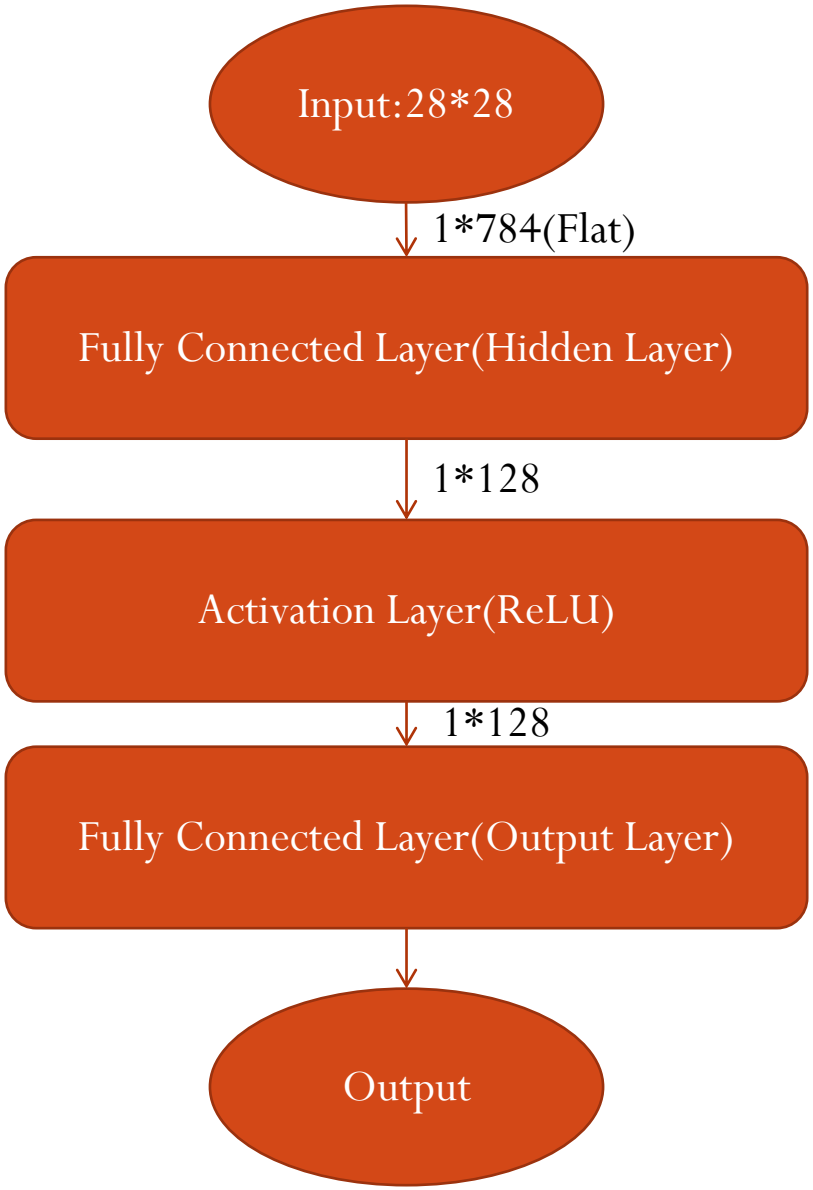
# AILAB-LAB1
# TASK1-MNIST

- NN架構:


Input:28*28
1*784(Flat)
Fully Connected Layer(Hidden Layer)
1*128
Activation Layer(ReLU)
1*128
Fully Connected Layer(Output Layer)
Output

- 架構實作(Coding) －Fully Connected Layer

```python
class Linear(Module):
    def __init__(self, in_features, out_features) -> None:
        super().__init__()
        self.in_features = in_features
        self.out_features = out_features

        # Initialize weights and biases
        init_factor = 0.01
        self.W = Parameter(np.random.randn(in_features, out_features) * init_factor)
        self.b = Parameter(np.zeros((1, out_features)))

        # Cache for backward pass
        self.x = None

    def forward(self, x):
        # 學生實作部分：reutrn output of linear layer

        self.x = x
        # 執行矩陣乘法 XW + b
        # x.shape: (N, in_features), W.data.shape: (in_features, out_features)
        # output.shape: (N, out_features)
        output = np.dot(x, self.W.data) + self.b.data # y = wx+b
        return output

    def backward(self, dy):
        # 學生實作部分：return gradient w.r.t. input and compute gradients for weights and biases

        # 計算對權重W的梯度(dL/dW)
        # x.T.shape: (in_features, N), dy.shape: (N, out_features)
        # dW.shape: (in_features, out_features)
        self.W.grad = np.dot(self.x.T, dy)

        # 計算對偏置 b 的梯度 (dL/db)
        # 因為b對於batch中的每個樣本都起作用，所以要將所有樣本的梯度加總
        # dy.shape:(N, out_features)->sum over N (axis=0) -> (1, out_features)
        self.b.grad = np.sum(dy, axis=0, keepdims=True)

        # 計算要傳回給前一層的梯度 (dL/dX)
        # 為了維度，我們需要使用W的轉置
        # dy.shape: (N, out_features), W.data.T.shape: (out_features, in_features)
        # dx.shape: (N, in_features)
        dx = np.dot(dy, self.W.data.T)

        return dx
```

- 架構實作(Coding) — Activation Layer(ReLU)

```
class ReLU(Module):
    def __init__(self) -> None:
        super().__init__()
        self.x = None

    def forward(self, x):
        # 學生實作部分：return output of ReLU activation
        self.x = x
        return np.maximum(0, x) # max(0,x)

    def backward(self, dy):
        # 學生實作部分：return gradient w.r.t. input

        mask = (self.x > 0) # 建立一個mask
        # 將 dy 與遮罩相乘，只保留x>0的地方的梯度
        return dy * mask

    def __repr__(self) -> str:
        return f'{self.__class__.__name__}()'
```

