NYCU IEE Deep Learning
Lab 1 Report: Backpropagation

陳柏翔　313510156

1 Introduction

本作業旨在實現一個針對 Fashion MNIST Dataset 的 Neural Network 分類器。需於 Task 1 中使用 Numpy 套件實作整個 Neural Network 與 Forward、Backward、Step (參數更新) 等功能，再於 Task 2 中使用 PyTorch 套件實作與 Task 1 相同的 Neural Network。

我在本次作業中，基於 Numpy 套件與 CPU 執行時間的考量而設計了一個較簡單的 CNN 模型進行分類，並且透過 Adam Optimizer、Data Augmentation、Center Loss 等方法，使模型在不增加參數量的情況下也能提高準確率，最終不論 Task 1 還是 Task 2 都同樣在 Validation Set 上達到 94% 以上的準確率。

2 Dataset

2.1 Fashion MNIST

Fashion MNIST 是一個包含 10 種服飾類別的圖像資料集，如同 Figure 1 所示，每張圖片都是 28×28 像素的灰階圖片。而在此資料集中，存在著數個不同類別的圖像都是屬於上衣類型的圖片，因此容易造成分類上的困難。

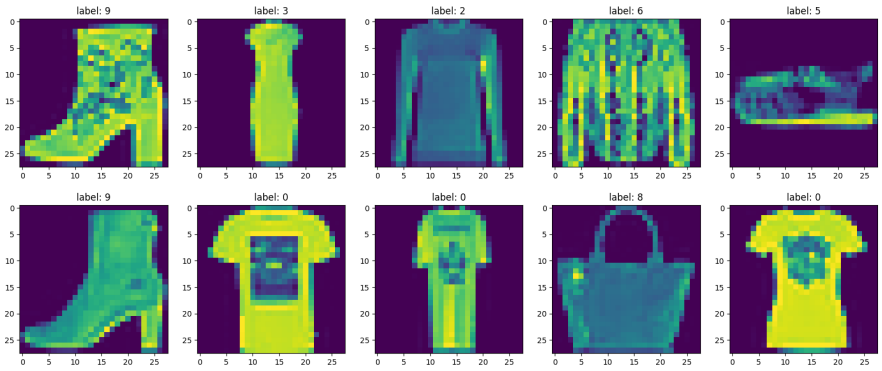

Figure 1: Images in Fashion MNIST dataset

2.2 Dataset Splitting

本次作業中，我們拿到的資料量為 60,000 張圖像。由於規定的限制，我們必須將 Training Set / Validation Set 的數量分別設置為 50,000 / 10,000，且基於公平考量，我使用助教預設的分配方式，每次執行分配時，分到 Training Set / Validation Set 的圖片都是固定的。

2.3 Preprocessing

由於我們所拿到的資料是圖像被展開後的一維 (784) 數值序列，因此需要先重整為三維 $(1, 28, 28)$ 陣列才有利於我們使用在 CNN 當中。此外，由於原始數值大小分布是在 [0, 255] 的區間，但是考量到數值的穩定性，以及梯度等問題，我對輸入圖像的數值處理縮放到 [-0.5, 0.5] 的區間。

2.4 Data Augmentation

為提高模型的泛化能力，訓練模型時我會對輸入圖片在送入模型之前做隨機的轉換，轉換包含 Shift、Flip、Rotate、None 四種，每一種轉換被選到的機率皆相同。而在做完第一次轉換後，還會再經過隨機平均值與隨機標準差的調整（等校於隨機亮度、對比度調整）。

此部分的有效性根據我多次實驗結果，大約可以提升 1% 至 2% 的準確率。(註：由於規定不能使用影像處理的套件，因此我在 Task 1 中是以 Numpy 實現以上幾種轉換，而在 Task 2 中使用 PyTorch 與 Numpy 實現。)

