深度視覺 HW4

M113040064 李冠宏

Data Preparation and Visualization

首先是一些先前工作，例如引入資料夾及其檔案，引入套件並使用 GPU 來做訓練，並載入 dataset。這次做的是 cifar10，因此會有 10 個 classes。

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/MyDrive/HW4/')
# You may change `MODEL_DIR` to a existing local directory if you are
not using Colab
MODEL_DIR = '/content/'
MODEL_PATH = MODEL_DIR + 'model.pt' if MODEL_DIR.endswith('/') else
MODEL_DIR + '/model.pt'
from _utils import load_data, Dataset, Dataloader, FCL_Tests,
ReLU_Tests, SCE_Tests, plot_curves, plot_result
import numpy as np
import matplotlib.pyplot as plt
import random
import torch
if torch.cuda.is_available():
  DEVICE = 'cuda'
  print(f'Using torch {torch.__version__}, device =
{torch.cuda.get_device_name(0)}')
else:
  DEVICE = 'cpu'
  print(f'Using torch {torch.__version__}, device = cpu')
images, labels, classes = load_data()
print(f'Data type of images: {images.dtype} (value range:
[{images.min()}, {images.max()}])')
print(f'Shape of images: {images.shape}')
print(f'Shape of labels: {labels.shape}')
print(f'Number of classes: {len(classes)}')
print(f'Classes: {classes}')
```

```
Data type of images: uint8 (value range: [0, 255])
Shape of images: (50000, 32, 32, 3)
Shape of labels: (50000,)
Number of classes: 10
Classes: ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']
```

這邊是測試 dataset 是否有被正確載入進來，並做視覺化。

```
# Display samples of each class
plt.rcParams['figure.figsize'] = (16, 16) #設定圖片尺寸
num_classes = len(classes) # 10
num_example = 10 #要列出每一個 class 的數量
for label, class_name in enumerate(classes):
  idxs = np.flatnonzero(labels == label)
  idxs = np.random.choice(idxs, num_example, replace=False)
  for i, idx in enumerate(idxs):
    plt_idx = i * num_classes + label + 1
    plt.subplot(num_example, num_classes, plt_idx)
    plt.imshow(images[idx]); plt.axis('off')
    if i == 0:
      plt.title(class_name)
plt.show()
```

接下來將資料切割為 training data，validation data 和 test data，比例是 60% : 20% : 20%

```
# 將資料切割為 training data, validation data, test data
# Data splitting with numpy array slicing
num_data = images.shape[0]
split_ratio = {'train': 0.6, 'valid': 0.2, 'test': 0.2}
num_valid_data = round(num_data * split_ratio['valid'])
num_test_data = round(num_data * split_ratio['test'])
num_train_data = num_data - (num_valid_data + num_test_data)

#設定 training data
start_idx = 0
train_dataset = Dataset(images[start_idx:start_idx+num_train_data],
labels[start_idx:start_idx+num_train_data])
start_idx += num_train_data

#設定 valid data
valid_dataset = Dataset(images[start_idx:start_idx+num_valid_data],
labels[start_idx:start_idx+num_valid_data])
```

```
start_idx += num_valid_data

#設定 test data
test_dataset = Dataset(images[start_idx:start_idx+num_test_data],
labels[start_idx:start_idx+num_test_data])

print(f'Number of training data: {len(train_dataset)}')
print(f'Number of validation data: {len(valid_dataset)}')
print(f'Number of test data: {len(test_dataset)}')
```

執行結果：

```
Number of training data: 30000
Number of validation data: 10000
Number of test data: 10000
```

Layer Definition

接下來是定義全連接層、激勵函數(ReLU)和損失函數(Cross Entropy Softmax)

首先是全連接層，它的 forward 主要是要將 input 乘上權重，並加上一個偏移量 bias，因此我們利用 torch.matmul 函式將 tensor 做相乘再用 torch.add 將偏移量加上去。而 backward 的部分，算偏移量梯度時，偏移量的偏微分部分會變成常數 1，因此算出平均即可，而權重的偏微分會變成 input x 的轉置乘上 backprop 再算平均，以得到權重的梯度

```
class FullyConnectedLayer(object):
  def __init__(self, device='cpu'):
    self.dv = device

  def forward(self, x, w, b):
    N, D = x.shape #input 為 N*D
    Dw, F = w.shape #第一層權重的維度為 D*H
    assert Dw == D, f'Wrong shape of weights, expected ({D},{F}) but
got ({Dw},{F})'
    Fb = b.shape[0] #bias 的維度為 H
    assert Fb == F, f'Wrong shape of bias, expected ({F},) but got
({Fb},)'
    out = torch.zeros((N, F)).to(self.dv)
```

