accuracy did not improve for 3 epoch(s).
Epoch 19/100 - Cost: 0.003173 - Accuracy: 0.9854 - Time: 8.46s
  -> Accuracy did not improve for 4 epoch(s).
Epoch 20/100 - Cost: 0.001940 - Accuracy: 0.9843 - Time: 8.43s
  -> Accuracy did not improve for 5 epoch(s).
Epoch 21/100 - Cost: 0.001605 - Accuracy: 0.9855 - Time: 8.46s
  -> Accuracy improved to 0.9855! Saving model.
Epoch 22/100 - Cost: 0.000699 - Accuracy: 0.9855 - Time: 9.67s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 23/100 - Cost: 0.000546 - Accuracy: 0.9853 - Time: 9.06s
  -> Accuracy did not improve for 2 epoch(s).
Epoch 24/100 - Cost: 0.000499 - Accuracy: 0.9859 - Time: 8.41s
  -> Accuracy improved to 0.9859! Saving model.
Epoch 25/100 - Cost: 0.000452 - Accuracy: 0.9850 - Time: 8.43s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 26/100 - Cost: 0.000570 - Accuracy: 0.9856 - Time: 8.44s
  -> Accuracy did not improve for 2 epoch(s).
Epoch 27/100 - Cost: 0.000286 - Accuracy: 0.9859 - Time: 8.50s
  -> Accuracy did not improve for 3 epoch(s).
Epoch 28/100 - Cost: 0.000190 - Accuracy: 0.9860 - Time: 8.41s
  -> Accuracy improved to 0.9860! Saving model.
Epoch 29/100 - Cost: 0.000160 - Accuracy: 0.9857 - Time: 8.70s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 30/100 - Cost: 0.000152 - Accuracy: 0.9854 - Time: 8.44s
  -> Accuracy did not improve for 2 epoch(s).
Epoch 31/100 - Cost: 0.000148 - Accuracy: 0.9856 - Time: 8.46s
  -> Accuracy did not improve for 3 epoch(s).
Epoch 32/100 - Cost: 0.000131 - Accuracy: 0.9860 - Time: 8.42s
  -> Accuracy did not improve for 4 epoch(s).
Epoch 33/100 - Cost: 0.000127 - Accuracy: 0.9860 - Time: 8.48s
  -> Accuracy did not improve for 5 epoch(s).
Epoch 34/100 - Cost: 0.000113 - Accuracy: 0.9856 - Time: 8.53s
  -> Accuracy did not improve for 6 epoch(s).
Epoch 35/100 - Cost: 0.000103 - Accuracy: 0.9858 - Time: 8.43s
  -> Accuracy did not improve for 7 epoch(s).
Epoch 36/100 - Cost: 0.000099 - Accuracy: 0.9857 - Time: 8.49s
  -> Accuracy did not improve for 8 epoch(s).
```

```
Epoch 37/100 - Cost: 0.000095 - Accuracy: 0.9856 - Time: 8.48s
  -> Accuracy did not improve for 9 epoch(s).
Epoch 38/100 - Cost: 0.000090 - Accuracy: 0.9854 - Time: 8.46s
  -> Accuracy did not improve for 10 epoch(s).

Early stopping triggered after 10 epochs with no improvement.
```

5.3 Step 3-3: 儲存及載入參數的函數

儲存模型的權重參數

參數:

- parameters: 包含 CuPy 陣列的權重字典
- filename: 要儲存的檔案名稱

[ ]:
```python
def save_parameters(parameters, filename="my_cnn_model.npz"):
    print(f"Saving parameters to {filename}...")

    # 建立一個新的字典，用來存放轉換後的 NumPy 陣列
    numpy_params = {}
    for key, value in parameters.items():
        # 使用 cp.asnumpy() 將 CuPy 陣列轉為 NumPy 陣列
        numpy_params[key] = cp.asnumpy(value)

    # 使用 np.savez_compressed 儲存，**numpy_params 會自動解包成關鍵字參數
    np.savez_compressed(filename, **numpy_params)

    print("Parameters saved successfully.")
```

[2]:
```python
import os

def load_parameters(filename="my_cnn_model.npz"):
    if not os.path.exists(filename):
        print(f"Error: File '{filename}' not found.")
        return None

    print(f"Loading parameters from {filename}...")

    # 載入 .npz 檔案，它是一個類字典物件
    npzfile = np.load(filename)

    # 建立一個新的字典，用來存放轉換後的 CuPy 陣列
    parameters = {}
    for key in npzfile.keys():
        # 使用 cp.asarray() 將 NumPy 陣列轉為 CuPy 陣列
        parameters[key] = cp.asarray(npzfile[key])

    print("Parameters loaded successfully.")
```

```
    return parameters
```

```
[ ]: save_parameters(trained_parameters, "my_cnn_model.npz")
```

```
Saving parameters to my_cnn_model.npz…
Parameters saved successfully.
```

6 Step 4: 最終評估與報告

6.1 Step 4-1: 最終預測