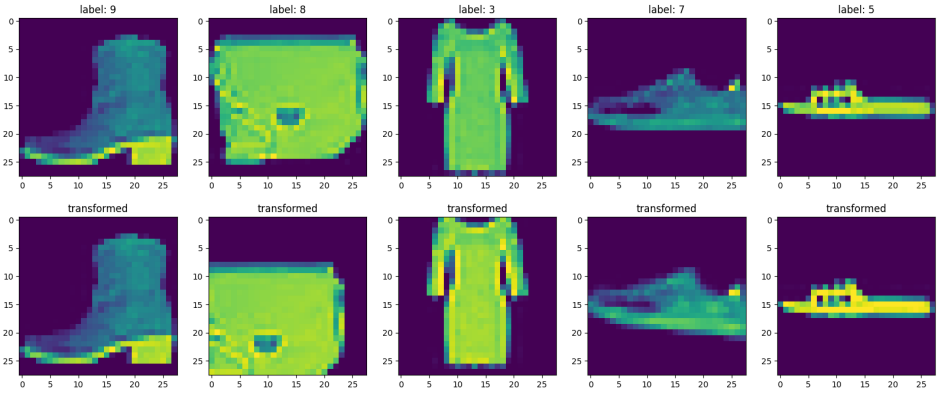

Figure 2: Example of Random Transformed Images

2.5 Balanced Data Sampling

為了確保每一個類別的訓練資料量都是平衡的，避免訓練完的模型偏好去猜某個特定的類別，我會先將每個類別的資料整理在一起，當訓練進行時再從各個類別中隨機抽樣出 `(batch_size // num_class)` 數量張圖片作為訓練資料。

3 Architecture

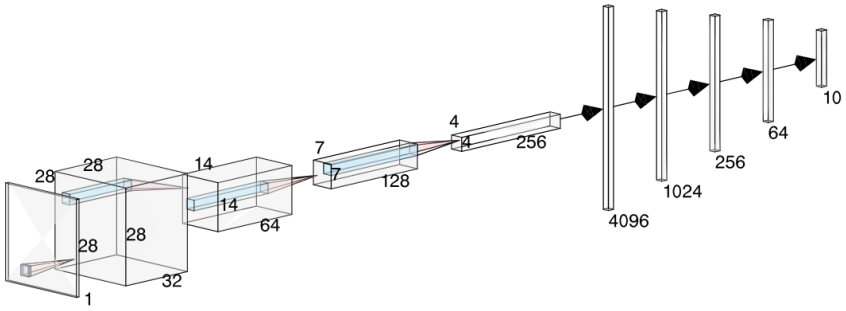

Figure 3: Network Architecture

3.1 Network

我在本作業中所設計的模型如 Figure 3 所示。大致上可以劃分為前半部分以特徵圖為主的 Encoder，負責提取空間特徵；以及後半部分以特徵向量為主的 Decoder，依照 Encoder 提取的特徵來進行分類。各層之參數量與輸入、輸出特徵向量形狀詳見章節 5。

Encoder. 在此部分由 Conv2d、BatchNorm2d、SiLU 的組合重複四次堆疊而成。在此設計中，我除了用 Convolution 來提取特徵，也透過設置 stride 為 2 的方式對特徵圖進行降維；而 Batch Normalization 則是對輸入的特徵做標準化，可以維持梯度傳遞的有效性；最後，SiLU 是一種 Activation Function，提供模型擬合非線性函數的能力，將於下一小節討論。

Decoder. 在此部分則是以 Linear、BatchNorm1d、SiLU 的組合重複四次堆疊而成，最後的輸出是 10 維的向量，經過 Softmax 轉換就能得到各個類別的機率。在此設計中，後兩者所扮演的功能與其在 Encoder 中的功能相同；而 Linear (Fully-connected) 則是用來從已經不具有明顯空間相關性的特徵向量中提取特徵。

3.2 Activation Function

在深度學習模型中，Activation Function 的選擇會直接影響模型擬合非線性函數的能力與收斂速度。SiLU（Sigmoid Linear Unit，又稱 Swish），如 Figure 4 所示是一種平滑非單調的函數，形式為 $f(x) = x \cdot \sigma(x)$，能在負值區域保留部分小於零的輸出，而非 ReLU 那樣將所有負值截斷為零。

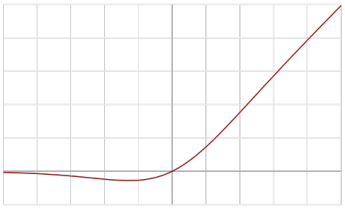

Figure 4: SiLU (Swish) Function [1]

這樣的特性有助於減緩梯度消失的問題 (在 ReLU 中，輸入為負值的元素在 Backward 時梯度為 0)，能使模型在訓練過程中更穩定。此外，SiLU 在輸出上具有連續且可微的特性，能為優化器提供更平滑的 Loss 曲面，相對於 ReLU 更容易達到收斂。至於沒有選用其他 Activation Function (E.g. ELU) 的原因，是根據過去的實驗結果顯示 SiLU 有更好的效果。