```
    ####################
    # TODO:
    # 1. Implement fully connected layer with `w` * `x` + `b` and store
the result in variable `out`
    ####################
    # -----START OF YOUR CODE-----
    out = torch.add(torch.matmul(x, w), b)
    # ------END OF YOUR CODE------

    self.cache = (x, w, b, out)
    return out

  def backward(self, backprop):
    N, F = backprop.shape
    dw = None
    db = None
    # print("backprop:",backprop) # tensor([[0.5759, 0.9293, 0.3186,
0.6674], [0.1318, 0.7163, 0.2894, 0.1832], [0.5865, 0.0201, 0.8289,
0.0047]], dtype=torch.float64)
    # print("backprop.shape:", backprop.shape) #[3, 4]

    ####################
    # TODO:
    # 1. Derive the gradient of bias, i.e., d(out)/d(b)
    # 2. Multiply d(out)/d(b) with `backprop` due to chain rule, and
then store the AVERAGED result in variable `db`
    # 3. Then, derive the gradient of weights, i.e., d(out)/d(w)
    # 4. Multiply d(out)/d(w) with `backprop` due to chain rule, and
then store the AVERAGED result in variable `dw`
    # HINT: Shapes of `db` and `dw` should be (F,) and (D, F) respectively
    ####################
    # -----START OF YOUR CODE-----
    x, w, b, out = self.cache
    db = torch.mean(backprop, dim=0)
    dw = torch.div(torch.matmul(torch.transpose(x, 0, 1), backprop),
x.shape[0])
    # ------END OF YOUR CODE------
```

```
    return dw, db
```

測試的結果為通過

```
layer = FullyConnectedLayer()
FCL_Tests(layer)
```

```
Results of fully connected layer forward and backward tests: All passed.
```

接下來是激勵函數(ReLU)，激勵函數的主要目的是要使得分類從線性變成非線性，而 ReLU 是將小於 0 的 input 都設為 0，其他的維持一樣，也就是 ReLU(x) = max(0, x)，因此 forward 的部分是將小於 0 的透過 torch.clamp 都設為 0 即可。Backward 的部分則是算出 ReLU 的偏微分，他的偏微分是若 input 小於 0 則輸出為 0，若 input 大於 0 則輸出為 1，因此我們把 out 的非 0 部分都改成 1 並乘上 backprop 來反向傳遞。

```
class ReLU(object):
  def __init__(self):
    pass

  def forward(self, t):
    out = t.clone()
    ####################
    # TODO:
    # 1. Implement ReLU on `t` and store in variable `out`
    ####################
    # -----START OF YOUR CODE-----
    out = torch.clamp(t, min=0.0)
    # ------END OF YOUR CODE------
    # print("t:",t)
    # print("out:", out)
    self.cache = (t, out)

    return out

  def backward(self, backprop):
    dt = torch.zeros_like(backprop)

    ####################
    # TODO:
```

```
    # 1. Derive d(out)/d(t)
    # 2. Multiply d(out)/d(t) with `backprop` due to chain rule, and
then store the result in variable `dt`
    ####################
    # -----START OF YOUR CODE-----
    t, out = self.cache
    out[out != 0] = 1
    dt = out * backprop
    # ------END OF YOUR CODE------

    return dt
```

測試的結果為通過

```
layer = ReLU()
ReLU_Tests(layer)
```

```
Results of ReLU function tests: All passed.
```

最後是 loss function 損失函數 cross entropy softmax，forward 的部分，首先實作出 softmax 函式，算出每個類別可能的機率，接下來再實作 cross entropy，透過機率來算出 loss。Backward 的部分則是算出 cross entropy softmax 的偏微分並計算平均。

```
class Softmax_CrossEntropy(object):
  def __init__(self):
    pass

  def forward(self, y, net_out):
    N, F = net_out.shape
    prob = torch.zeros_like(net_out)
    # print("net_out:",net_out)
    # print("net_out.shape:",net_out.shape)

    ####################
    # TODO:
    # 1. Implement Softmax and store the result in variable `prob`
    # HINT: Shape of `prob` should be (N, F)
    ####################
    # -----START OF YOUR CODE-----
```

```
    prob = torch.exp(net_out) / torch.sum(torch.exp(net_out),
dim=1).view(-1, 1)
    # ------END OF YOUR CODE------

    self.cache = (y, prob)
    pred_y = prob.argmax(dim=1) #我們所猜測的值 #tensor[1,0,0,1,0]

    y_one_hot = torch.zeros_like(net_out) #(5,2)
    y_one_hot[np.arange(N), y] = 1
    loss = torch.zeros(1).to(DEVICE)

    ####################
    # TODO:
    # 1. Implement cross-entropy loss and store the result in variable
`loss`
    ####################
    # -----START OF YOUR CODE-----
    loss = torch.div(-torch.sum(y_one_hot * torch.log(prob)),
prob.shape[0])
    # ------END OF YOUR CODE------

    return pred_y, loss

  def backward(self):
    y, grad = self.cache
    N, F = grad.shape
    # print("grad.shape:",grad.shape)
    # print("y.shape:",y.shape)
    # print("grad:",grad)
    # print("y:",y)