```
[ ]: trained_parameters = load_parameters("my_cnn_model.npz")
```

```
Loading parameters from my_cnn_model.npz…
Parameters loaded successfully.
```

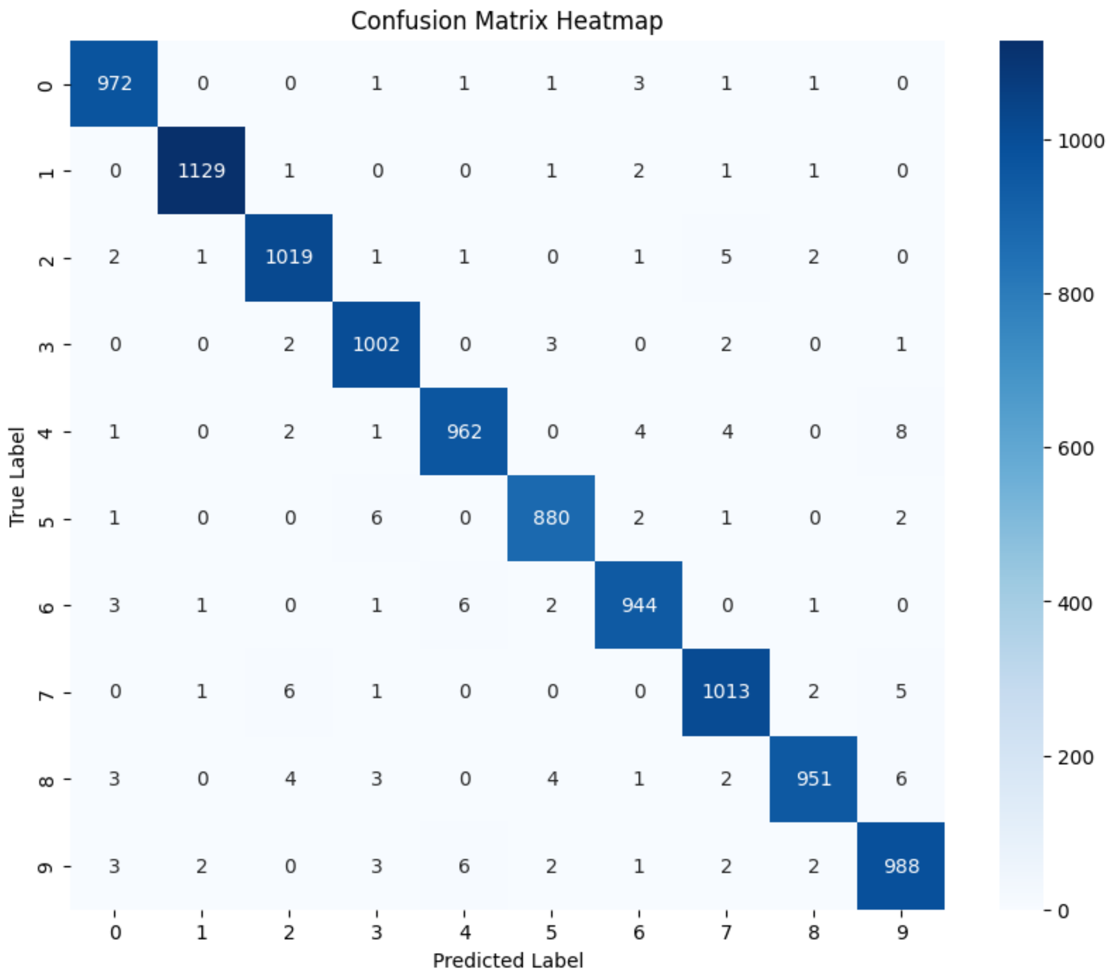
```
[ ]: # 1. 進行最終預測
print("Making final predictions on the test set...")
test_predictions = predict(test_x, trained_parameters)

# 2. 確認最終準確率
final_accuracy = calculate_accuracy(test_predictions, test_y_orig)
# 使用 cp.asnumpy() 將 CuPy 純量轉為 Python 純量以便格式化輸出
print(f"Final model accuracy: {cp.asnumpy(final_accuracy)*100:.2f}%")
```

```
Making final predictions on the test set…
Final model accuracy: 98.60%
```

```
[ ]: # 3. 建立混淆矩陣
num_classes = 10
# 初始化一個 10x10 的 CuPy 矩陣
confusion_matrix_cp = cp.zeros((num_classes, num_classes), dtype=int)

# 遍歷所有測試集樣本，填充混淆矩陣
for i in range(len(test_y_orig)):
    true_label = test_y_orig[i]
    predicted_label = test_predictions[i]
    confusion_matrix_cp[true_label, predicted_label] += 1
```

```
[ ]: # 4. 視覺化混淆矩陣
# 記得要先用 .asnumpy() 將矩陣從 GPU 移回 CPU
confusion_matrix_np = cp.asnumpy(confusion_matrix_cp)

import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 8))
sns.heatmap(
```

```
    confusion_matrix_np,
    annot=True,        # 在格子中顯示數字
    fmt="d",           # 將數字格式化為整數
    cmap="Blues",      # 使用藍色系的色盤
    xticklabels=range(num_classes),
    yticklabels=range(num_classes),
)
plt.title("Confusion Matrix Heatmap")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```