- Cross Validation－採取10%分割

```python
def load_mnist_data(
    root="/content/drive/MyDrive/Colab-AILAB/LAB1", batch_size=1, split_ratio=0.1, transform=None
) -> tuple[DataLoader, DataLoader, DataLoader]:
    def _split_dataset(dataset, split_ratio):
        # 學生實作部分：split dataset into training and validation sets
        # hint: return Subset(dataset, train_indices), Subset(dataset, valid_indices)

        dataset_size = len(dataset)
        # 產生一個從 0 到 (dataset_size - 1) 的索引列表
        indices = list(range(dataset_size))
        # 計算驗證集的樣本數量
        split_point = int(np.floor(split_ratio * dataset_size))
        # 為了讓每次執行的結果可重現，可以設定一個隨機種子 np.random.seed()
        np.random.shuffle(indices)
        # 5. 分割索引列表
        valid_indices = indices[:split_point]
        train_indices = indices[split_point:]
        # hint: return Subset(dataset, train_indices), Subset(dataset, valid_indices)

        return Subset(dataset, train_indices),Subset(dataset, valid_indices)
        #這裡會測試切10%跟20%的結果進行比對
```

- 訓練過程

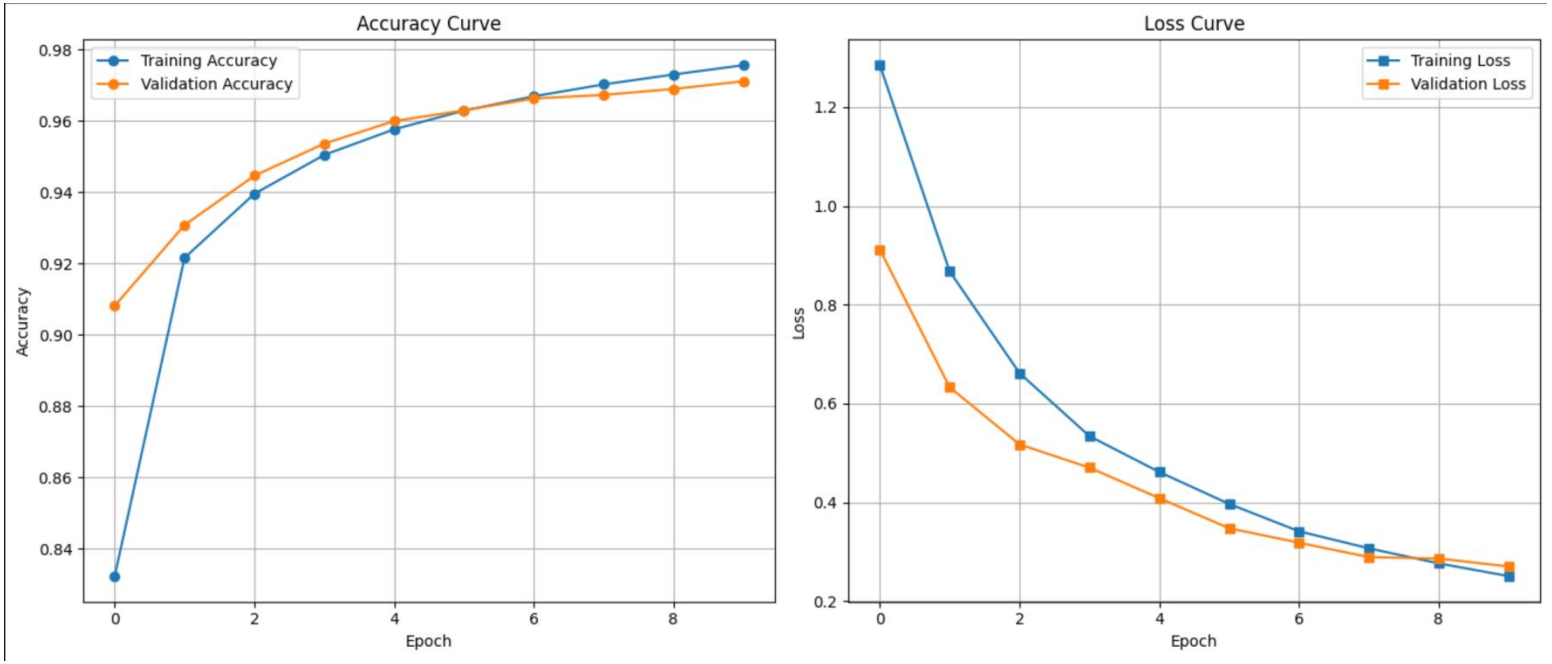
```
#在開始新一輪計算前，清除上一輪的梯度
optimizer.zero_grad()
#forward propagation
y_pred = model(x) #將圖片數據x餵給模型，得到模型的預測輸出 y_pred

#compute loss
loss = criterion(y_pred, y) #loss function criterion，比較預測值y_pred和真實y 計算Loss

#compute accuracy
total_loss += loss * x.shape[0] #loss 是平均值，乘以 batch size 還原總和

predicted_labels = np.argmax(y_pred, axis=1) #找出模型預測的類別 (0-9)
true_labels = np.argmax(y, axis=1) #找出真實的類別 (0-9)

total += y.shape[0] #累加處理過的樣本總數
correct += (predicted_labels == true_labels).sum() #累加預測正確的樣本數

# backward propagation
initial_grad = criterion.backward()#計算初始梯度
model.backward(initial_grad)#將梯度傳回模型，計算模型中所有權重的梯度

# update parameters
optimizer.step()#更新模型的權重
```

利用前面已經建立好的架構進行forward訓練
再進行backward的計算梯度去更新權重已達到更好的結果

# 實驗結果展示與討論

固定epoch10來訓練，去調整learning rate數值以及切多少資料集當validation set觀察結果

```
epoch 9: train_loss = 0.25051297409444717, train_acc = 0.9756666666666667
100%|██████████████| 6000/6000 [00:00<00:00, 9846.79it/s]epoch 9: valid_loss = 0.26993248749074567, valid_acc = 0.9711666666666666
```