    ####################
    # TODO:
    # 1. Derive the gradient of `net_out` and store the result in
variable `grad`
    ####################
    # -----START OF YOUR CODE-----
    grad[range(y.shape[0]), y] -= 1
```

```
    grad = torch.div(grad, y.shape[0])
    # ------END OF YOUR CODE------

    return grad
```

測試的結果為通過

```
layer = Softmax_CrossEntropy()
SCE_Tests(layer)
```

```
Results of softmax and cross entropy forward and backward tests: All passed.
```

Network Definition

這個階段為訓練模型，透過剛剛定義好的類別來經過全連接層、激勵函數和用損失函數來計算 loss，並不斷的 forward 再 backward 來調整參數(權重及偏移量)。每經過一個 batch，就會將權重和偏移量減去 learning rate * 梯度來更新。

```
class Classifier(object):
  def __init__(self, dim_in, dim_hidden, dim_out, device='cpu'):
    self.dv = device
    np.random.seed(0)

    #初始化參數
    self.params = {'w1': torch.from_numpy(np.random.randn(dim_in, 
dim_hidden)*0.8).to(self.dv),
            'b1': torch.zeros(dim_hidden).to(self.dv),
            'w2': torch.from_numpy(np.random.randn(dim_hidden, 
dim_out)*0.8).to(self.dv),
            'b2': torch.zeros(dim_out).to(self.dv)}

    #定義 layer
    layer1 = FullyConnectedLayer() #layer1
    activation = ReLU() #activation function
    layer2 = FullyConnectedLayer() #layer2
    self.net = [layer1, activation, layer2] #the layer list
    self.loss_func = Softmax_CrossEntropy() #loss function

  #儲存模型參數
  def save_model(self, path):
    torch.save(self.params, path)
```

```
  #載入模型參數
  def load_model(self, path):
    checkpoint = torch.load(path, map_location=self.dv)
    self.params = checkpoint

  #訓練 model
  def training(self, train_dataloader, valid_dataloader, epochs=200,
learning_rate=1e-3, learning_rate_decay=0.95, verbose=1):
    print(f'Training with
batch_size={train_dataloader[0][0].shape[0]},
hidden_dim={self.params["w1"].shape[1]},
learning_rate={learning_rate},
learning_rate_decay={learning_rate_decay}')
    train_loss_log = []
    train_acc_log = []
    valid_loss_log = []
    valid_acc_log = []
    best_val_loss = None

    for epoch in range(epochs):
      if verbose > 0 and (epoch+1) % verbose == 0:
        print(f'Epoch {epoch+1}:')

      single_epoch_log = ''
      train_loss = 0.0
      train_corr = 0.0; train_eval = 0.0

      for batch_idx in range(len(train_dataloader)):
        ### Model Input and Loss Calculation ###
        batch_data = train_dataloader[batch_idx]
        batch_img, batch_label = batch_data
        layer1_out =
self.net[0].forward(torch.from_numpy(batch_img).to(self.dv),
self.params['w1'], self.params['b1']) #layer1
        layer1_out = self.net[1].forward(layer1_out) #activation
function (ReLU)
```

```
        layer2_out = self.net[2].forward(layer1_out,
self.params['w2'], self.params['b2']) #layer2
        pred_y, loss = self.loss_func.forward(batch_label, layer2_out)
#loss function
        train_loss += loss.cpu().numpy()
        train_corr += torch.count_nonzero(pred_y ==
torch.from_numpy(batch_label).to(self.dv)).cpu().numpy()
        train_eval += batch_img.shape[0]
        ### Backpropagation and Optimization ###
        params_grad = {}
        dloss_dout2 = self.loss_func.backward() #loss function
        dout2_dw2, dout2_db2 = self.net[2].backward(dloss_dout2)
#layer2
        params_grad['w2'] = dout2_dw2; params_grad['b2'] = dout2_db2
#更新參數之梯度
        dout2_dout1 =
self.net[1].backward(dloss_dout2.mm(self.params['w2'].T))
#activation function
        dout1_dw1, dout1_db1 = self.net[0].backward(dout2_dout1)
#layer1
        params_grad['w1'] = dout1_dw1; params_grad['b1'] = dout1_db1
#更新參數之梯度

        ####################
        # TODO:
        # 1. Update model parameters, i.e., `self.params`, by gradient
descent
        ####################
        # -----START OF YOUR CODE-----
        self.params['w1'] -= learning_rate * params_grad['w1']
        self.params['b1'] -= learning_rate * params_grad['b1']
        self.params['w2'] -= learning_rate * params_grad['w2']
        self.params['b2'] -= learning_rate * params_grad['b2']
        # ------END OF YOUR CODE------

      train_loss_log.append(train_loss/train_eval)
      train_acc_log.append(100*train_corr/train_eval)
```