6.2 Step 4-2: 計算精確率 (Precision) 與召回率 (Recall)

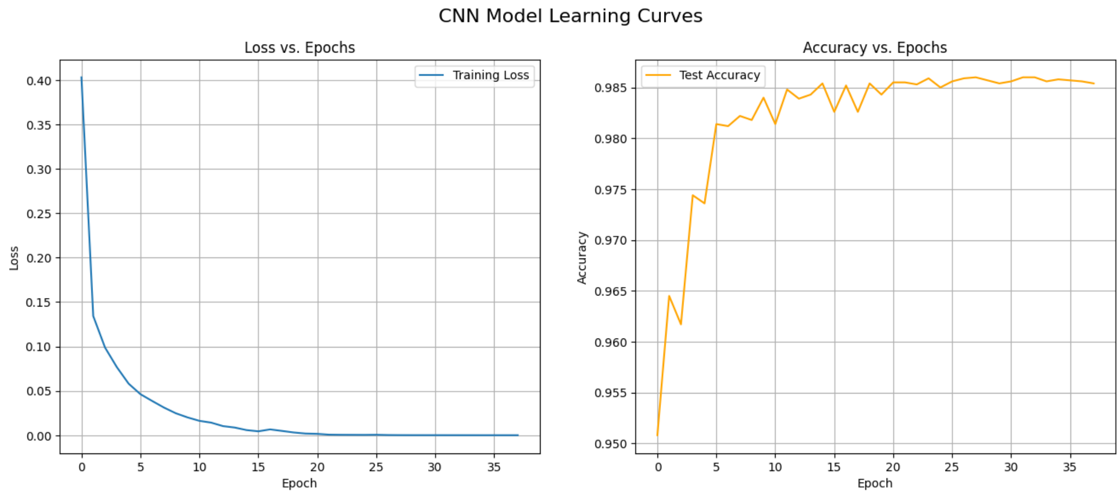
```
[ ]: # 我們直接在 CuPy 陣列上進行計算
print("--- Performance Metrics per Class ---")
for i in range(num_classes):
    true_positives = confusion_matrix_cp[i, i]
    # 該 Col 的總和 (所有被預測為 i 的) - TP
    false_positives = cp.sum(confusion_matrix_cp[:, i]) -␣
  ↪true_positives
    # 該 Row 的總和 (所有實際上是 i 的) - TP
    false_negatives = cp.sum(confusion_matrix_cp[i, :]) -␣
  ↪true_positives

    # 計算 Precision 和 Recall
    precision = true_positives / (true_positives + false_positives)
    recall = true_positives / (true_positives + false_negatives)

    print(f"數字 '{i}':")
    # 使用 .asnumpy() 以便格式化輸出
    print(f"  精確率 (Precision): {cp.asnumpy(precision):.4f}")
    print(f"  召回率 (Recall):    {cp.asnumpy(recall):.4f}")
```

```
--- Performance Metrics per Class ---
數字 '0':
  精確率 (Precision): 0.9868
  召回率 (Recall):    0.9918
數字 '1':
  精確率 (Precision): 0.9956
  召回率 (Recall):    0.9947
數字 '2':
  精確率 (Precision): 0.9855
  召回率 (Recall):    0.9874
數字 '3':
  精確率 (Precision): 0.9833
  召回率 (Recall):    0.9921
數字 '4':
  精確率 (Precision): 0.9857
  召回率 (Recall):    0.9796
數字 '5':
  精確率 (Precision): 0.9854
  召回率 (Recall):    0.9865
數字 '6':
  精確率 (Precision): 0.9854
  召回率 (Recall):    0.9854
數字 '7':
  精確率 (Precision): 0.9825
  召回率 (Recall):    0.9854
數字 '8':
```

```
  精確率 (Precision): 0.9906
  召回率 (Recall):    0.9764
數字 '9':
  精確率 (Precision): 0.9782
  召回率 (Recall):    0.9792
```

6.3 Step 4-3: 繪制曲線圖

[ ]:
```
# 將 CuPy 陣列轉為 NumPy 陣列
# 1. 從 history 中取出資料
costs_cp = cp.array(history['costs'])
accuracies_cp = cp.array(history['accuracies'])

# 2. 使用 asnumpy() 轉換
costs_np = cp.asnumpy(costs_cp)
accuracies_np = cp.asnumpy(accuracies_cp)


# 開始繪圖
# 建立一個 1x2 的子圖畫布，方便同時顯示兩張圖
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

# 設定一個主標題
fig.suptitle('CNN Model Learning Curves', fontsize=16)

# 圖一：損失 (Loss) vs. 世代 (Epochs)
ax1.plot(costs_np, label='Training Loss')
ax1.set_title("Loss vs. Epochs")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Loss")
ax1.grid(True)
ax1.legend()

# 圖二：準確率 (Accuracy) vs. 世代 (Epochs)
ax2.plot(accuracies_np, label='Test Accuracy', color='orange')
ax2.set_title("Accuracy vs. Epochs")
ax2.set_xlabel("Epoch")
ax2.set_ylabel("Accuracy")
ax2.grid(True)
ax2.legend()

# 顯示圖表
plt.show()
```