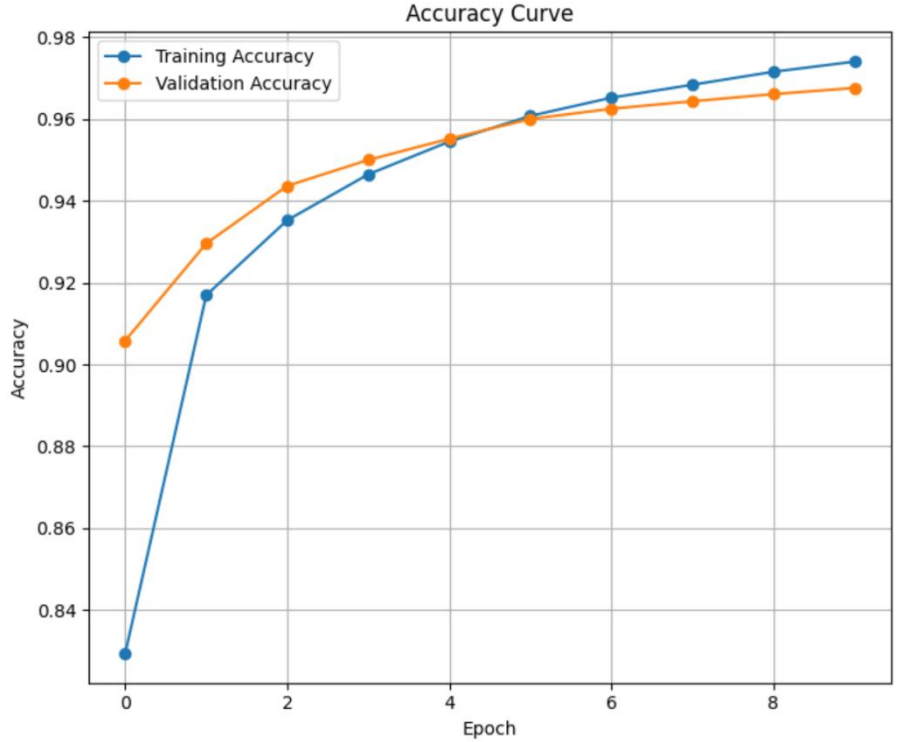


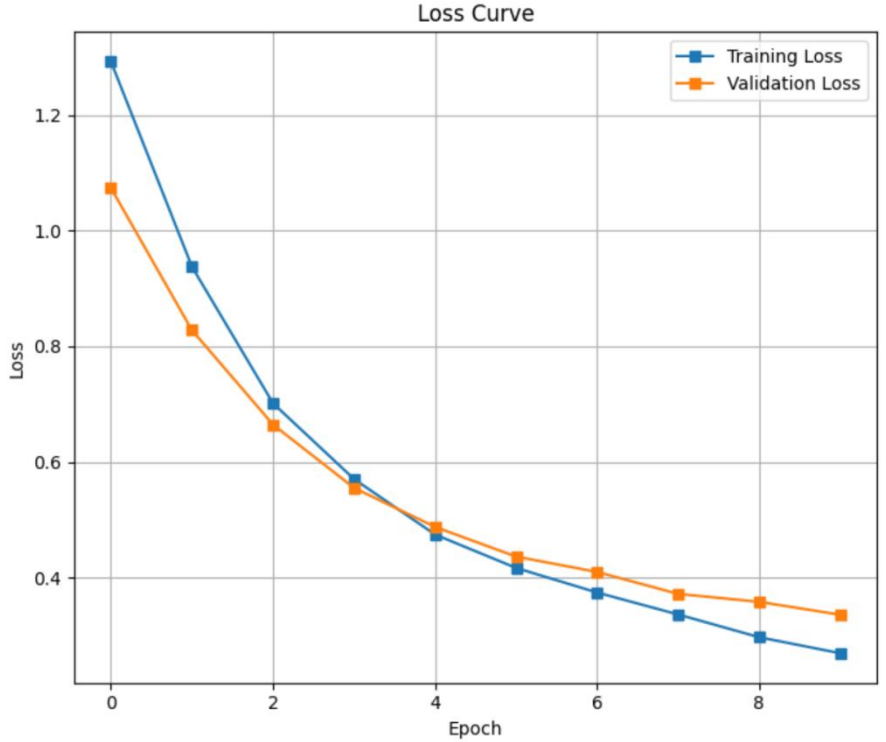

以目前測試的結果發現

epoch=10、lr調整為0.001(固定)、切10%作為validation set 結果最好

# 實驗結果展示與討論

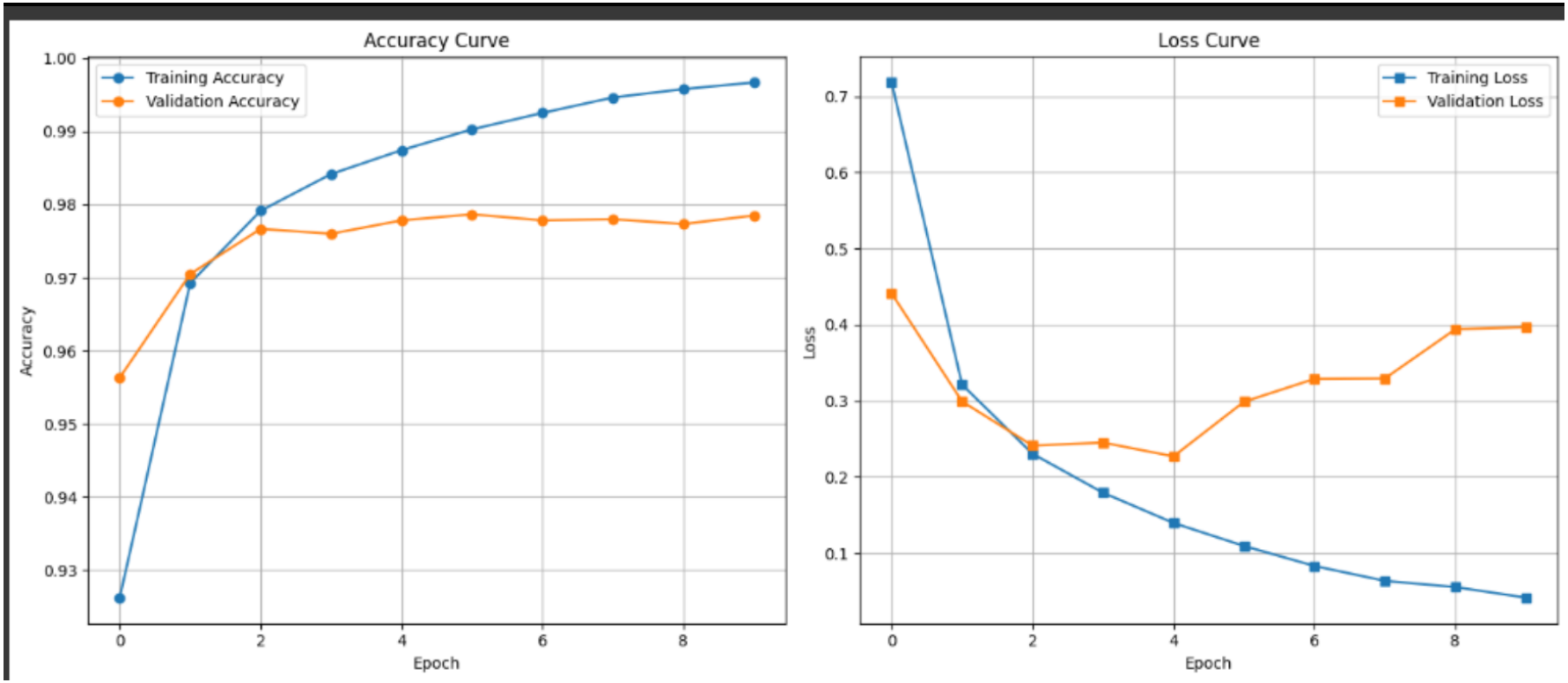
```
epoch 9: train_loss = 0.26946484317091973, train_acc = 0.9740208333333333
100%|██████████| 12000/12000 [00:01<00:00, 10015.74it/s]epoch 9: valid_loss = 0.33600235352638486, valid_acc = 0.9675833333333334
```



