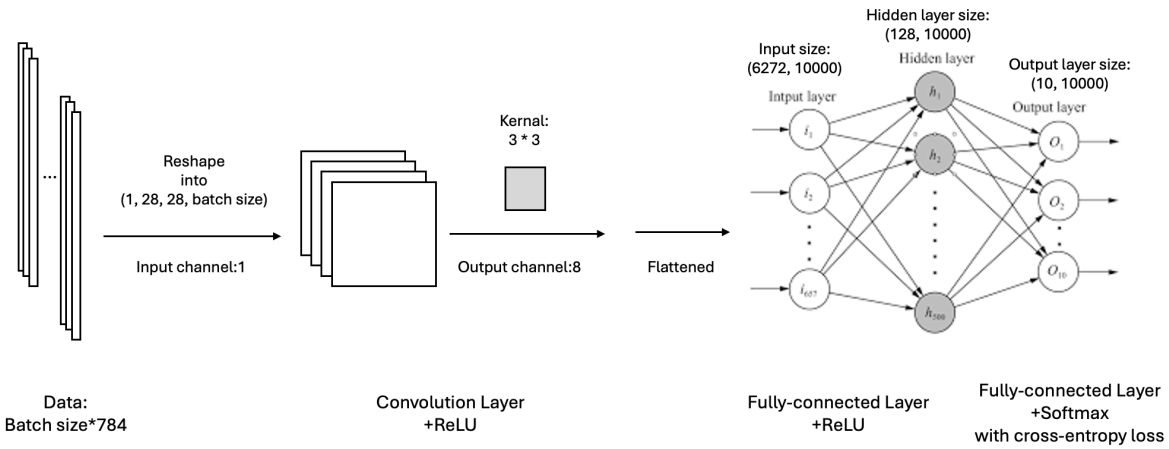
# Deep Learning Lab01

## 111261022 林品妤

# < Task 1 >

### Framework:



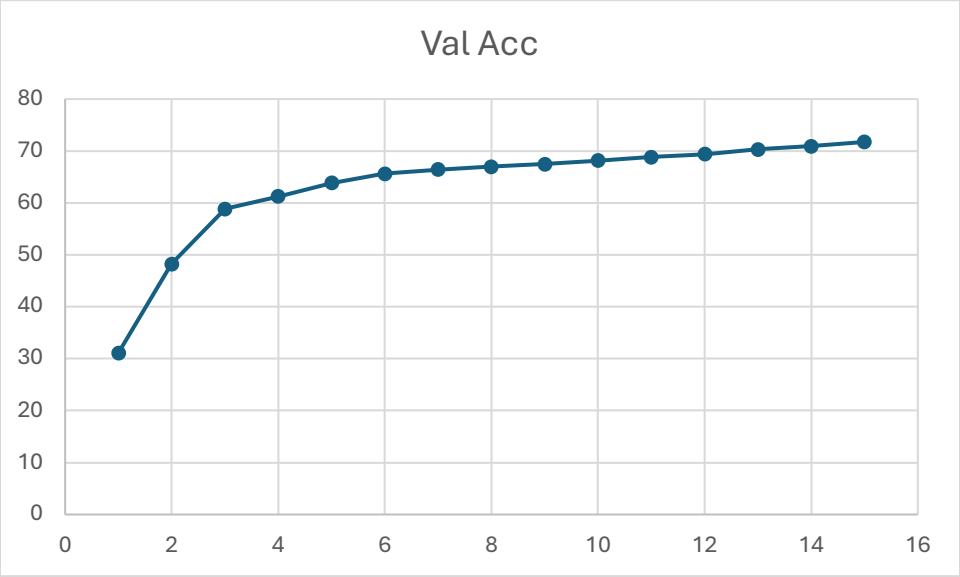

### Data shape in flow:

### (60000, 784)

### <u>Validation accuracy: 71.74%</u>

```
        val_hit += (pred_index==train_label[tit*Batch_size:(tit+1)*Batch_size]).sum()
        total_val_loss += val_loss
    end_time = time.time()
    epoch_time = end_time - start_time
    print('Task-1  | Epoch:%3d'%epoch, ' |Train Loss:%8.4f'%(total_train_loss/train_batch_num), ' |Train
     Acc:%3.4f'%(train_hit/(train_image_num-val_image_num)*100.0)
          , ' |Val Loss:%8.4f'%(total_val_loss/val_batch_num), ' |Val Acc:%3.4f'%
           (val_hit/val_image_num*100.0), ' |Epoch time:%5.2f'%(epoch_time),' sec')
✓ [15] 17h 57m
current output shape:  (8, 28, 28, 10000)
finish running layer:  <model.layer.Activation1 object at 0x1107353a0>
current output shape:  (8, 28, 28, 10000)
finish running layer:  <model.layer.Flatten object at 0x110729c70>
current output shape:  (6272, 10000)
finish running layer:  <model.layer.FullyConnected object at 0x110729370>
current output shape:  (128, 10000)
finish running layer:  <model.layer.Activation1 object at 0x1105dc7f0>
current output shape:  (128, 10000)
finish running layer:  <model.layer.FullyConnected object at 0x1105dcd60>
current output shape:  (10, 10000)
Task-1  | Epoch: 15  |Train Loss:  0.8385  |Train Acc:70.6080  |Val Loss:  0.8196  |Val Acc:71.7400
|Epoch time:1975.33  sec
```

**Validation Accuracy:**




# < Task 2 >

## Validation Accuracy: 83.7%

layer.py layer_wrong.py Lab1_task1.ipynb Lab1_task2.ipynb network.py __init__.py

```
    for titt in range(val_batch_num):
        tit=train_batch_num+titt
        inputs = train_data_tensor[tit*Batch_size:(tit+1)*Batch_size]
        labels = train_label_tensor[tit*Batch_size:(tit+1)*Batch_size]

        outputs = net(inputs)
        loss = criterion(outputs, labels)

        total_val_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        val_hit += (predicted == labels).sum().item()