```
    single_epoch_log += f'Training accuracy:
{format(train_acc_log[-1], ".2f")}%, loss:
{format(train_loss_log[-1], ".4f")}\n'
    train_dataloader.shuffle()

    valid_loss = 0.0
    valid_corr = 0.0; valid_eval = 0.0
    for batch_idx in range(len(valid_dataloader)):
      ### Model Input and Loss Calculation ###
      batch_data = valid_dataloader[batch_idx]
      batch_img, batch_label = batch_data
      layer1_out =
self.net[0].forward(torch.from_numpy(batch_img).to(self.dv),
self.params['w1'], self.params['b1']) #full connected layer1
      layer1_out = self.net[1].forward(layer1_out) # activation
function: ReLU
      layer2_out = self.net[2].forward(layer1_out,
self.params['w2'], self.params['b2']) #full connected layer2
      pred_y, loss = self.loss_func.forward(batch_label,
layer2_out)
      valid_loss += loss.cpu().numpy()
      valid_corr += torch.count_nonzero(pred_y ==
torch.from_numpy(batch_label).to(self.dv)).cpu().numpy()
      valid_eval += batch_img.shape[0]
    valid_loss_log.append(valid_loss/valid_eval)
    valid_acc_log.append(100*valid_corr/valid_eval)
    single_epoch_log += f'Validaion accuracy:
{format(valid_acc_log[-1], ".2f")}%, loss:
{format(valid_loss_log[-1], ".4f")}'
    if best_val_loss is None or valid_loss < best_val_loss:
      best_val_loss = valid_loss
      self.save_model(MODEL_PATH)
      if verbose == 1:
        single_epoch_log = '[MODEL SAVED]\n' + single_epoch_log
    if verbose > 0 and (epoch+1) % verbose == 0:
      print(single_epoch_log)
```

```
        learning_rate *= learning_rate_decay # update learning_rate every epoch

    return train_loss_log,train_acc_log,valid_loss_log,valid_acc_log

  def test(self, test_dataloader):
    total_corr = 0; total_eval = 0
    for batch_idx in range(len(test_dataloader)):
      ### Model Input ###
      batch_data = test_dataloader[batch_idx]
      batch_img, batch_label = batch_data
      layer1_out = self.net[0].forward(torch.from_numpy(batch_img).to(self.dv), self.params['w1'], self.params['b1'])
      layer1_out = self.net[1].forward(layer1_out)
      layer2_out = self.net[2].forward(layer1_out, self.params['w2'], self.params['b2'])
      pred_y, _ = self.loss_func.forward(batch_label, layer2_out)
      total_corr += torch.count_nonzero(pred_y == torch.from_numpy(batch_label).to(self.dv)).cpu().numpy()
      total_eval += batch_img.shape[0]
    print(f'Got {total_corr} correct prediction in {total_eval} test data, accuracy: {format((total_corr*1.0/total_eval)*100, ".2f")}%')
```

Searching for Best Hyperparameters

這個階段主要是要測試不同 hyperparameters 所帶來的訓練影響，變因有以下四個

Batch size	Learning rate	Learning rate decay	Hidden layer dimension

```
# Configurations
IN_DIM = np.prod(images[0].shape)
OUT_DIM = len(classes)
# Hyperparameters
BATCH_SIZE = 16
LR = 1e-3
LR_DECAY = 0.95
HIDEEN_DIM = 64
```

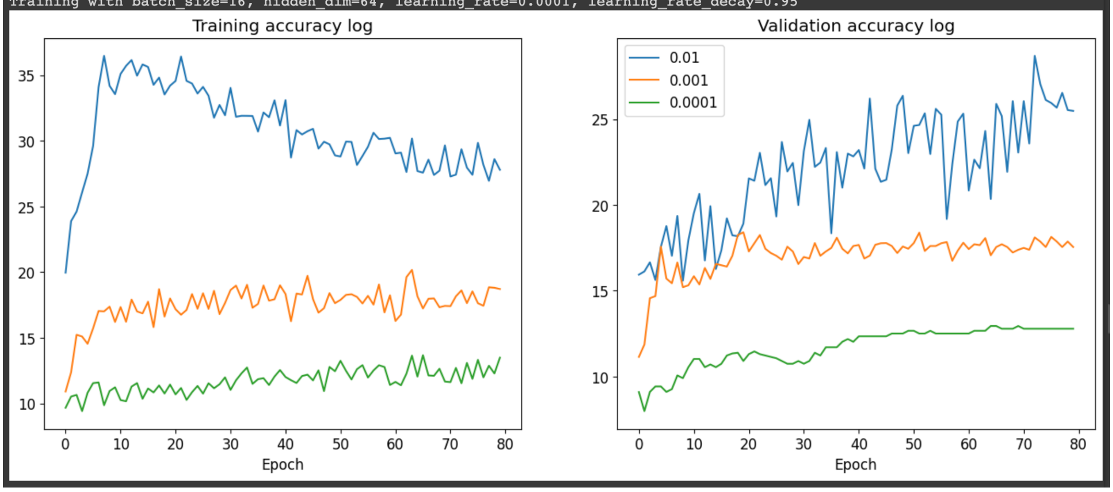
首先我們先對 learning rate 做測試

