深度視覺 HW3

M113040064 李冠宏

首先是一些前置作業(載入資料夾、引入套件、測試 GPU 和載入 dataset)

```
from google.colab import drive
drive.mount('/content/drive')
import sys
sys.path.append('/content/drive/MyDrive/HW3/')
# 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, Model_Tests, Loss_Test, Optimizer_Test,
evaluate
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()
```

接下來是測試資料的個數和 shape

```
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}') #10000 筆 data
print(f'Number of classes: {len(classes)}')
print(f'Classes: {classes}')
```

執行結果：

```
Data type of images: uint8 (value range: [0, 255])
Shape of images: (10000, 32, 32, 3)
Shape of labels: (10000,)
Number of classes: 2
Classes: ['cat', 'dog']
```

測試 dataset (會印出圖片)

```
# Display samples of each class
plt.rcParams['figure.figsize'] = (3, 16)
num_classes = len(classes) #2
num_example = 10 #每一個類別的數量
for label, class_name in enumerate(classes):
  idxs = np.flatnonzero(labels == label) #flatnonzero 會回傳非 0 元素的index
  idxs = np.random.choice(idxs, num_example, replace=False) #代表從idxs 中隨機抽取 num_example 的"不重複"數字
  for i, idx in enumerate(idxs):
    plt_idx = i * num_classes + label + 1
    plt.subplot(num_example, num_classes, plt_idx) #plt.subplot(row, column, index)
    plt.imshow(images[idx]); plt.axis('off')
    if i == 0:
      plt.title(class_name) #印出類別的 index
plt.show()
```

接下來建立 dataset

首先我們會先在建構式的地方進行 attributes 的初始化，以及資料的壓縮把原本 pixel 的值從[0,255]壓縮到[0,1]，並將 pixel 拉平，從 32x32x3 變成扁平的 3072，建立回傳資料大小的__len__(也就是 label 的大小)和__getitem__(透過 index 取值)

```
class Dataset(object):
  def __init__(self, images, labels):
    self.images = None
    self.labels = None
    ####################
    # TODO:
```

```
    # 1. Flatten data, that is, change shape of images from (num_images,
image_height, image_width, image_channel) to (num_images,
flatten_data)
    # 2. Normalize data from range [0, 255] to range [0, 1], you should
cast its type to np.float64 first
    # 3. Store flatten and normalized data in variable `self.images`
    # 4. Store labels in variable `self.labels`
    ####################

    # -----START OF YOUR CODE-----
    images = np.reshape(images, (images.shape[0],
-1)).astype(np.float64) #flatten the numpy array(10000, 3072) and
cast its type to np.float64 first
    images = images.astype(np.float64) #將 datatype 轉換成 float64
    for i in range(len(images)):
      images[i] = np.divide(images[i], 255 * np.ones(32*32*3)) #正規
化
    self.images = images
    self.labels = labels
    # ------END OF YOUR CODE------

  def __len__(self):
    num_data = None
    ####################
    # TODO:
    # 1. Store the length of dataset, that is, the quantity of data
in variable `num_data`
    ####################

    # -----START OF YOUR CODE-----
    num_data = len(self.labels)
    # ------END OF YOUR CODE------

    return num_data

  def __getitem__(self, idx):
    image = None
    label = None
```

```
    ####################
    # TODO:
    # 1. Store idx-th image in variable `image` and its label in variable
`label`
    ####################

    # -----START OF YOUR CODE-----
    image = self.images[idx]
    label = self.labels[idx]
    # ------END OF YOUR CODE------

    return image, label
```