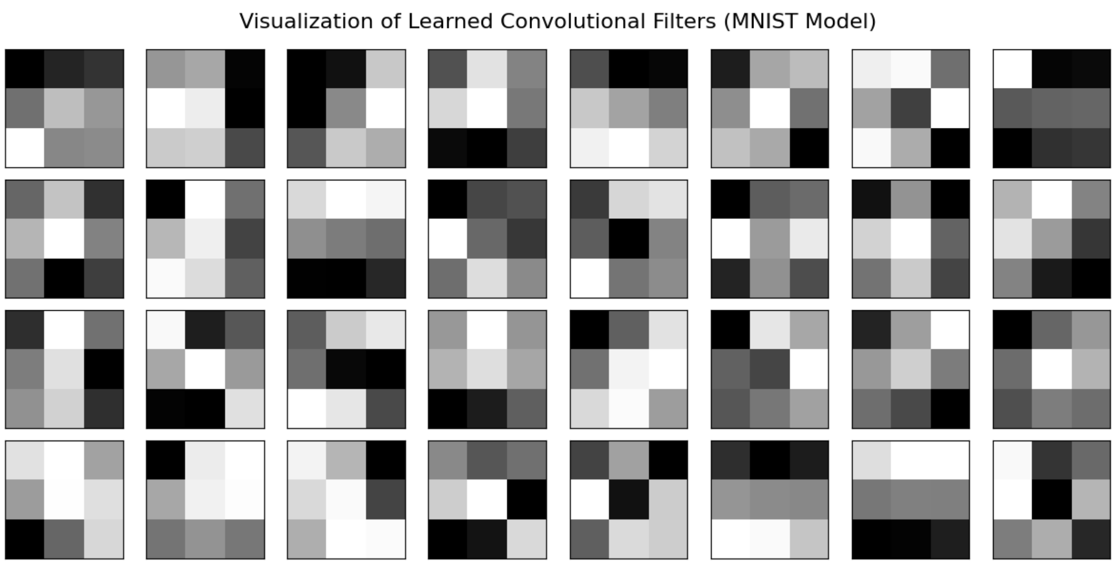


6.4 Step 4-4: 加分：視覺化卷積核 (Visualize Filters)

```
[ ]: trained_parameters = load_parameters("my_cnn_model.npz")
assert trained_parameters is not None
if 'trained_parameters' in locals() or 'trained_parameters' in␣
↪globals():
    print("讀取已訓練的 MNIST 模型權重...")

    # 1. 從權重字典中取出第一個卷積層的權重 (W_conv)
    # 它的 shape 是 (n_filters, C_in, f, f)，在此為 (32, 1, 3, 3)
    W_conv_cp = trained_parameters['W_conv']

    # 2. 將 CuPy 陣列移至 CPU 並轉為 NumPy 陣列，以便 Matplotlib 處理
    W_conv_np = cp.asnumpy(W_conv_cp)

    # 3. 設定視覺化圖表
    # 我們有 32 個 filters，可以用一個 4x8 的網格來顯示
    num_filters = W_conv_np.shape[0]
    fig, axes = plt.subplots(4, 8, figsize=(12, 6))

    # 設定主標題
    fig.suptitle('Visualization of Learned Convolutional Filters␣
↪(MNIST Model)', fontsize=16)

    # 4. 遍歷所有 filters 並在子圖中繪製
    for i, ax in enumerate(axes.flat):
        # 檢查是否還有 filter 可以顯示
        if i < num_filters:
            # 取出第 i 個 filter 的權重。
            # 由於 MNIST 是灰階，輸入頻道只有 1，所以我們用 [i, 0, :, :]
```

```
            filt = W_conv_np[i, 0, :, :]

            # 使用 imshow 繪製 filter，'gray' 色彩映射適合顯示權重
            ax.imshow(filt, cmap='gray')

        # 隱藏座標軸刻度，讓圖表更簡潔
        ax.set_xticks([])
        ax.set_yticks([])

    # 調整子圖之間的間距
    plt.tight_layout()
    # 顯示圖表
    plt.show()

else:
    print("錯誤：找不到 'trained_parameters'。請先確定您已成功訓練並載入␣
↪MNIST 模型。")
```

```
Loading parameters from my_cnn_model.npz…
Parameters loaded successfully.
讀取已訓練的 MNIST 模型權重…
```



7 Bonus: CIFAR-10

[ ]:
```
import gc # 引入垃圾回收模組

# 刪除所有不再需要的 MNIST 變數
mnist_vars_to_delete = [
```

```
    'train_x', 'train_y', 'train_x_orig', 'train_y_orig',
    'test_x', 'test_y', 'test_x_orig', 'test_y_orig',
    'trained_parameters',
    'test_predictions',
    'confusion_matrix_cp'
]

for var_name in mnist_vars_to_delete:
    if var_name in locals() or var_name in globals():
        del globals()[var_name]
        print(f"變數 '{var_name}' 已刪除。")

# 強制進行垃圾回收
gc.collect()

# 清空 CuPy 的記憶體池
mempool = cp.get_default_memory_pool()
mempool.free_all_blocks()

print("\nCuPy 記憶體池已清空。")
print(f"目前 VRAM 已分配 (Used Bytes): {mempool.used_bytes()} bytes")
print(f"目前 VRAM 總量 (Total Bytes): {mempool.total_bytes()} bytes")
```

```
變數 'train_x' 已刪除。
變數 'train_y' 已刪除。
變數 'train_x_orig' 已刪除。
變數 'train_y_orig' 已刪除。
變數 'test_x' 已刪除。
變數 'test_y' 已刪除。
變數 'test_x_orig' 已刪除。
變數 'test_y_orig' 已刪除。
變數 'trained_parameters' 已刪除。
變數 'test_predictions' 已刪除。
變數 'confusion_matrix_cp' 已刪除。

CuPy 記憶體池已清空。
目前 VRAM 已分配 (Used Bytes): 1670149120 bytes
目前 VRAM 總量 (Total Bytes): 3036867584 bytes
```

7.1 Bonus Step 1: 下載並加載 CIFAR-10

[ ]:
```
!wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
```

```
--2025-06-21 15:19:58--  https://www.cs.toronto.edu/~kriz/
↪cifar-10-python.tar.gz
Resolving www.cs.toronto.edu (www.cs.toronto.edu)… 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443…
connected.
```

```
HTTP request sent, awaiting response… 200 OK
Length: 170498071 (163M) [application/x-gzip]
Saving to:  'cifar-10-python.tar.gz'

cifar-10-python.tar 100%[===================>] 162.60M  1.05MB/s    in
↪2m 53s

2025-06-21 15:22:53 (961 KB/s) -  'cifar-10-python.tar.gz'  saved
[170498071/170498071]
```

[ ]:
```
!tar zxvf cifar-10-python.tar.gz
```

```
cifar-10-batches-py/
cifar-10-batches-py/data_batch_4
cifar-10-batches-py/readme.html
cifar-10-batches-py/test_batch
cifar-10-batches-py/data_batch_3
cifar-10-batches-py/batches.meta
cifar-10-batches-py/data_batch_2
cifar-10-batches-py/data_batch_5
cifar-10-batches-py/data_batch_1
```

[ ]:
```
import pickle
import os
```

[ ]:
```
def load_cifa10(root):
  def load_batch(filename):
      with open(os.path.join(root, filename), 'rb') as f:
          # 使用 latin1 編碼來讀取 Python 2 生成的 pickle 檔案
          datadict = pickle.load(f, encoding='latin1')
          X = datadict['data']
          Y = datadict['labels']
          # 將資料塑形為 (樣本數, 色彩頻道, 高度, 寬度)
          # 原始資料格式為 (N, 3072)，其中 3072 = 3 * 32 * 32
          # 儲存順序是 RRR...GGG...BBB...
          X = X.reshape(10000, 3, 32, 32)
          Y = np.array(Y)
          return X, Y

  # 載入訓練資料 (共 5 個 batch)
  xs, ys = [], []
  for i in range(1, 6):
      x, y = load_batch(f"data_batch_{i}")
      xs.append(x)
      ys.append(y)

  train_x = np.concatenate(xs)
```

```
    train_y = np.concatenate(ys)

    # 載入測試資料
    test_x, test_y = load_batch("test_batch")

    # 載入類別名稱
    with open(os.path.join(root, 'batches.meta'), 'rb') as f:
        meta = pickle.load(f, encoding='latin1')
        class_names = meta['label_names']

    return train_x, train_y, test_x, test_y, class_names
```