```
lr_cand = [1e-2, 1e-3, 1e-4]
train_acc = []
valid_acc = []

for lr in lr_cand:
  train_dataloader = Dataloader(train_dataset, 
batch_size=BATCH_SIZE)
  valid_dataloader = Dataloader(valid_dataset, 
batch_size=BATCH_SIZE)
  model = Classifier(IN_DIM, HIDEEN_DIM, OUT_DIM, device=DEVICE)
  _, ta, _, va = model.training(train_dataloader, valid_dataloader, 
epochs=80, learning_rate=lr, learning_rate_decay=LR_DECAY, 
verbose=0)
  train_acc.append(ta); valid_acc.append(va)

plot_curves(lr_cand, train_acc, valid_acc)
```

執行結果：

我們可以發現在三個 learning rate 中，0.01 的準確率是最高的，因為過小的 learning rate 可能會導致更新速度太慢，使得結果不好，以此例來看，0.01 的 learning rate 效果最好

```
Training with batch_size=16, hidden_dim=64, learning_rate=0.01, learning_rate_decay=0.95
Training with batch_size=16, hidden_dim=64, learning_rate=0.001, learning_rate_decay=0.95
Training with batch_size=16, hidden_dim=64, learning_rate=0.0001, learning_rate_decay=0.95
```

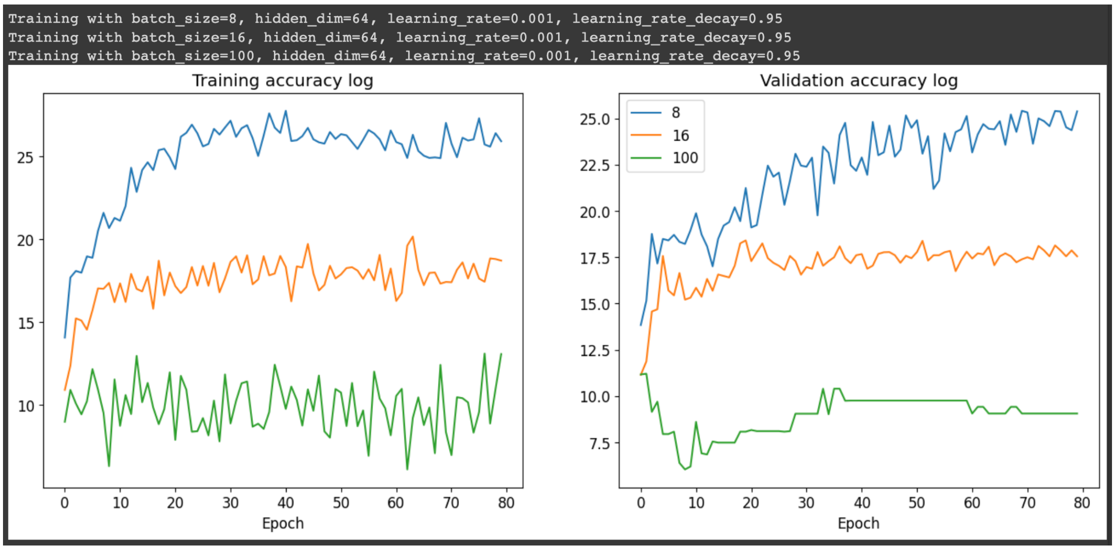


再來是 batch size 的測試

```
bs_cand = [8, 16, 100]
train_acc = []
valid_acc = []

for bs in bs_cand:
  train_dataloader = Dataloader(train_dataset, batch_size=bs)
  valid_dataloader = Dataloader(valid_dataset, batch_size=bs)
  model = Classifier(IN_DIM, HIDEEN_DIM, OUT_DIM, device=DEVICE)
  _, ta, _, va = model.training(train_dataloader, valid_dataloader,
epochs=80, learning_rate=LR, learning_rate_decay=LR_DECAY,
verbose=0)
  train_acc.append(ta); valid_acc.append(va)