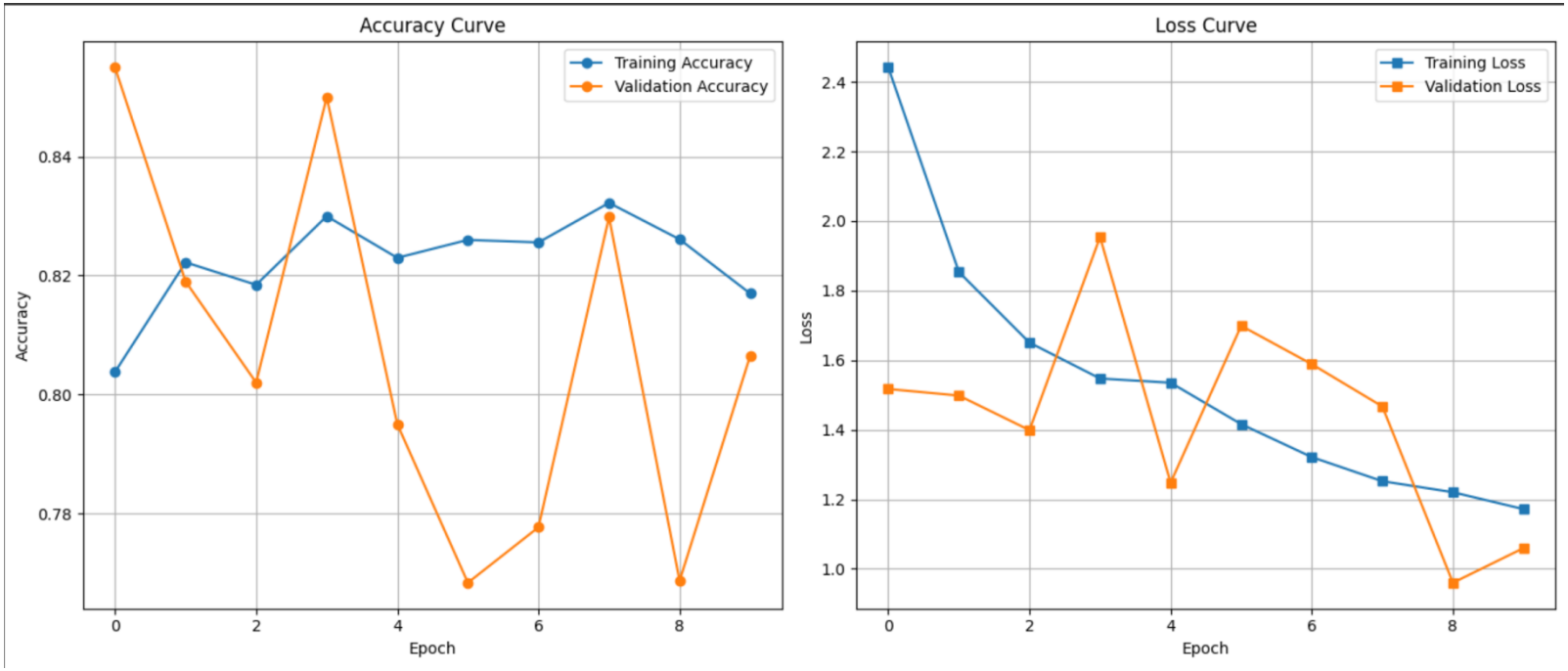

epoch=10 lr=0.001 切20%作為validation set
可以看到結果上並沒有差異太大，但準確率0.974比0.975低了0.001
所以後續測試皆是切10%作為validation set

- 實驗結果展示與討論




epoch=10 調整lr=0.01 切10%作為validation set
從這個結果可以看到，lr調整成為0.01之後，一開始收斂非常快，
但後面的結果並沒有超越lr=0.001(不過也是差距很相近)
有趣的是Loss Curve中，validation loss 反而越來越高

# 實驗結果展示與討論




```
epoch 9: train_loss = 1.1719057061672407, train_acc = 0.8169814814814815
100%|██████████| 6000/6000 [00:00<00:00, 8885.10it/s]epoch 9: valid_loss = 1.059816815141852, valid_acc = 0.8065
```

epoch=10 調整lr=0.1 切10%作為validation set
從圖表可以看到，雖然一開始收斂的非常快
甚至還有震盪的問題，最後的Training ACC也只有0.81，非常低