3.3 Loss Functions

在此作業中，我使用的 Loss Functions 包含了兩個部分，首先以 Cross Entropy Loss 為主要的 Loss Function，會直接作用在最後一層的 10 維輸出向量；其次以 Center Loss 作為輔助，作用在輸出向量維度為 256 的 Linear 層進行訓練。

Cross Entropy Loss. 主要是用來計算模型的輸出與期望的輸出結果間的相近程度 (特別適用於多維度的輸出，如果輸出只有一個數值的話就可以改用 Binary Cross Entropy)，計算式如下：

$$\text{Cross Entropy Loss}(\boldsymbol{y}, \hat{\boldsymbol{y}}) = -\sum_i y_i \log(\hat{y}_i)$$

其中 $\boldsymbol{y}$ 代表期望的輸出，而 $\hat{\boldsymbol{y}}$ 代表模型的輸出。如果結果越相近，那麼計算後的 Loss 值就會越小，反之差距越大的話 Loss 值就會越大。在計算 Backward 時，對模型的輸出偏微分就可以得到：

$$\frac{\partial L}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$$

但是由於實作時是結合 Softmax 一起計算，而 Softmax 計算式與其對輸入偏微分為：

$$\hat{y}_i = \frac{e^{x_i}}{\sum_i e^{x_i}}, \qquad \frac{\partial \hat{y}_i}{\partial x_j} = \begin{cases} \hat{y}_i(1 - \hat{y}_i), & i = j \\ -\hat{y}_i \hat{y}_j, & i \neq j \end{cases}$$

結合兩個計算式就能得到 Cross Entropy Loss 給模型輸出要更新的量：

$$\frac{\partial L}{\partial x_j} = \sum_i \frac{\partial L}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial x_j} = \hat{y}_j - y_j$$

Center Loss. 在多次的實驗過程中，我發現我的模型只能勉強地達到約 93% 的準確率，因此我想確認模型內部的學習成效使否良好。結果如 Figure 5 左圖所示，我透過將不同類別的圖像輸入到模型中，並紀錄倒數第三個 Linear 層輸出的特徵向量，將特徵向量的分布利用 t-SNE 從 256 維降維至 2 維畫出來看他們的分布狀況。

如果模型學習得越好，不同類別的特徵向量就要分得越開；而學習得越差，就越會看到不同類別的向量彼此混雜在一起。因此我決定採用 Center Loss [2] 來幫助模型訓練，使同個類別的特徵向量彼此更加緊湊，其計算式如下：

$$\text{Center Loss} = \sum_{k} \left\| \boldsymbol{x}_k - \boldsymbol{c}_{y_k} \right\|_2^2$$

其中，$\boldsymbol{x}_k$ 代表模型輸出的特徵向量，而 $\boldsymbol{c}_{y_k}$ 代表對應類別的中心點。而中心點是可學習的參數，每次更新都會往該類別的特徵向量平均中心點移動，其他的特徵向量則都要向著中心點靠攏。

在加上 Center Loss 後，訓練結果如 Figure 5 右圖所示，雖然仍有部分不同類別的特徵向量重疊，但是可以看出 Label 為 0, 2, 3, 4, 6 的部分較原本的更分開，而我認為沒辦法分的更好是因為模型的參數量不夠大，因此在訓練後也難以區分較相似的不同類別圖像，但是經過我的測試，即使增加模型複雜度，準確率也沒有明顯的提升，因此模型的部分我沒有再做改動。