plot_curves(bs_cand, train_acc, valid_acc)
```

執行結果：

我們可以發現 batch size 為 8 的準確率最高，而過大的 batch size 可能會導致更新參數的速度較慢，且會使得隨機性較低，訓練出來的模型準確率會有不夠高的問題。在此例中，batch size 為 8 的效果最好。

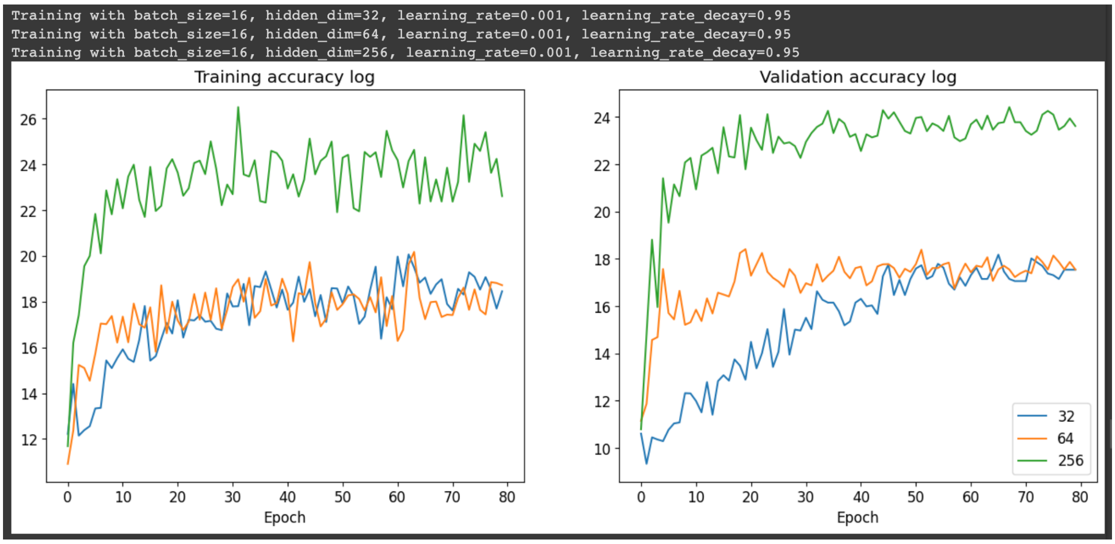


再來是 hidden layer dimension 的部分

```
hd_cand = [32, 64, 256]
train_acc = []
valid_acc = []

for hd in hd_cand:
  train_dataloader = Dataloader(train_dataset,
batch_size=BATCH_SIZE)
  valid_dataloader = Dataloader(valid_dataset,
batch_size=BATCH_SIZE)
  model = Classifier(IN_DIM, hd, OUT_DIM, device=DEVICE)
  _, ta, _, va = model.training(train_dataloader, valid_dataloader,
epochs=80, learning_rate=LR, learning_rate_decay=LR_DECAY,
verbose=0)
  train_acc.append(ta); valid_acc.append(va)

plot_curves(hd_cand, train_acc, valid_acc)
```

執行結果：

我們可以發現隱藏層的維度愈高，模型的準確率也會愈高，這是因為如果隱藏層複雜(也就是參數愈多)，則訓練出來的模型會更完善，準確率也較高。

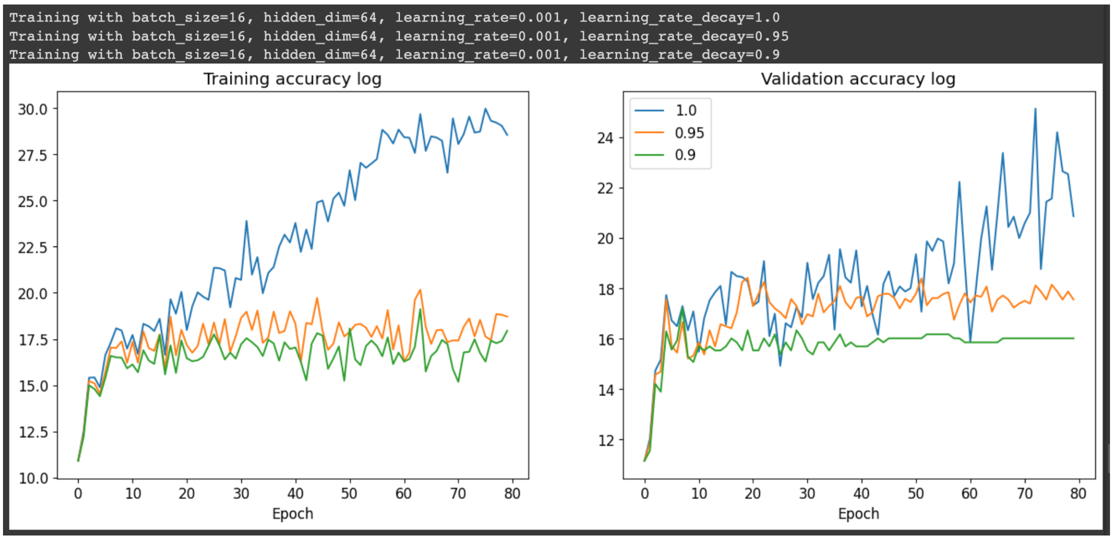


最後是 learning rate decay 的測試