# AILAB-LAB1
# TASK2-CIFAR10

# 資料前處理

使用sample code提供的Normalization

```
data = next(iter(dataloader))[0]     # get the first iteration data
mean = np.mean(data.numpy(), axis=(0,2,3)) #計算RGB通道的平均亮度
std = np.std(data.numpy(), axis=(0,2,3))  #計算pixel和mean的偏離程度，偏離越多表示對比度越高
return mean, std
```

Data Augmentation:

```
transforms.RandomCrop(32,padding=4),            #  先在圖片四周各填充4個像素，再從中隨機裁切出  32x32  的區域
transforms.RandomHorizontalFlip(),
transforms.ToTensor(),  #  Transform  to  tensor  &  image  normalization中的"Min/Max  Normalization",將range縮在0~1  &  Dimension  Reordering
#transforms.RandomErasing(p=0.5,  scale=(0.02,  0.33),  ratio=(0.3,  3.3),  value=0),
#模擬「遮擋」的情況。它會在圖片上隨機選擇一個矩形區域，並用隨機值（例如黑色或雜訊）來填充它。這強迫模型不要只依賴單一的顯著特徵，而是學習從物體的不同部分來辨識。
transforms.Normalize(mean=train_mean,  std=train_std),  #  Normalization(Standardization)
```

使用了最基本的RandomCrop和RandomHorizontalFlip進行data augmentation
(有嘗試過RandomErasing嘗試加雜訊給他，不過最後訓練結果並沒有提升)

1.RandomCrop是將圖片 的四周填充像素，再隨機切出nxn的區域

2.RandomHorizontalFlip是將圖片 進行一次隨機水平翻轉(左右顛倒)(50%機率)

3.RandomErasing是將圖片 中隨機選擇一個區域，加雜訊在那邊讓圖片 更模糊
(不過因為訓練完後結果並沒有提升，所以我已經將它註解掉了)

# 參數調整

1.Batch size測試 32/128/256，以我最終結果來看，Batch Size設為128結果最好

2.Epoch皆固定50

3.Learning Rate測試0.1/0.01/0.001，並再作一個固定lr以及變動lr去做對比

```
#lr_scheduler,動態調整lr
from  torch.optim.lr_scheduler  import  StepLR
optimizer  =  optim.Adam(model.parameters(),  lr=0.001)
scheduler  =  StepLR(optimizer,  step_size=10,  gamma=0.1)  #每10個epoch，學習率變為原來的0.1倍
```

有設計一個動態learning rate去更動，每10個epoch學習率會除以10去做調降
希望能依此讓ACC提高

# 架構實作

```python
class ResNet18(nn.Module):
    def __init__(self, num_classes=1000):
        super(ResNet18, self).__init__()
        # 學生實作部分：Define the ResNet-18 architecture using BasicBlock

        self.in_channels = 64 # 設定一個初始的輸入通道數，給第一個卷積層之後使用

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False) # 這一層負責接收 (3, 32, 32) 的圖片，輸出 (64, 32, 32) 的特徵圖
        self.bn1 = nn.BatchNorm2d(64)

        # ResNet-18 的每個 layer 都有 2 個 BasicBlock
        # Layer1: 輸入 (64, 32, 32) -> 輸出 (64, 32, 32)
        self.layer1 = self._make_layer(BasicBlock, 64, num_blocks=2, stride=1)

        # Layer2: 輸入 (64, 32, 32) -> 輸出 (128, 16, 16)
        # stride=2 讓圖片尺寸減半
        self.layer2 = self._make_layer(BasicBlock, 128, num_blocks=2, stride=2)

        # Layer3: 輸入 (128, 16, 16) -> 輸出 (256, 8, 8)
        self.layer3 = self._make_layer(BasicBlock, 256, num_blocks=2, stride=2)

        # Layer4: 輸入 (256, 8, 8) -> 輸出 (512, 4, 4)
        self.layer4 = self._make_layer(BasicBlock, 512, num_blocks=2, stride=2)

        #分類層
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1)) # 將 (512, 4, 4) 池化成 (512, 1, 1)
        self.linear = nn.Linear(512 * BasicBlock.expansion, num_classes) # 全連接層

    def _make_layer(self, block, out_channels, num_blocks, stride):
        # 學生實作部分：Define make_layer function to create layers of blocks

        strides = [stride] + [1] * (num_blocks - 1)
        layers = [] #準備一個空列表來存放要建立的所有block層

        # 根據 stride 列表，依序建立每一個 block
        for s in strides:
            layers.append(block(self.in_channels, out_channels, s))
            self.in_channels = out_channels * block.expansion

        return nn.Sequential(*layers) #所有 block 層打包成一個單一的模組
```

```python
def forward(self, x):
    # 學生實作部分：Define the forward pass of ResNet-18
    out = F.relu(self.bn1(self.conv1(x)))

    #四層layer
    out = self.layer1(out)
    out = self.layer2(out)
    out = self.layer3(out)
    out = self.layer4(out)

    #Classifier
    out = self.avgpool(out) #pooling
    out = out.view(out.size(0), -1) #flat
    out = self.linear(out) #Pass Fully-Connected layer

    return out
```