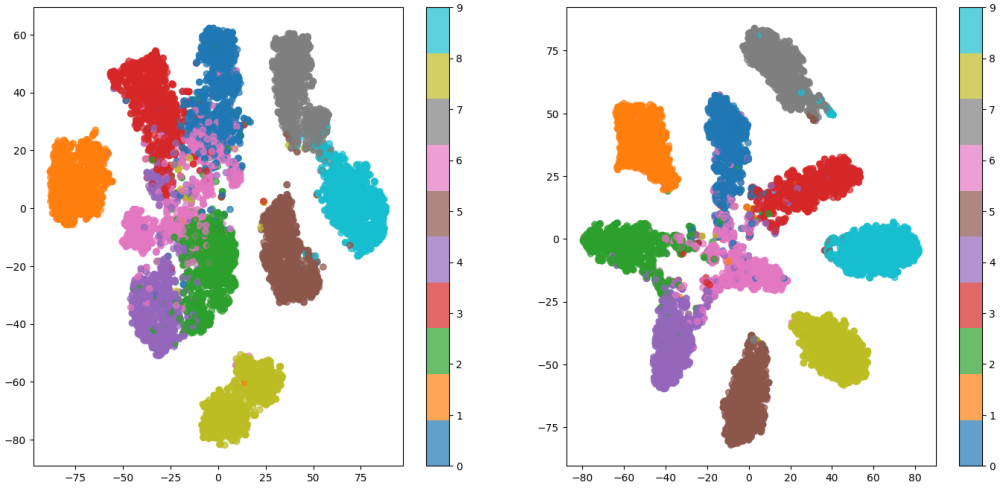

Figure 5: Features Distribution Plot. 圖中的顏色對應各個類別的圖片在輸入後的特徵向量分布位置。未加上 Center Loss 訓練 100 Epochs 後的結果 (左圖)；以及加上 Center Loss 訓練 100 Epochs 後的結果 (右圖)，可見 Center Loss 可以使同個類別的特徵向量更加集中。

3.4 Optimizer

在更新模型參數時，我選用 Adam Optimizer 來進行優化。Adam Optimizer 結合了 Momentum 以及 RMSProp 的能力，能夠依照過去的梯度方向做當前梯度的速度調整，還能夠自適應調整學習能力 (相當於自動調整 Learning rate)。

4 Experiments

4.1 Confusion Matrix

Figure 6 為模型訓練超過 100 Epochs 後的預測結果所畫成的 Confusion Matrix，可以發現 Label 為 6 的圖片特別難以分類。

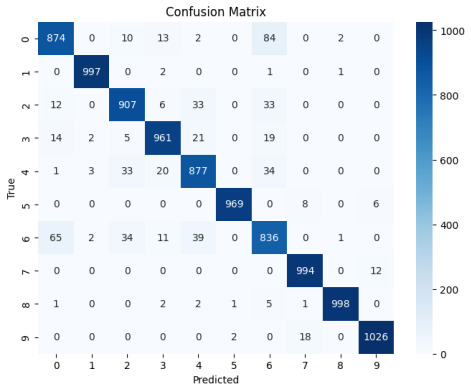

Figure 6: Confusion Matrix

由於明顯有個類別分類的最差，因此我就試著針對 Label 為 6 的資料設定較大的 Loss 作為懲罰，結果卻無法提升模型訓練後的精準度；接著我又改成對於每個 Batch 組成時刻意用 Imbalanced Sampling 的方式，對 Label 為 6 的圖片故意取樣較多，這樣雖然能讓預測結果為 6 的機率上升，但是卻會造成其他相近的類別 (E.g. Label 為 0) 更容易被誤判為類別 6 。因此在綜合考量之下，我還是採用 Balanced Sampling 的方式進行訓練。

4.2 Implementation Results

在本次作業中，我用 Numpy 與 PyTorch 實作兩個版本的 CNN 其結果大致相同，訓練到最終都能在 Validation Set 上達到 94% 以上的準確率。但是經過我的多次嘗試發現，在 Numpy 實作的模型比起 PyTorch 實作的模型難收斂許多，需要在訓練過程中手動調整一些參數 (E.g. Learning Rate, Center Loss Factor) 來幫助模型在適合的時間點改變優化目標。

4.2.1 Task 1: Numpy Results

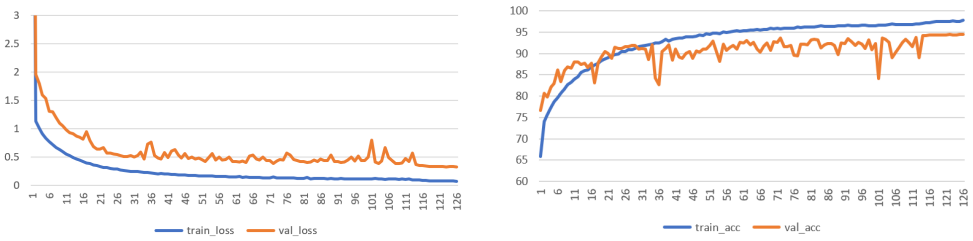

Figure 7: (Task 1) Loss and Accuracy during Training Process

```
Task-1 | Epoch: 122 | Train Loss:  0.0741 | Train Acc: 97.5720 | Val Loss:  0.2509 | Val Acc: 94.3200 | Epoch time: 237.17 sec
Task-1 | Epoch: 123 | Train Loss:  0.0761 | Train Acc: 97.5440 | Val Loss:  0.2509 | Val Acc: 94.3500 | Epoch time: 240.92 sec
Task-1 | Epoch: 124 | Train Loss:  0.0752 | Train Acc: 97.5000 | Val Loss:  0.2503 | Val Acc: 94.4100 | Epoch time: 240.90 sec
Task-1 | Epoch: 125 | Train Loss:  0.0695 | Train Acc: 97.7040 | Val Loss:  0.2495 | Val Acc: 94.4900 | Epoch time: 240.43 sec
Task-1 | Epoch: 126 | Train Loss:  0.0725 | Train Acc: 97.5600 | Val Loss:  0.2507 | Val Acc: 94.4900 | Epoch time: 239.77 sec
```