[ ]:
```
cifar10_train_x_np, cifar10_train_y_np, cifar10_test_x_np,␣
  ↪cifar10_test_y_np, cifar10_class_names =␣
  ↪load_cifa10("cifar-10-batches-py")
print("\n--- CIFAR-10 資料維度 (NumPy) ---")
print("訓練資料 X:", cifar10_train_x_np.shape)
print("訓練標籤 Y:", cifar10_train_y_np.shape)
print("測試資料 X:", cifar10_test_x_np.shape)
print("測試標籤 Y:", cifar10_test_y_np.shape)
print("類別名稱:", cifar10_class_names)
```

```
--- CIFAR-10 資料維度 (NumPy) ---
訓練資料 X: (50000, 3, 32, 32)
訓練標籤 Y: (50000,)
測試資料 X: (10000, 3, 32, 32)
測試標籤 Y: (10000,)
類別名稱: ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',␣
 ↪'frog', 'horse',
'ship', 'truck']
```

7.2 Bonus Step 2: 轉為 CuPy

[ ]:
```
# 將 NumPy 陣列轉換為 CuPy 陣列
cifar10_train_x_orig = cp.asarray(cifar10_train_x_np)
cifar10_train_y_orig = cp.asarray(cifar10_train_y_np)
cifar10_test_x_orig = cp.asarray(cifar10_test_x_np)
cifar10_test_y_orig = cp.asarray(cifar10_test_y_np)

# --- 正規化 (Normalize) ---
# 將像素值從 0-255 正規化到 0-1 之間
cifar10_train_x = cifar10_train_x_orig / 255.0
cifar10_test_x = cifar10_test_x_orig / 255.0

# --- 對 Label 進行 One-Hot Encode ---
num_classes_cifar10 = 10
```

```
cifar10_train_y = one_hot_encode(cifar10_train_y_orig,␣
  ↪num_classes_cifar10)
cifar10_test_y = one_hot_encode(cifar10_test_y_orig,␣
  ↪num_classes_cifar10)

print("\n--- CIFAR-10 資料維度 (CuPy) ---")
print("train_x shape:", cifar10_train_x.shape)
print("train_y shape:", cifar10_train_y.shape)
print("test_x shape:", cifar10_test_x.shape)
print("test_y shape:", cifar10_test_y.shape)
print("Data type:", type(cifar10_train_x))
```

```
--- CIFAR-10 資料維度 (CuPy) ---
train_x shape: (50000, 3, 32, 32)
train_y shape: (50000, 10)
test_x shape: (10000, 3, 32, 32)
test_y shape: (10000, 10)
Data type: <class 'cupy.ndarray'>
```

7.3 Bonus Step 3: 訓練

[ ]:
```
# 設定 hyperparameters
LEARNING_RATE_CIFAR = 0.05
NUM_EPOCHS_CIFAR = 150 # 增加 Epoch 上限
BATCH_SIZE_CIFAR = 128
N_FILTERS_CIFAR = 32 # 增加 filter 數量以捕捉更複雜的特徵
FILTER_SIZE_CIFAR = 3
PATIENCE_CIFAR = 15

# 開始訓練
cifar10_params, cifar10_history = cnn_model(
    cifar10_train_x, cifar10_train_y, cifar10_train_y_orig,
    cifar10_test_x, cifar10_test_y_orig,
    learning_rate=LEARNING_RATE_CIFAR,
    num_epochs=NUM_EPOCHS_CIFAR,
    batch_size=BATCH_SIZE_CIFAR,
    n_filters=N_FILTERS_CIFAR,
    filter_size=FILTER_SIZE_CIFAR,
    use_early_stopping=True,
    early_stopping_patience=PATIENCE_CIFAR
)

# 儲存訓練好的 CIFAR-10 模型權重
save_parameters(cifar10_params, "my_cifar10_cnn_model.npz")
```

```
Epoch 1/150 - Cost: 2.072176 - Accuracy: 0.3469 - Time: 10.08s
  -> Accuracy improved to 0.3469! Saving model.
```

```
Epoch 2/150 - Cost: 1.753204 - Accuracy: 0.4441 - Time: 9.98s
  -> Accuracy improved to 0.4441! Saving model.
Epoch 3/150 - Cost: 1.541613 - Accuracy: 0.4770 - Time: 9.69s
  -> Accuracy improved to 0.4770! Saving model.
Epoch 4/150 - Cost: 1.422470 - Accuracy: 0.5105 - Time: 9.27s
  -> Accuracy improved to 0.5105! Saving model.
Epoch 5/150 - Cost: 1.330337 - Accuracy: 0.5092 - Time: 9.34s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 6/150 - Cost: 1.248691 - Accuracy: 0.5237 - Time: 9.18s
  -> Accuracy improved to 0.5237! Saving model.
Epoch 7/150 - Cost: 1.158501 - Accuracy: 0.5591 - Time: 9.20s
  -> Accuracy improved to 0.5591! Saving model.
Epoch 8/150 - Cost: 1.072573 - Accuracy: 0.5901 - Time: 9.22s
  -> Accuracy improved to 0.5901! Saving model.
Epoch 9/150 - Cost: 1.013198 - Accuracy: 0.5719 - Time: 9.25s
  -> Accuracy did not improve for 1 epoch(s).
Epoch 10/150 - Cost: 0.944731 - Accuracy: 0.5490 - Time: 9.27s
  -> Accuracy did not improve for 2 epoch(s).
Epoch 11/150 - Cost: 0.883016 - Accuracy: 0.5827 - Time: 9.27s
  -> Accuracy did not improve for 3 epoch(s).
Epoch 12/150 - Cost: 0.830277 - Accuracy: 0.5769 - Time: 9.28s
  -> Accuracy did not improve for 4 epoch(s).
Epoch 13/150 - Cost: 0.776206 - Accuracy: 0.5644 - Time: 9.26s
  -> Accuracy did not improve for 5 epoch(s).
Epoch 14/150 - Cost: 0.728672 - Accuracy: 0.5807 - Time: 9.26s
  -> Accuracy did not improve for 6 epoch(s).
Epoch 15/150 - Cost: 0.684573 - Accuracy: 0.5901 - Time: 9.24s
  -> Accuracy did not improve for 7 epoch(s).
Epoch 16/150 - Cost: 0.624721 - Accuracy: 0.5801 - Time: 9.23s
  -> Accuracy did not improve for 8 epoch(s).
Epoch 17/150 - Cost: 0.587469 - Accuracy: 0.5757 - Time: 9.23s
  -> Accuracy did not improve for 9 epoch(s).
Epoch 18/150 - Cost: 0.542030 - Accuracy: 0.5882 - Time: 9.25s
  -> Accuracy did not improve for 10 epoch(s).
Epoch 19/150 - Cost: 0.508496 - Accuracy: 0.5852 - Time: 9.34s
  -> Accuracy did not improve for 11 epoch(s).
Epoch 20/150 - Cost: 0.482322 - Accuracy: 0.5689 - Time: 9.25s
  -> Accuracy did not improve for 12 epoch(s).
Epoch 21/150 - Cost: 0.442902 - Accuracy: 0.5829 - Time: 9.27s
  -> Accuracy did not improve for 13 epoch(s).
Epoch 22/150 - Cost: 0.412536 - Accuracy: 0.5788 - Time: 9.26s
  -> Accuracy did not improve for 14 epoch(s).
Epoch 23/150 - Cost: 0.372546 - Accuracy: 0.5764 - Time: 10.03s
  -> Accuracy did not improve for 15 epoch(s).

Early stopping triggered after 15 epochs with no improvement.
Saving parameters to my_cifar10_cnn_model.npz…
Parameters saved successfully.
```