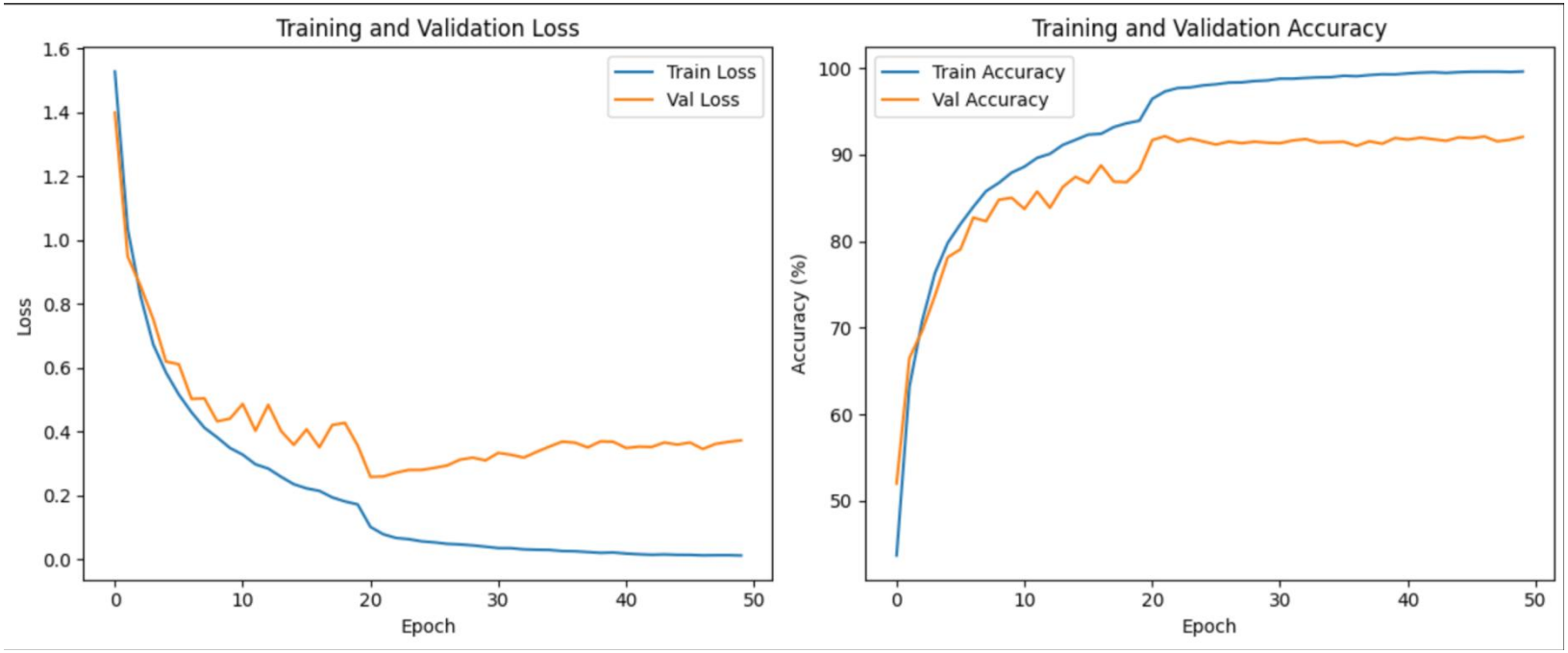
```python
class BasicBlock(nn.Module):
    expansion = 1 #後面ResNet會用到,用來表示這個 block會把輸出的通道數放大幾倍。,ResNet18通常=1
    def __init__(self, in_channels, out_channels, stride=1):
        super(BasicBlock, self).__init__()
        # 學生實作部分：Define the two convolutional layers and the shortcut connection

        # 第一個Convolution layer
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3, stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels) # Batch Normalization

        # 第二個Convolution layer
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)

        # Shortcut Connection(實線部分)
        self.shortcut = nn.Sequential() # 先建立一個空的 Sequential

        #shortcut,虛線部分(若大小不一樣進行架構轉換)
        if stride != 1 or in_channels != self.expansion * out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, self.expansion * out_channels, kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(self.expansion * out_channels)
            )

    def forward(self, x):
        # 學生實作部分：Define the forward pass using convolutional layers and the shortcut connection
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out
```

# 實驗結果展示與討論

固定epoch=50來訓練，去調整learning rate數值與batch size去觀察結果的變化

參數為 epoch=50 batch size=128 初始lr=0.01 動態調降

```
Epoch [50/50] Train Loss: 0.0121 | Train Acc: 99.63% | Val Loss: 0.3727 | Val Acc: 92.08%
```




以目前測試的結果發現

Learning初始為0.01，並且每10epoch去除以10降低lr的結果為最好

最高測得出Train ACC=99.63%

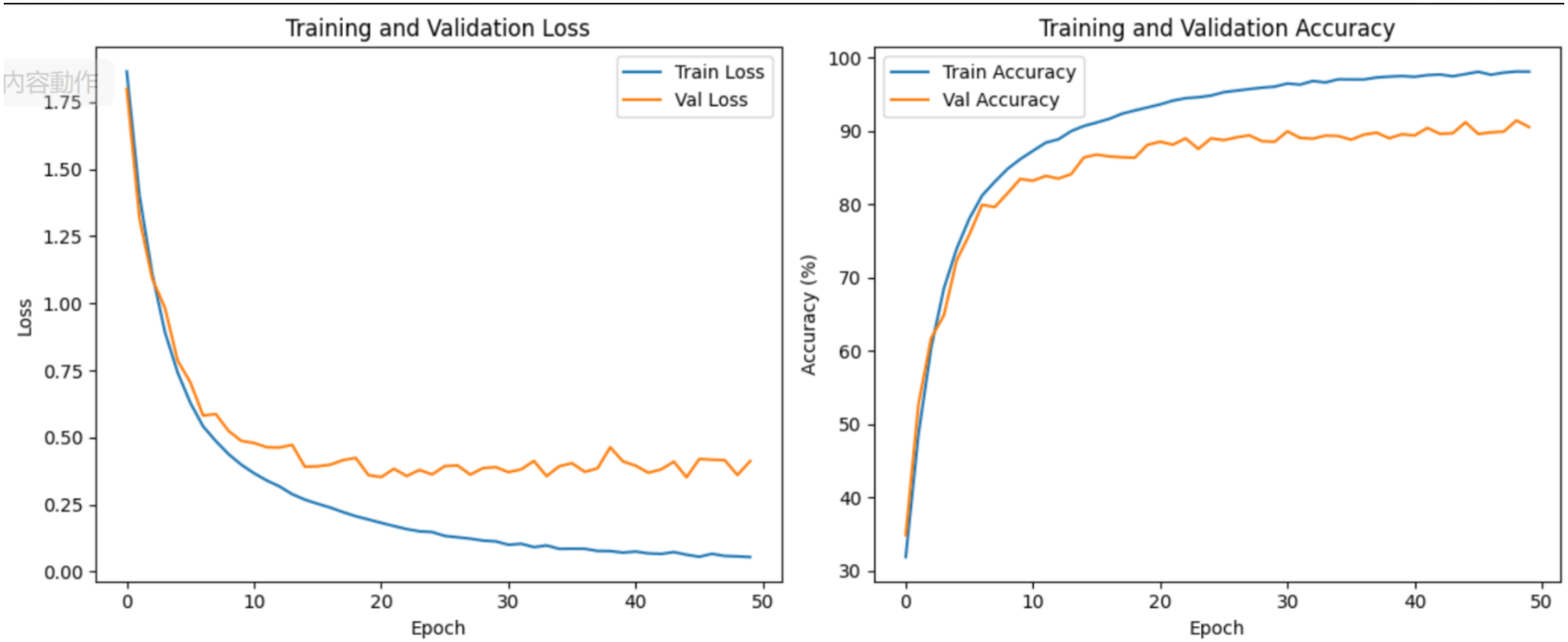
*小記*:我有試過初始0.1lr動態調降，雖然收斂的非常快，不過結果並沒有預期的好，我認為是我每10epoch才去調降太慢了，可能要調整成每5epoch調降結果會好一點(因為沒存到訓練完的圖片只好用文字敘述)