此處是將剛剛定義好的 Dataset 建立物件，並將 dataset 分為 3 種類型，分別是 train data、valid data 和 test data (比例是 60% : 20% : 20%)，分別用來訓練模型、測試模型準確率和當作最後以訓練好模型的輸入

```
# Data splitting with numpy array slicing
num_data = images.shape[0] #總共資料的筆數
split_ratio = {'train': 0.6, 'valid': 0.2, 'test': 0.2} #各種資料的比例
num_valid_data = round(num_data * split_ratio['valid']) #valid data 的筆數
num_test_data = round(num_data * split_ratio['test']) #test data 的筆數
num_train_data = num_data - (num_valid_data + num_test_data) #train data 的筆數

start_idx = 0
#train data 的 dataset
train_dataset = Dataset(images[start_idx:start_idx+num_train_data], labels[start_idx:start_idx+num_train_data])
start_idx += num_train_data

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

```
#test data 的 dataset
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)}')
```

這邊會確認資料壓縮及拉平是否有正確完成，以及將資料視覺化(印出 RGB 三色的圖片)

```
from mpl_toolkits.axes_grid1 import make_axes_locatable
plt.rcParams['figure.figsize'] = (10, 5)

#第一筆資料(分別是初始的資料和 flatten 資料)
original_data = images[0]
train_data_img, train_data_label = train_dataset[0]

#印出原本資料的資訊
print(f'Data type of original image: {original_data.dtype} (value
range: [{original_data.min()}, {original_data.max()}])')
print(f'Shape of original image: {original_data.shape}')
#印出處理過資料(flatten train data)的資訊
print(f'Data type of dataset image: {train_data_img.dtype} (value
range: [{train_data_img.min()}, {train_data_img.max()}])')
print(f'Shape of dataset image: {train_data_img.shape}\n')

#將 flatten data 變回初始資料的 shape 以進行視覺化
heigth, width, channel = original_data.shape
train_data_img = train_data_img.reshape(heigth, width, channel) #
Reshape flatten data for visualization

for channel_idx, color_channel in enumerate(['red', 'green',
'blue']):
  fig, (ax1, ax2) = plt.subplots(1, 2)

  #初始資料的圖(左邊)
  ax1.set_title(f'Original data\nin {color_channel} color channel')
```

```
  img1 = ax1.imshow(original_data[:, :, channel_idx],
cmap=f'{color_channel.capitalize()}s')
  #圖片右邊的直條圖
  divider = make_axes_locatable(ax1)
  cax = divider.append_axes('right', size='5%', pad=0.05)
  fig.colorbar(img1, cax=cax, orientation='vertical')

  #flatten data 的圖(右邊)
  ax2.set_title(f'Training dataset data\nin {color_channel} color
channel')
  img2 = ax2.imshow(train_data_img[:, :, channel_idx],
cmap=f'{color_channel.capitalize()}s')
  #圖片右邊的直條圖
  divider = make_axes_locatable(ax2)
  cax = divider.append_axes('right', size='5%', pad=0.05)
  fig.colorbar(img2, cax=cax, orientation='vertical')

  fig.show()
```

執行結果：

可以發現 pixel 的值都已經被壓縮到[0,1]之間的值，且每張圖片的資料 shape 為(3072,)，已經拉平

```
Data type of original image: uint8 (value range: [11, 218])
Shape of original image: (32, 32, 3)
Data type of dataset image: float64 (value range: [0.043137254901960784, 0.8549019607843137])
Shape of dataset image: (3072,)
```

接下來是 dataloader 的建立，他主要的目的是每次訓練時不要都跑完所有資料才更新一次權重，而是每訓練完一個 batch 就調整一次，是有效增進訓練時間的方式，而 batch 的數量就是全部資料的比數除以一個 batch 的大小

```
# data loader 可以一次載入一些些 dataset 的資料就好，而無須全部載入
class Dataloader(object):
  def __init__(self, dataset, batch_size=1):
    self.dataset = dataset
    self.indice = np.array(range(len(self.dataset)))
    self.batch_size = batch_size

  def __len__(self):
    num_batch = None
```

```
    ####################
    # TODO:
    # 1. Store the number of batches in variable `num_batch`
    ####################