7.4 Bonus Step 4: 效能評估

[ ]:
```
cifar10_trained_parameters = load_parameters("my_cifar10_cnn_model.
  ↪npz")
```

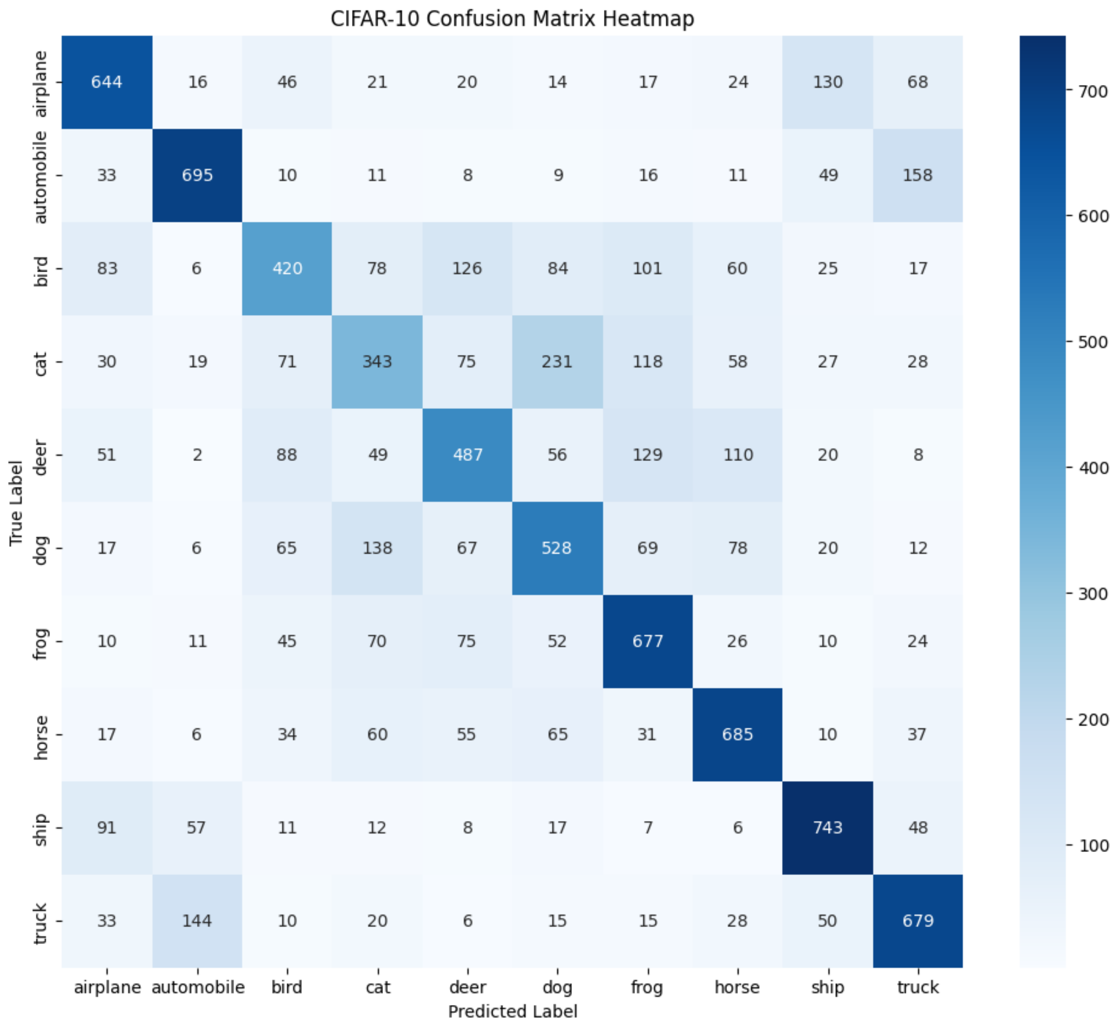
```
Loading parameters from my_cifar10_cnn_model.npz…
Parameters loaded successfully.
```

[ ]:
```
# 1. 進行最終預測
print("在 CIFAR-10 測試集上進行最終預測")
cifar10_test_predictions = predict(cifar10_test_x,␣
  ↪cifar10_trained_parameters)

# 2. 確認最終準確率
final_accuracy_cifar10 = calculate_accuracy(cifar10_test_predictions,␣
  ↪cifar10_test_y_orig)
print(f"CIFAR-10 模型最終準確率: {cp.asnumpy(final_accuracy_cifar10)*100:.
  ↪2f}%")
```

```
在 CIFAR-10 測試集上進行最終預測
CIFAR-10 模型最終準確率: 59.01%
```

