**Deep Learning Lab01**

**111261022 林品妤**

**< Task 1 >**

**Framework:**

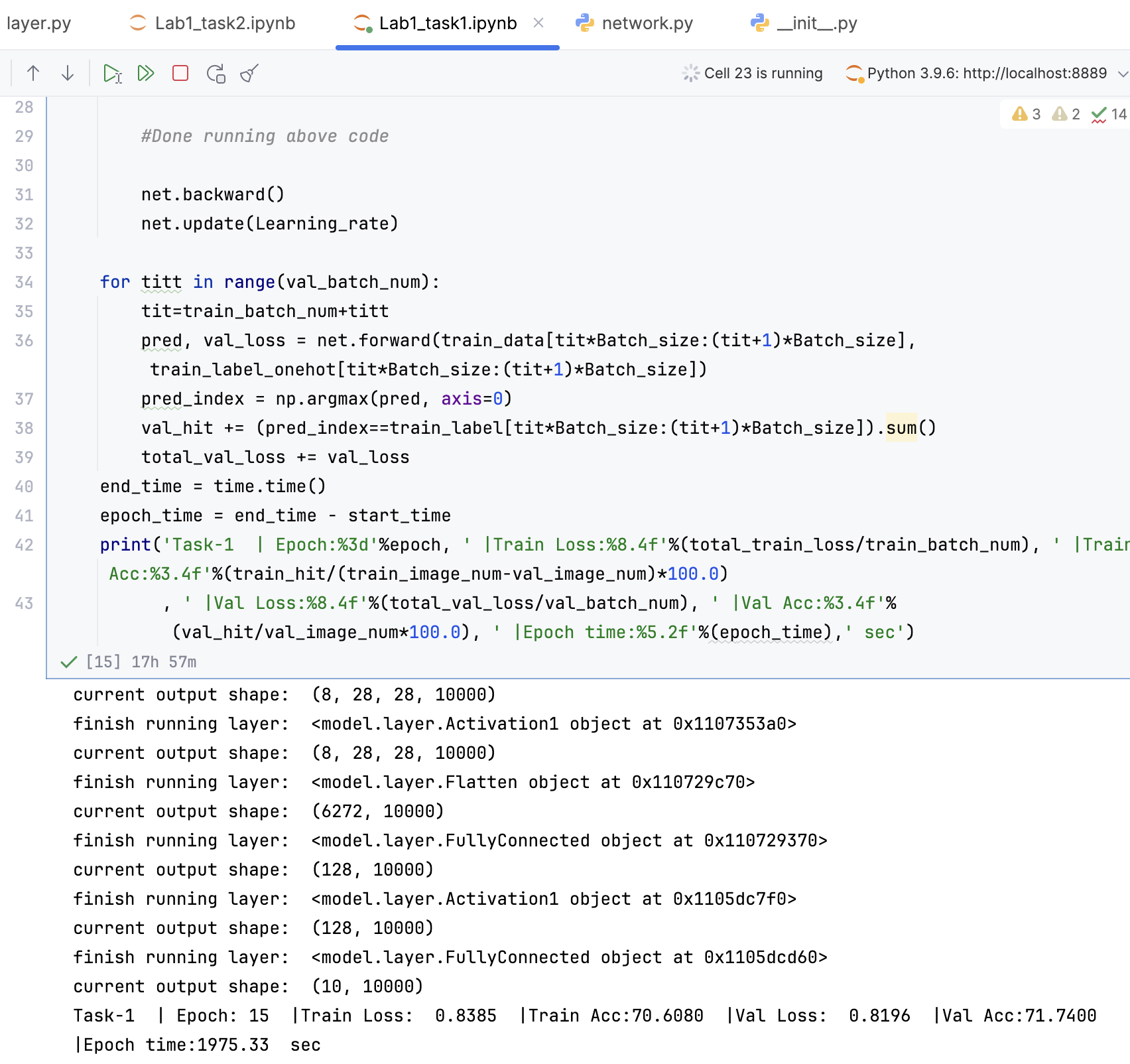
**A diagram of a computer network

AI-generated content may be incorrect.**

**Data shape in flow:**

**(60000, 784)**

**Validation accuracy: 71.74%**

****

**Validation Accuracy:**

**< Task 2 >**

**Validation Accuracy: 83.7%**

**A screenshot of a computer

AI-generated content may be incorrect.**

**< 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）：  
A screenshot of a computer

AI-generated content may be incorrect.

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

1. **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。

1. **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。

1. **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）。

**A screenshot of a computer

AI-generated content may be incorrect.**

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。

A screenshot of a computer program

AI-generated content may be incorrect.

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

1. **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。

1. 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）。

1. 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。

1. 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。

1. 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。

1. **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)。