# 實驗結果展示與討論

固定epoch=50來訓練，去調整learning rate數值與batch size去觀察結果的變化

參數為 epoch=50 batch size=128 固定lr=0.01

```
Train Loss: 0.0545 | Train Acc: 98.07%
```



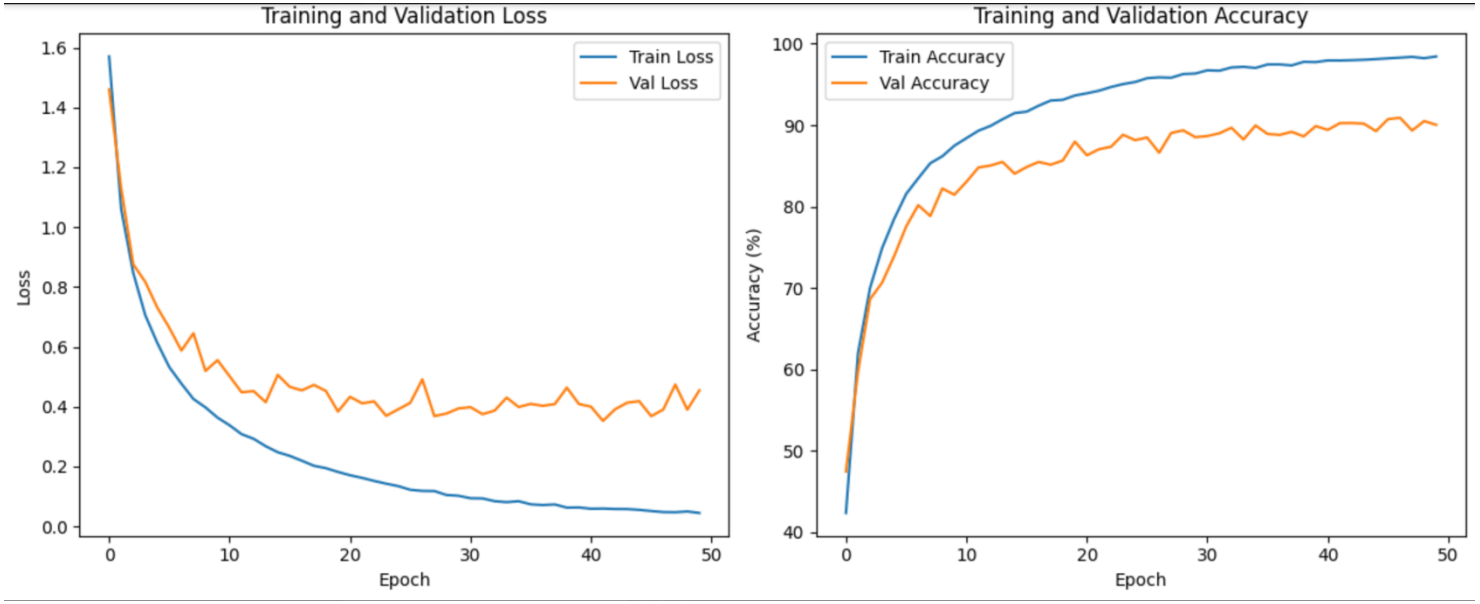

以這個結果來看，ACC表現得很不錯，不過沒有動態調降lr來的好
ACC為98.07%

# 實驗結果展示與討論

固定epoch=50來訓練，去調整learning rate數值與batch size去觀察結果的變化

參數為 epoch=50 batch size=256 初始lr=0.01 動態調降

```
Train Loss: 0.0444 | Train Acc: 98.43%
```