Figure 8: Task 1 Screenshot (Val Acc. 94%)

4.2.2 Task 2: PyTorch Results

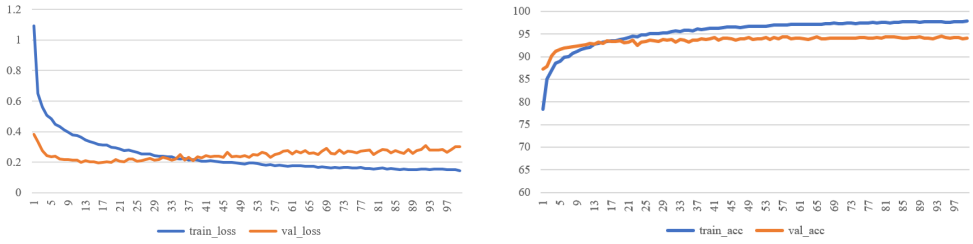

Figure 9: (Task 2) Loss and Accuracy during Training Process

```
Task-2  | Epoch: 97 | Train Loss:  0.1508 | Train Acc: 97.7620 | Val Loss:  0.2634 | Val Acc: 94.1900 | Epoch time: 2.6
Task-2  | Epoch: 98 | Train Loss:  0.1515 | Train Acc: 97.7020 | Val Loss:  0.2849 | Val Acc: 94.1900 | Epoch time: 2.6
Task-2  | Epoch: 99 | Train Loss:  0.1511 | Train Acc: 97.7360 | Val Loss:  0.3003 | Val Acc: 93.9200 | Epoch time: 2.6
Task-2  | Epoch:100 | Train Loss:  0.1448 | Train Acc: 97.8700 | Val Loss:  0.3016 | Val Acc: 94.1300 | Epoch time: 2.6
```

Figure 10: Task 2 Screenshot (Val Acc. 94%)

最後在 Task 2 中，我使用 PyTorch 做出同樣的模型，收斂明顯較 Numpy 穩定許多，我認為這展示出了 PyTorch 的計算圖機制會比我們自己去推導來的完備許多，因此很感謝現在已經有這些工具能夠使用，方便我們開發複雜的演算法架構。

雖然此次作業無須使用 GPU 進行訓練，但是為了在 Task 2 更有效率的得到訓練結果，我後來將 GPU 的使用也加了上去，因此每 Epoch 的執行時間縮短到個位數秒。

5 Other Details

5.1 Network Parameters

- Layer details: (詳見 Table 1)
- Total number of parameters: 4,866,058

5.2 Hyperparameters

- Epochs: 200
- Learning rate: 0.01 (Task 1) or 0.001 (Task 2)
- Learning rate decay: times 0.5 every 10 epochs after 100 epoch.
- Batch size: 250 (Task 1) or 500 (Task 2)

考量到 Numpy 與 PyTorch 的收斂速度差異，我在 Task 1 與 Task 2 的設置不相同。此外，Epochs 有時候不需要訓練太久但有時候卻又要訓練很久，因此我的設置不完全固定是 200。

Layer Name	Parameters	Input Shape	Output Shape
Conv2d	320	(1, 28, 28)	(32, 28, 28)
BatchNorm2d	64	(32, 28, 28)	(32, 28, 28)
SiLU	0	(32, 28, 28)	(32, 28, 28)
Conv2d	18,496	(32, 28, 28)	(64, 14, 14)
BatchNorm2d	128	(64, 14, 14)	(64, 14, 14)
SiLU	0	(64, 14, 14)	(64, 14, 14)
Conv2d	73,856	(64, 14, 14)	(128, 7, 7)
BatchNorm2d	256	(128, 7, 7)	(128, 7, 7)
SiLU	0	(128, 7, 7)	(128, 7, 7)
Conv2d	294,912	(128, 7, 7)	(256, 4, 4)
BatchNorm2d	512	(256, 4, 4)	(256, 4, 4)
SiLU	0	(256, 4, 4)	(256, 4, 4)
Flatten	0	(256, 4, 4)	(4096)
Linear	4,195,328	(4096)	(1024)
BatchNorm1d	2,048	(1024)	(1024)
SiLU	0	(1024)	(1024)
Linear	262,400	(1024)	(256)
BatchNorm1d	512	(256)	(256)
SiLU	0	(256)	(256)
Linear	16,448	(256)	(64)
BatchNorm1d	128	(64)	(64)
SiLU	0	(64)	(64)
Linear	650	(64)	(10)
Softmax	0	(10)	(10)