end_time = time.time()
epoch_time = end_time - start_time
print('Task-2  | Epoch:%3d'%epoch, ' |Train Loss:%8.4f'%(total_train_loss/train_batch_num), ' |Train Acc:%3.4f'%(train_hit
      , ' |Val Loss:%8.4f'%(total_val_loss/val_batch_num), ' |Val Acc:%3.4f'%(val_hit/val_image_num*100.0), ' |Epoch time:
```

[63] 34s 352ms

```
Task-2  | Epoch:  8  |Train Loss:  1.1426  |Train Acc:75.5540  |Val Loss:  1.0774  |Val Acc:76.1600  |Epoch time: 1.71  sec
Task-2  | Epoch:  9  |Train Loss:  1.0137  |Train Acc:75.8880  |Val Loss:  0.9137  |Val Acc:77.0100  |Epoch time: 1.71  sec
Task-2  | Epoch: 10  |Train Loss:  0.8820  |Train Acc:77.3620  |Val Loss:  0.8185  |Val Acc:78.4200  |Epoch time: 1.70  sec
Task-2  | Epoch: 11  |Train Loss:  0.7922  |Train Acc:78.1300  |Val Loss:  0.7315  |Val Acc:78.4800  |Epoch time: 1.70  sec
Task-2  | Epoch: 12  |Train Loss:  0.7246  |Train Acc:78.4500  |Val Loss:  0.6956  |Val Acc:79.8800  |Epoch time: 1.70  sec
Task-2  | Epoch: 13  |Train Loss:  0.6733  |Train Acc:79.6500  |Val Loss:  0.6370  |Val Acc:81.1600  |Epoch time: 1.70  sec
Task-2  | Epoch: 14  |Train Loss:  0.6301  |Train Acc:80.8400  |Val Loss:  0.6052  |Val Acc:81.7800  |Epoch time: 1.70  sec
Task-2  | Epoch: 15  |Train Loss:  0.5946  |Train Acc:81.3720  |Val Loss:  0.5801  |Val Acc:82.2000  |Epoch time: 1.70  sec
Task-2  | Epoch: 16  |Train Loss:  0.5706  |Train Acc:82.0920  |Val Loss:  0.5542  |Val Acc:82.6600  |Epoch time: 1.70  sec
Task-2  | Epoch: 17  |Train Loss:  0.5461  |Train Acc:82.4220  |Val Loss:  0.5352  |Val Acc:82.7900  |Epoch time: 1.70  sec
Task-2  | Epoch: 18  |Train Loss:  0.5279  |Train Acc:82.6980  |Val Loss:  0.5167  |Val Acc:83.0400  |Epoch time: 1.70  sec
Task-2  | Epoch: 19  |Train Loss:  0.5054  |Train Acc:83.2780  |Val Loss:  0.5020  |Val Acc:83.3800  |Epoch time: 1.70  sec
Task-2  | Epoch: 20  |Train Loss:  0.4992  |Train Acc:83.3740  |Val Loss:  0.4872  |Val Acc:83.7000  |Epoch time: 1.70  sec
```