```
lr_decay_cand = [1.0, 0.95, 0.9]
train_acc = []
valid_acc = []

for lrd in lr_decay_cand:
  train_dataloader = Dataloader(train_dataset,
batch_size=BATCH_SIZE)
  valid_dataloader = Dataloader(valid_dataset,
batch_size=BATCH_SIZE)
  model = Classifier(IN_DIM, HIDEEN_DIM, OUT_DIM, device=DEVICE)
  _, ta, _, va = model.training(train_dataloader, valid_dataloader,
epochs=80, learning_rate=LR, learning_rate_decay=lrd, verbose=0)
  train_acc.append(ta); valid_acc.append(va)

plot_curves(lr_decay_cand, train_acc, valid_acc)
```

執行結果：

以此例來看，learning rate decay 為 1 最好(也就是每回合不去更改 learning rate)，因為在先前我們測試 learning rate 是 0.01 的結果已經是裡面比較好的了，因此如果我們在中途去讓 learning rate 變低的話，可能會使的模型訓練的準確率沒有那麼高，因此不更動 learning rate 或許是一種好方法。



Training with Best Hyperparameters

就上述觀察，我們可以發現四個 hyperparameters 用以下的配置會有局部最好的結果

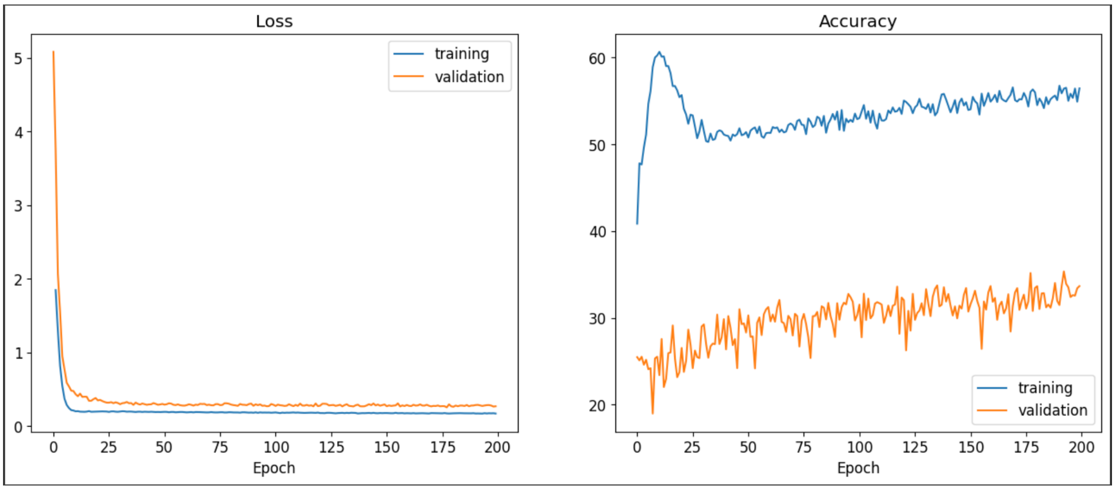
Batch size = 8
Learning rate = 0.01
Learning rate decay = 1.0
Hidden layer dimension = 256

因此我們使用以上配置來訓練看看

```
# Configurations
IN_DIM = np.prod(images[0].shape)
OUT_DIM = len(classes)
# Please set your hyperparameters here
BATCH_SIZE = 8
LR = 1e-2
LR_DECAY = 1.0
HIDEEN_DIM = 256
train_dataloader = Dataloader(train_dataset, batch_size=BATCH_SIZE)
valid_dataloader = Dataloader(valid_dataset, batch_size=BATCH_SIZE)
model = Classifier(IN_DIM, HIDEEN_DIM, OUT_DIM, device=DEVICE)
train_loss, train_acc, valid_loss, valid_acc =
model.training(train_dataloader, valid_dataloader, learning_rate=LR,
learning_rate_decay=LR_DECAY)
plot_result(train_loss, train_acc, valid_loss, valid_acc)
```

執行結果：(只擷取最後幾個 EPOCHS 的結果)