[ ]:
```
# 3. 建立混淆矩陣
confusion_matrix_cifar10_cp = cp.zeros((num_classes_cifar10,␣
  ↪num_classes_cifar10), dtype=int)
for i in range(len(cifar10_test_y_orig)):
    true_label = cifar10_test_y_orig[i]
    predicted_label = cifar10_test_predictions[i]
    confusion_matrix_cifar10_cp[true_label, predicted_label] += 1

# 4. 視覺化混淆矩陣 (使用類別名稱)
confusion_matrix_cifar10_np = cp.asnumpy(confusion_matrix_cifar10_cp)

import seaborn as sns
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 10))
sns.heatmap(
    confusion_matrix_cifar10_np,
    annot=True,
    fmt="d",
    cmap="Blues",
    xticklabels=cifar10_class_names,
    yticklabels=cifar10_class_names,
)
plt.title("CIFAR-10 Confusion Matrix Heatmap")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
```

```
plt.show()
```



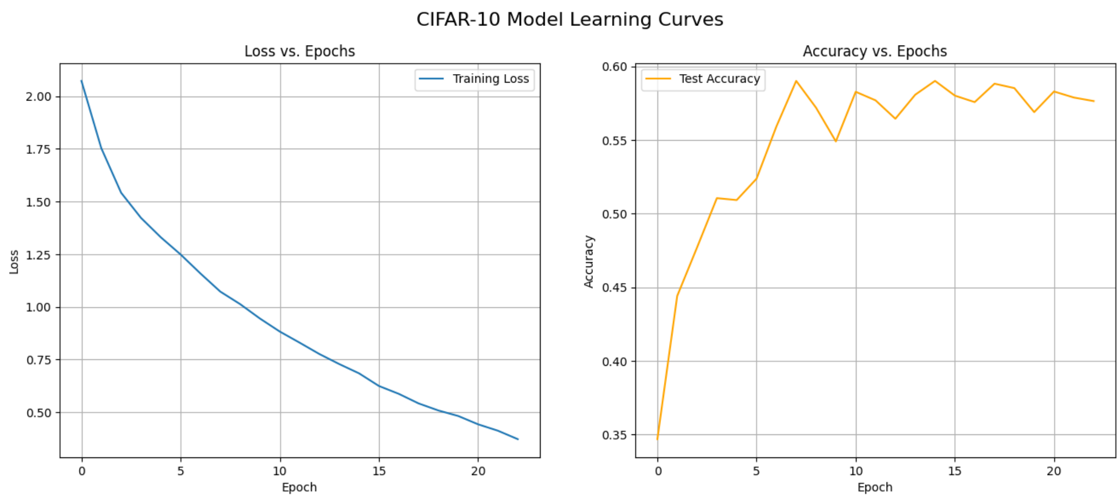
[ ]:
```
# 5. 計算精確率與召回率
print("\n--- CIFAR-10 各類別的性能指標 ---")
for i in range(num_classes_cifar10):
    true_positives = confusion_matrix_cifar10_cp[i, i]
    false_positives = cp.sum(confusion_matrix_cifar10_cp[:, i]) - true_positives
    false_negatives = cp.sum(confusion_matrix_cifar10_cp[i, :]) - true_positives

    precision = true_positives / (true_positives + false_positives + 1e-9) # 避免除以零
    recall = true_positives / (true_positives + false_negatives + 1e-9)
```

```
    print(f"類別 '{cifar10_class_names[i]}':")
    print(f"  精確率 (Precision): {cp.asnumpy(precision):.4f}")
    print(f"  召回率 (Recall):    {cp.asnumpy(recall):.4f}")
```

```
--- CIFAR-10 各類別的性能指標 ---
類別 'airplane':
  精確率 (Precision): 0.6383
  召回率 (Recall):    0.6440
類別 'automobile':
  精確率 (Precision): 0.7225
  召回率 (Recall):    0.6950
類別 'bird':
  精確率 (Precision): 0.5250
  召回率 (Recall):    0.4200
類別 'cat':
  精確率 (Precision): 0.4277
  召回率 (Recall):    0.3430
類別 'deer':
  精確率 (Precision): 0.5254
  召回率 (Recall):    0.4870
類別 'dog':
  精確率 (Precision): 0.4930
  召回率 (Recall):    0.5280
類別 'frog':
  精確率 (Precision): 0.5737
  召回率 (Recall):    0.6770
類別 'horse':
  精確率 (Precision): 0.6308
  召回率 (Recall):    0.6850
類別 'ship':
  精確率 (Precision): 0.6854
  召回率 (Recall):    0.7430
類別 'truck':
  精確率 (Precision): 0.6293
  召回率 (Recall):    0.6790
```

[ ]:
```
# 6. 繪製學習曲線
costs_cifar10_cp = cp.array(cifar10_history['costs'])
accuracies_cifar10_cp = cp.array(cifar10_history['accuracies'])
costs_cifar10_np = cp.asnumpy(costs_cifar10_cp)
accuracies_cifar10_np = cp.asnumpy(accuracies_cifar10_cp)

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))
fig.suptitle('CIFAR-10 Model Learning Curves', fontsize=16)

ax1.plot(costs_cifar10_np, label='Training Loss')
```

```
ax1.set_title("Loss vs. Epochs")
ax1.set_xlabel("Epoch")
ax1.set_ylabel("Loss")
ax1.grid(True)
ax1.legend()

ax2.plot(accuracies_cifar10_np, label='Test Accuracy', color='orange')
ax2.set_title("Accuracy vs. Epochs")
ax2.set_xlabel("Epoch")
ax2.set_ylabel("Accuracy")
ax2.grid(True)
ax2.legend()

plt.show()
```