Code Markdown

## < Methods for improve accuracy >

### 1. Add Convolution Layer

優點：由於大部分在處理圖形時，convolution layer 能較好的截取到影像內的特徵，因此，在 Task 1 & 2 的學習架構中，我使用了 convolution layer 作為第一層

下圖為沒有加入 convolution 層時跑出的結果（僅用兩層 fully connected layers 和 ReLU、Softmax）：

```
current grad shape:  (128, 10000)
Finish running layer:  <model.layer.Activation1 object at 0x107716760>
current grad shape:  (128, 10000)
Finish running layer:  <model.layer.FullyConnected object at 0x107b70c10>
current grad shape:  (784, 10000)
finish running layer:  <model.layer.FullyConnected object at 0x107b70c10>
current output shape:  (128, 10000)
finish running layer:  <model.layer.Activation1 object at 0x107716760>
current output shape:  (128, 10000)
finish running layer:  <model.layer.FullyConnected object at 0x103851a60>
current output shape:  (10, 10000)
Task-1  | Epoch: 15  |Train Loss:  2.2828  |Train Acc:13.5980  |Val Loss:  2.2998  |Val
Acc:10.4700  |Epoch time: 0.24  sec
```

可以看到訓練出來的正確率低到不可思議，但耗時也變非常非常短。

缺點：大大增加模型訓練的時間，尤其在 Task 1 中，由於繁複的迴圈計算，需要耗費更大量的時間

可改進處：將迴圈改為以向量計算，可節省運算時間

### 2. Use ReLU as activation function

為保留截取影像的非線性關係的特徵，需要使用 Activation Function 來將非線性關係考慮進入學習過程。

ReLU 優點：

- 讓網路能學習 非線性關係，避免僅限於線性分類。
- 梯度在正區域不會飽和，能加快收斂速度。
- 相較於 sigmoid/tanh，ReLU 的計算更簡單，訓練效率更高。

### 3. Increase the numbers of epoch

訓練更多 epoch 讓模型能多次觀察完整資料集，使權重逐漸收斂。

同時搭配監控訓練與驗證集的 loss、accuracy 以避免過擬合。

4. **將 Convolution layer 以向量計算形式替代多重迴圈**

一開始進行訓練時，由於 Convolution 實在耗時太長，導致無法做太多 epoch，可以從下圖看到原本的訓練過程需要耗時 17 小時（Epoch=15）。

```
        (val_hit/val_image_num*100.0), ' |
✓ [15] 17h 57m
current output shape:  (8, 28, 28, 10000)
finish running layer:  <model.layer.Activat
current output shape:  (8, 28, 28, 10000)
```

因此，為了縮短訓練時間，我查找了能縮短計算時間的方法，發現多重迴圈若能改為向量運算能縮短許多倍的時間。

如下圖所示，Epoch 運算到第 17 個僅花費了 3 小時。

```
43          , ' |Val Loss:%8.4f'%(total_val_loss/val_batch_num), ' |Val Acc:%3.4f'%
             (val_hit/val_image_num*100.0), ' |Epoch time:%5.2f'%(epoch_time),' sec')
  3h 45m
current output shape:  (128, 10000)
finish running layer:  <model.layer.Activation1 object at 0x102f0a3d0>
current output shape:  (128, 10000)
finish running layer:  <model.layer.FullyConnected object at 0x103418b50>
current output shape:  (10, 10000)
Task-1  | Epoch: 16  |Train Loss:  0.8446  |Train Acc:68.8160  |Val Loss:  0.8160  |Val
```