Table 1: Layers of This Work

5.3 Fast Convolution in Numpy

儘管 Numpy 也提供 FFT (快速傅立葉轉換) 相關函式，理論上可以在 Frequency Domain 中運算來加速 Convolution 的計算，但是在實際的 CNN 中，常用的卷積核多半是尺寸較小（E.g. 3×3, 5×5），且數量眾多（E.g. 32、64、128），此時頻繁地對影像與卷積核進行 FFT 反而可能無法帶來效能提升，甚至增加計算成本。

因此在這類情況下，於 Spatial Domain 中進行 Convolution 通常會更有效率。不過，若僅以 For Loop 在 Numpy 中實作乘加運算，效能仍然不足。相對地，利用矩陣運算函式可直接調用 BLAS 庫，能更高效地完成計算。因此，較佳的作法是先透過 `im2col` 將影像轉換為列向量，再進行矩陣運算，最後再藉由 `col2im` 將結果還原回影像形式，詳見 Appendix A。

References

[1] Wikipedia contributors. Swish function — Wikipedia, the free encyclopedia. https://en.wikipedia.org/wiki/Swish_function, 2025. [Online; accessed 20-September-2025].

[2] Yandong Wen, Kaipeng Zhang, Zhifeng Li, and Yu Qiao. A discriminative feature learning approach for deep face recognition. In *European conference on computer vision*, pages 499–515. Springer, 2016.

A `im2col` and `col2im`

Algorithm 1 Python and Numpy-like Pseudo Code of `im2col` and `col2im` Functions

```python
def im2col(
        input: np.ndarray, k_size: int, stride: int, padding: int
    ) -> tuple[np.ndarray, int, int]:
    r""" Convert image to column matrix.
    - `input`: (batch_size, in_channels, height, width)
    - `return`: (batch_size, in_channels * k_size * k_size, out_h * out_w)
    """
    batch_size, in_c, in_h, in_w = input.shape
    out_h = (in_h + 2 * padding - k_size) // stride + 1
    out_w = (in_w + 2 * padding - k_size) // stride + 1

    # Padding
    input_padded = np.pad(
        input, ((0, 0), (0, 0), (padding, padding), (padding, padding)), mode="constant"
    ) if padding > 0 else input

    col_idx = 0
    cols = np.zeros((batch_size, in_c * k_size * k_size, out_h * out_w))
    for r in range(0, in_h + 2 * padding - k_size + 1, stride):
        for c in range(0, in_w + 2 * padding - k_size + 1, stride):
            patch = input_padded[:, :, r:r+k_size, c:c+k_size]
            cols[:, :, col_idx] = patch.reshape(batch_size, -1)
            col_idx += 1

    return cols, out_h, out_w


def col2im(
        cols: np.ndarray, input_shape: tuple[int, int, int, int],
        k_size: int, stride: int, padding: int
    ) -> np.ndarray:
    r""" Convert column matrix back to image.
    - `input`: (batch_size, out_h * out_w, in_channels * k_size * k_size)
    - `return`: (batch_size, in_channels, height, width)
    """
    batch_size, in_c, in_h, in_w = input_shape
    out_h = (in_h + 2 * padding - k_size) // stride + 1
    out_w = (in_w + 2 * padding - k_size) // stride + 1

    cols_reshaped = cols.reshape(batch_size, out_h, out_w, in_c, k_size, k_size)

    # Initialize padded image
    h_padded, w_padded = in_h + 2 * padding, in_w + 2 * padding
    img_padded = np.zeros((batch_size, in_c, h_padded, w_padded), dtype=cols.dtype)

    # Reconstruct image
    for y in range(out_h):
        for x in range(out_w):
            h_start, w_start = y * stride, x * stride
            img_padded[
                :, :, h_start:h_start+k_size, w_start:w_start+k_size
            ] += cols_reshaped[:, y, x, :, :, :]

    return img_padded if padding == 0 else img_padded[:, :, padding:-padding, padding:-padding]
```