    # -----START OF YOUR CODE-----
    num_batch = int(len(self.dataset)/self.batch_size)
    # ------END OF YOUR CODE------

    return num_batch

  def __getitem__(self, idx):
    batch_data = self.dataset[self.indice[idx: idx+self.batch_size]]
    return batch_data

  def shuffle(self):
    np.random.shuffle(self.indice)
```

進行 dataloader 的建立和測試

```
BATCH_SIZE = 16

train_dataloader = Dataloader(train_dataset, batch_size=BATCH_SIZE)
valid_dataloader = Dataloader(valid_dataset, batch_size=BATCH_SIZE)

print(f'Number of batches in training dataloader:
{len(train_dataloader)}')
print(f'Number of batches validation dataloader:
{len(valid_dataloader)}')
```

執行結果：

現在 batch size 是 16，training data 有 6000 筆，因此有 375 個 batches，而 validation data 有 2000 筆，因此有 125 個 batches

```
Number of batches in training dataloader: 375
Number of batches validation dataloader: 125
```

建立 model
Sigmoid function 的公式如下：

$$S(x) = \frac{1}{1+e^{-x}}$$

而在 forward 中，我們要先把每個 pixel 乘上一個權重，做完乘累加後送進 sigmoid 函式，最後得到一個 pred_y 向量，存每一筆資料所預測的值

backward 中則是用到 chain rule 來算 loss function 的梯度，並計算其平均，以利之後更新參數

在 BCE loss 中，我們定義了 L，也就是 loss function，並算出他的導數 dL，之後要傳到 backward 函式中做 back propagation(反向傳播)

最後在 optimizer 中，每經過一個 batch 我們就更新一次參數，而更新方式是每次將權重減去 learning rate * loss function 的梯度，也就是剛剛 backward 函式回傳的 dW

```
#建立 model
class LinearClassifier(object):
  def __init__(self, device='cpu'):
    self.W = None # classifier weights
    self.dv = device

  def save_weights(self, path):
    torch.save(self.W, path)

  def load_weights(self, path):
    self.W = torch.load(path)

  def sigmoid(self, score):
    prob = torch.zeros_like(score)
    ####################
    # TODO:
    # 1. Implement sigmoid function on `score` and store in variable
`prob`
    ####################
```

```
    # -----START OF YOUR CODE-----
    prob = 1/(1 + np.exp(-score))
    # ------END OF YOUR CODE------

    return prob

  def forward(self, x):
    num_data, data_dim = x.shape #(16, 3072)

    #權重為空 -> 要初始化
    if self.W is None:
      np.random.seed(0)
      self.W = torch.from_numpy(np.random.randn(data_dim) *
1e-4).to(self.dv) #一開始先給隨機的權重(未訓練過的)，其維度為
data_dim(3072)

    x = x.to(self.dv)
    pred_y = torch.zeros(num_data).to(self.dv) #產生一個維度和 num_data
一樣的全為 0 之 tensor
    ####################
    # TODO:
    # 1. Implement linear classifier f(x) = W * x, and then transform
the predicted values to probabilities by sigmoid function
    # 2. Store probabilities in variable `pred_y`
    ####################

    # -----START OF YOUR CODE-----
    # for i in range(num_data): # num_data = 16
    #   pred_y[i] = self.sigmoid(torch.sum(torch.mul(self.W, x[i])))
    temp = torch.matmul(x,self.W)
    pred_y = self.sigmoid(temp)
    # ------END OF YOUR CODE------

    self.cache = (x, pred_y)
    return pred_y

  def backward(self, dL):
```

```
    dL = dL.to(self.dv)
    dW = torch.zeros(1).to(self.dv)
    ####################
    # TODO:
    # 1. Derive the gradients of weights, calculate their average and store it in variable `dW`
    #   HINT 1: Chain rule d(loss)/d(weight) = d(loss)/d(pred_y) * d(pred_y)/d(weight)
    #   HINT 2: Use self.cache (batch flatten data x that multiplied weights in forward process, preditcted output that calculated by weights and input)
    ####################