不過因為時間安排不當的關係，最後來不及跑完 20 個 Epoch。T__T

5. **Add pooling layer**（最後因為修改重新訓練時間不足，並未真的加入使用）

池化層透過對卷積後的特徵圖進行降維，能：

- 縮小特徵圖尺寸，降低參數與計算成本。

常見做法：

- Max Pooling：取區域內最大值，強調最顯著特徵。
- Average Pooling：取區域內平均值，保留整體平滑資訊。

適當插入池化層，可使模型在保持關鍵特徵的同時，有效控制複雜度。

## < Differences in Task 1 and 2 >

1. **Spending time**

   Task1 - 30 min/epoch **vs.** Task2 - 1.7 sec/epoch

   程式運算時，可以發現到手刻和 pytorch 所需要的時間差高達一千多倍。而原因將在以下討論：

   (1) 底層運算最佳化程度

   雖然 Numpy 底層是以 C/Fortran 實作的向量化運算，但手刻的「forward / backward pass」和「梯度計算」通常是 Python 迴圈或大量矩陣切片組合，無法完全利用 NumPy 的向量化優勢。每一步都會觸發 Python 解譯器執行，產生大量函式呼叫與記憶體配置成本。

   PyTorch 核心是用 C++/CUDA 寫的，並針對 CPU、GPU 進行高度優化（使用 BLAS、cuDNN、MKL）。計算圖和梯度的流程是批次化、向量化的，不需要逐步在 Python 中顯式寫出每個計算。

   (2) GPU / SIMD 平行化

   手刻 NumPy 版本幾乎只用到‘CPU 單執行緒’或有限的 BLAS 加速。
   PyTorch 則在 CPU 上會用到‘多執行緒 + SIMD’（Intel MKL/OpenBLAS）。

   (3) 記憶體與計算圖管理

   手刻時，可能每次 forward/backward 都要重新配置中介變數。PyTorch 的’autograd’會動態建立計算圖、重複使用 buffer，減少額外開銷。

   (4) 數值穩定性與特殊 Kernel

   手刻的 ReLU、Softmax、交叉熵損失等，皆為直接翻公式。PyTorch 的實作會針對：溢位/下溢（overflow/underflow）、log-sum-exp 等技巧進行最佳化，既穩定又快速。

   (5) 編譯與 JIT:

NumPy 是解譯執行。PyTorch 有 JIT (Just-In-Time) 編譯、C++ Extension、以及針對常見運算的 fused kernel，可把多個操作融合成單一 kernel 執行，減少 I/O overhead。

要讓手刻程式更接近 PyTorch 的效能，可以嘗試：

* 將所有運算改為純向量化（避免 for-loop）。
* 使用 `numba` 或 `Cython` 加速。
* 使用 `cupy` 或 `jax`，讓運算轉到 GPU。

2. **權重初始值的設定**

**Task 1 np.random.randn * 0.01 vs. Task 2 依照 Activation function 自動調整**

若時間充分，其實 Task 1 應該也能選擇最適合的權重初始化方式來寫入程式進行計算，不過由於訓練的耗時過長，最後並沒有嘗試帶入其他方法設定初始值。

**而 PyTorch 的預設權重初始值的方法大致如下：**

nn.Linear 和 nn.Conv2d 模組針對不同的激活函數，會自動選擇最適合的權重初始化方法。

**ReLU 激活函數:**
對於 ReLU 激活層，PyTorch 預設使用的是 Kaiming 統一初始化 (Kaiming Uniform Initialization)，也被稱為 He 初始化 (He Initialization)。

它能確保網路每一層的輸入和輸出在訓練過程中保持適當的方差，從而有效地防止梯度消失（vanishing gradient）或梯度爆炸（exploding gradient）的問題。

**其他激活函數:**
若使用 Sigmoid 或 Tanh，PyTorch 則會使用 Xavier 統一初始化 (Xavier Uniform Initialization)。