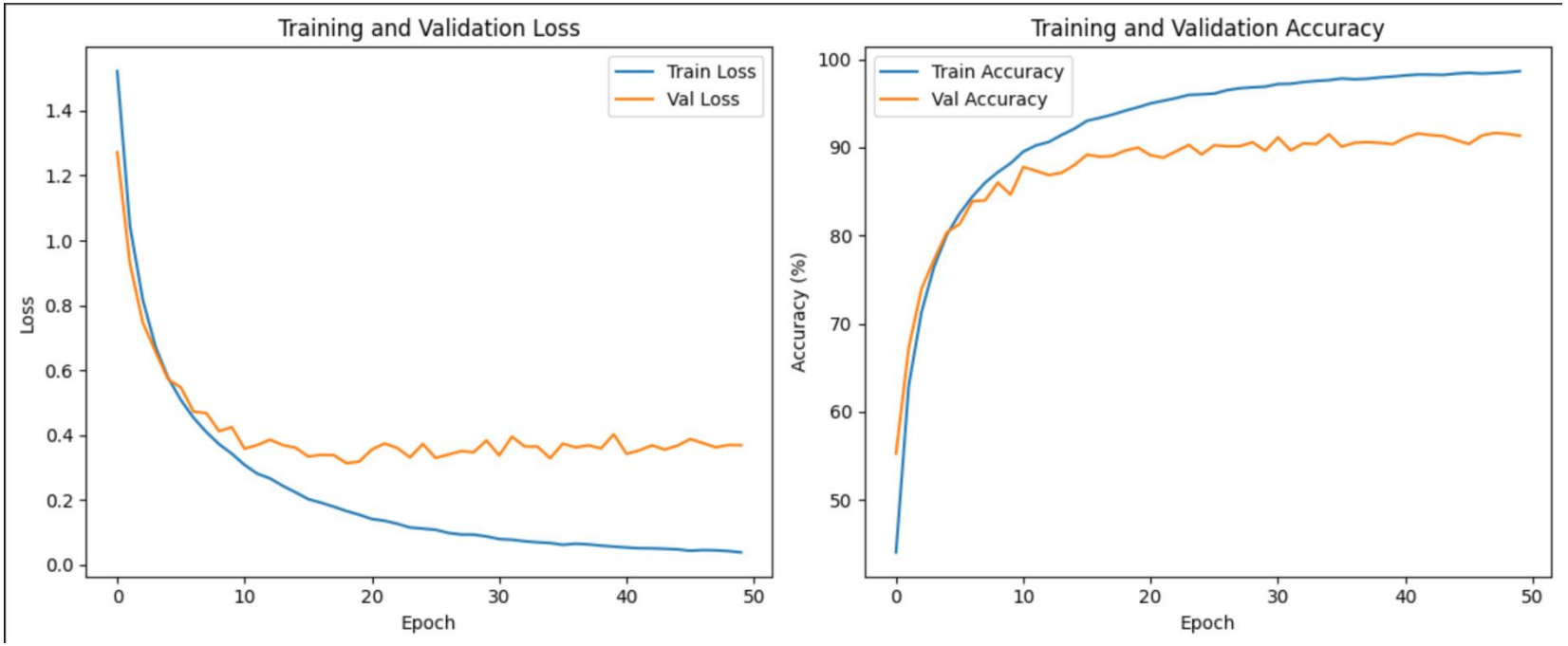

在單epoch訓練時，訓練速度比batch size=128來的快
測試下來結果沒有比batch size=128來的好(99.63% vs 98.43%)

# 實驗結果展示與討論

固定epoch=50來訓練，去調整learning rate數值與batch size去觀察結果的變化

參數為 epoch=50 batch size=32 初始lr=0.01 動態調降

```
Train Loss: 0.0383 | Train Acc: 98.65%
```



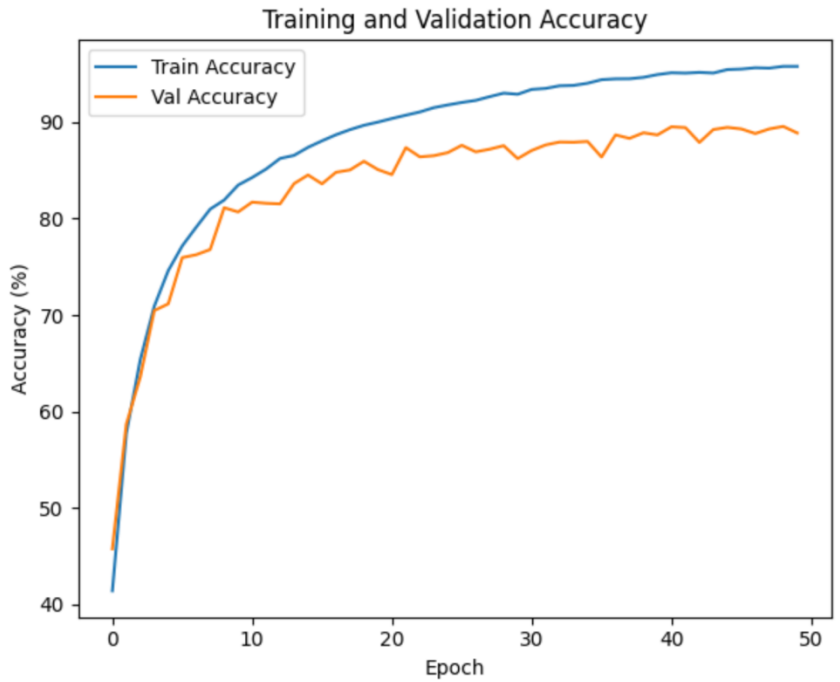

在單epoch訓練時，訓練速度非常的慢
而且從這個圖表可以看到，線抖動的更明顯
值得一提的是我原本以為ACC能更高(畢竟batch低的話generalized能力更好)但只有98.65%
不過已經得到第二高的ACC了

# 實驗結果展示與討論

固定epoch=50來訓練，去調整learning rate數值與batch size去觀察結果的變化

參數為 epoch=50 batch size=128 初始lr=0.01 動態調降 ，額外添加雜訊來訓練

```
Train Loss: 0.1199 | Train Acc: 95.82% | Val Loss: 0.3694 | Val Acc: 88.90%
```


拿前面訓練出來最高的結果再多一個資料處理的步驟: RandomErasing
嘗試加了一些雜訊讓他對資料不會有過度偏頗依賴
不過最後的結果並不如預期，在程式中也註解掉，ACC只得95.82%

# 訓練遇到的困難

1.純粹只聽架構與實際實作programming的困難(有許多function要熟悉)

2.不斷嘗試尋找適合的參數，調整後觀察得出最好的結果，到底該如何找到最適合的參數也是一件很困難的事情

3.在TASK1的MNIST的訓練中

原本整個寫完之後訓練的成果只有0.1ACC，甚至發生從第二個epoch開始ACC就完全沒有變動，後來才發現我transform()這個function沒有去修改到

```
def transform(x):  #Image Normalization
    """將像素值從 0-255 的範圍映射到 0.01-1.0 的範圍"""
    x_normalized = x / 255.0#將0~255的像素值正規化到[0,1] range
    return x_normalized * 0.99 + 0.01 #Normalization
```

那時候才發現到我只有return x*0.99+0.01，但假設x是255的話，255*0.99+0.01=252.46，這根本沒有做任何的Normalization，最後才導致梯度爆炸(他根本沒有學到任何的東西)，所以先對他做/255讓他落在[0,1]後，在return x*0.99+0.01就能夠正常的學習了