```
Epoch 195:
Training accuracy: 54.99%, loss: 0.1728
Validaion accuracy: 33.52%, loss: 0.2776
Epoch 196:
Training accuracy: 55.81%, loss: 0.1692
Validaion accuracy: 32.40%, loss: 0.2823
Epoch 197:
Training accuracy: 55.33%, loss: 0.1720
Validaion accuracy: 32.62%, loss: 0.2832
Epoch 198:
Training accuracy: 56.42%, loss: 0.1708
Validaion accuracy: 32.56%, loss: 0.2766
Epoch 199:
Training accuracy: 54.90%, loss: 0.1724
Validaion accuracy: 33.41%, loss: 0.2632
Epoch 200:
Training accuracy: 56.43%, loss: 0.1683
Validaion accuracy: 33.66%, loss: 0.2671
```



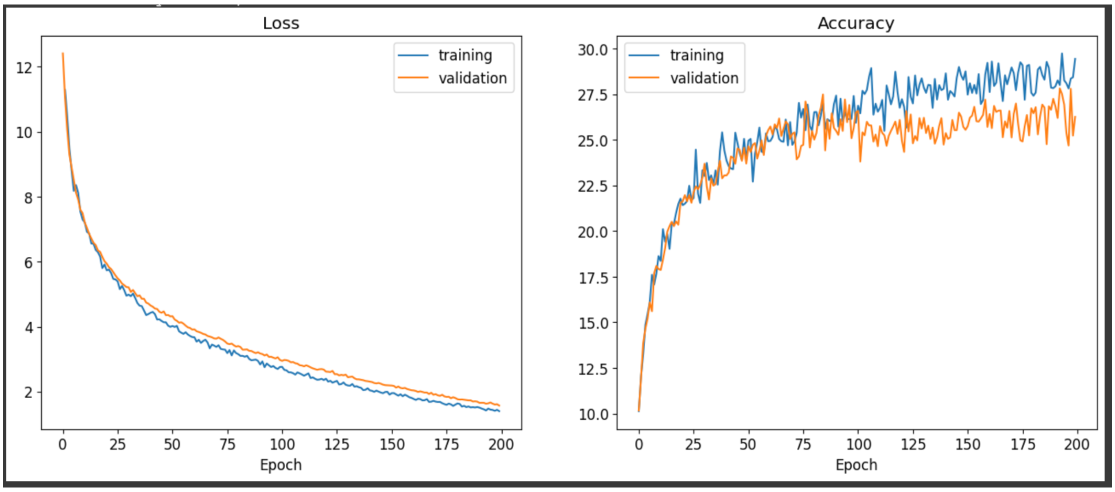
最後的準確率為 34.49 %

```
test_dataloader = Dataloader(test_dataset)
model.load_model(MODEL_PATH)
model.test(test_dataloader)
```

```
Got 3449 correct prediction in 10000 test data, accuracy: 34.49%
```

若將 learning rate 改成 0.0001，則準確率會大幅下降，變成 26.68 %

```
# Please set your hyperparameters here
BATCH_SIZE = 8
LR = 1e-4
LR_DECAY = 1.0
HIDEEN_DIM = 256
```



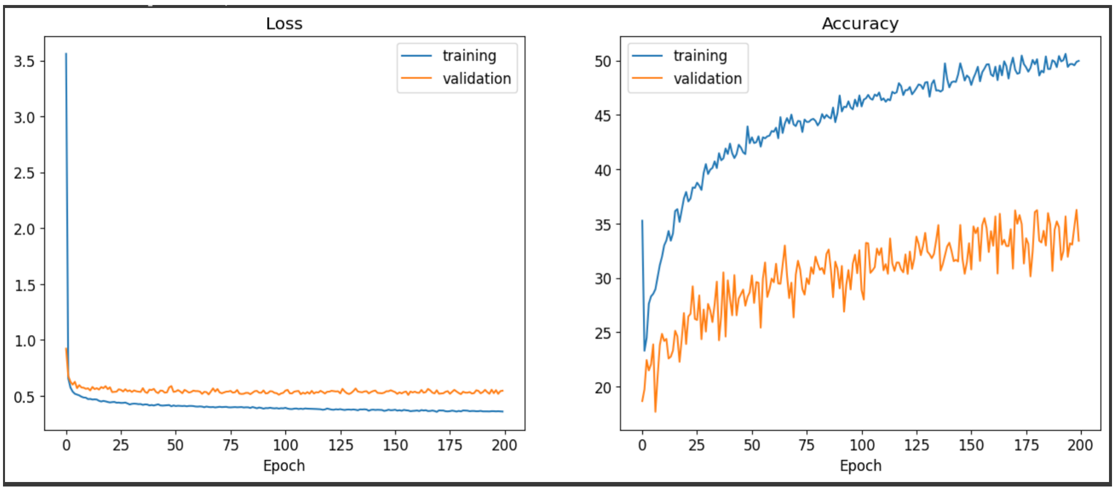

```
test_dataloader = Dataloader(test_dataset)
model.load_model(MODEL_PATH)
model.test(test_dataloader)
Got 2668 correct prediction in 10000 test data, accuracy: 26.68%
```

而若調大 batch size、調低 learning rate 和 learning rate decay 以及降低 hidden layer dimension，以上述結果來看，都會使得結果準確率降低，因此目前我們可以推論，最好的 hyperparameter 為：

Batch size = 8
Learning rate = 0.01
Learning rate decay = 1.0
Hidden layer dimension = 256

額外嘗試：將 batch size 改小變成 4，準確率會更高，為 35.42 %

(因為 batch size = 4 不在範例的候選名單內，因此程式碼的最終版本用 batch size = 8 的版本)



```
Got 3542 correct prediction in 10000 test data, accuracy: 35.42%
```