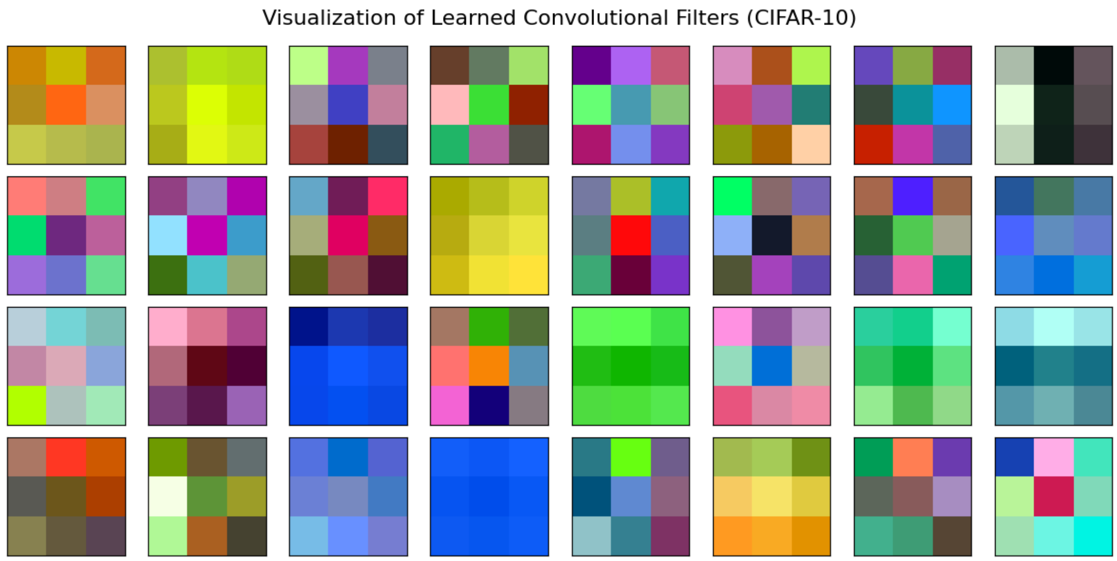
7.5 Bonus Step 5: 加分題：視覺化卷積核

[7]:
```
import numpy as np
import cupy as cp
import matplotlib.pyplot as plt
```

[11]:
```
trained_parameters = load_parameters("my_cifar10_cnn_model.npz")
assert trained_parameters is not None
if 'trained_parameters' in locals() or 'trained_parameters' in␣
↪globals():
    print("讀取已訓練的 CIFAR-10 模型權重...")

    # 1. 從權重字典中取出第一個卷積層的權重 (W_conv)
    W_conv_cp = trained_parameters['W_conv']
```

```
    # 2. 將 CuPy 陣列移至 CPU 並轉為 NumPy 陣列，以便 Matplotlib 處理
    W_conv_np = cp.asnumpy(W_conv_cp)

    # 3. 設定視覺化圖表
    # 我們有 32 個 filters，可以用一個 4x8 的網格來顯示
    num_filters = W_conv_np.shape[0]
    fig, axes = plt.subplots(4, 8, figsize=(12, 6))

    # 設定主標題
    fig.suptitle('Visualization of Learned Convolutional Filters␣
↪(CIFAR-10)', fontsize=16)

    # 4. 遍歷所有 filters 並在子圖中繪製
    for i, ax in enumerate(axes.flat):
        # 檢查是否還有 filter 可以顯示
        if i < num_filters:
            # 取出第 i 個 filter 的權重。
            filt = W_conv_np[i, :, :, :]

            f_min, f_max = filt.min(), filt.max()
            filt_normalized = (filt - f_min) / (f_max - f_min)

            # 轉置維度從 (C, H, W) 到 (H, W, C) 以符合 imshow 的格式
            filt_rgb = filt_normalized.transpose(1, 2, 0)

            # 使用 imshow 繪製 filter
            ax.imshow(filt_rgb)

        # 隱藏座標軸刻度，讓圖表更簡潔
        ax.set_xticks([])
        ax.set_yticks([])

    # 調整子圖之間的間距
    plt.tight_layout()
    # 顯示圖表
    plt.show()

else:
    print("錯誤：找不到 'trained_parameters'。請先確定您已成功訓練並載入␣
↪MNIST 模型。")
```

```
Loading parameters from my_cifar10_cnn_model.npz…
Parameters loaded successfully.
讀取已訓練的 CIFAR-10 模型權重…
```



8 Conclusion

本次作業成功地從零開始建構了一個功能完整的卷積神經網路，不僅達成了作業 hw4.pdf 的所有基本要求，也完成了加分挑戰項目，從而驗證了對 CNN 核心概念的深入理解。

在 MNIST 資料集上，模型表現出色，達到 98.60% 的高準確率。從學習曲線圖中可以看到，訓練損失穩定下降，而測試準確率則快速上升後趨於平穩，顯示模型有效地學習到了數字特徵。透過視覺化卷積核，可以觀察到模型在第一層學會了辨識數字的基礎筆劃，如邊緣、角落與線條等特徵。此外，提早停止（Early Stopping）機制的導入，成功地在模型性能達到巔峰時結束訓練，有效防止了過擬合。

在 CIFAR-10 加分項目中，模型雖然能夠運作，但 59.01% 的準確率遠低於 MNIST 的成果。這符合預期，因為 CIFAR-10 是由更複雜的自然圖像組成，其特徵（如紋理、顏色、物體形狀）遠比 MNIST 的單色筆劃複雜。本次實作的淺層 CNN 結構對於捕捉如此複雜的特徵能力有限。儘管準確率不高，但成功將模型拓展至處理 3 通道的彩色圖像，並完成端到端的訓練與評估，本身即證明了模型的擴展性與彈性。

總體而言，本次專案最大的收穫在於透過親手打造每一個環節，深刻體會了 CNN 的數學原理與運作流程。從巢狀迴圈到 im2col 的效能最佳化過程，也讓人理解到向量化運算在深度學習中的關鍵性。而 CuPy 的使用，則展示了在不依賴高階框架的前提下，利用 GPU 大幅提升訓練效率的可行性。