    # -----START OF YOUR CODE-----
    temp = dL * self.cache[1] * (1-self.cache[1]) #利用 chain rule 算出一個值
    dW = torch.matmul(self.cache[0].T, temp) #矩陣相乘
    dW = torch.sum(dW) / self.cache[0].shape[0] #算平均
    # ------END OF YOUR CODE------

    return dW

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

  def __call__(self, y, pred_y): #y:正確解答 pred_y:猜的答案
    y = y.to(self.dv) #正確答案
    pred_y = pred_y.to(self.dv) #猜的答案
    L = torch.zeros_like(y).to(self.dv)
    dL = torch.zeros_like(y).to(self.dv)
    ####################
    # TODO:
    # 1. Implement binary cross-entropy loss and store the results of each data in variable `L`
    # 2. Derive the gradient of binary cross-entropy loss and store the results of each data in variable `dL`
    ####################
```

```
    # -----START OF YOUR CODE-----
    L = -(y * torch.log(pred_y) + (1-y) * torch.log(1-pred_y))
    dL = -((y/pred_y)-((1-y)/(1-pred_y))) # loss function 的導數
    # ------END OF YOUR CODE------

    return L, dL

class Optimizer(object):
  def __init__(self, model, learning_rate):
    self.model = model
    self.lr = learning_rate

  def step(self, dW):
    dW = dW.to(model.dv)
    new_weights = self.model.W.clone()
    ####################
    # TODO:
    # 1. Update model weights by gradient descent and store it in
variable `new_weights`
    ####################

    # -----START OF YOUR CODE-----
    new_weights -= self.lr * dW
    # ------END OF YOUR CODE------

    self.model.W = new_weights
```

這邊是進行 model 的測試

```
model = LinearClassifier(DEVICE)
Model_Tests(model)
```

```
Result of sigmoid function with single value 0: Correct
Result of sigmoid function with all-zero array: Correct
Result of sigmoid function with single value 100: Correct
Result of sigmoid function with all-100 array: Correct
Result of forward: Correct
Result of backward: Correct
```

```
loss_func = BCEloss(DEVICE)
Loss_Test(loss_func)
```

```
Result of forward: Correct
Result of backward: Correct
```

```
optimizer = Optimizer(model, learning_rate=0.1)
Optimizer_Test(optimizer)
```

```
Result of optimizer test: Correct
```

前置作業都做完，data set、data loader 和 model 都建立完後，我們就可以來做訓練了

```
# -----You may change values here-----
LR = 1e-5
EPOCHS = 30
# ---------------------------------------

model = LinearClassifier(DEVICE) #建立 model
loss_func = BCEloss(DEVICE) #loss function
optimizer = Optimizer(model, learning_rate=LR) #做 gradient descent

best_val_loss = None

for epoch in range(EPOCHS):
  print(f'Epoch {epoch+1}:')

  # training data
  total_loss = 0.0
  total_corr = 0.0; total_eval = 0.0
  for batch_idx in range(len(train_dataloader)):
    ### Model Input ###
    # 將資料分為好幾個 batch，批次執行 (一次執行一個 batch, size 為 16)
    batch_data = train_dataloader[batch_idx] #len(batch_data)=16
    batch_img, batch_label = batch_data #len(batch_img)=3072
    batch_img = torch.from_numpy(batch_img) #將資料從 numpy array 轉成
tensor
    pred_label = model.forward(batch_img) #預測的 label
```

```
    ### Loss Calculation ###
    batch_label = torch.from_numpy(batch_label.astype(np.float64))
    loss, grad_loss = loss_func(batch_label, pred_label) #算 loss (解答,猜的)

    ### Backpropagation ###
    grad_weight = model.backward(grad_loss)
    optimizer.step(grad_weight)

    ### Model Evaluation ###
    total_loss += np.sum(loss.cpu().numpy())
    total_eval += BATCH_SIZE
    total_corr += evaluate(batch_label.cpu().numpy(), pred_label.cpu().numpy())

  total_corr /= total_eval
  total_loss /= total_eval
  print(f'Training accuracy: {format(total_corr*100, ".2f")}%, loss: {format(total_loss, ".4f")}')
  train_dataloader.shuffle()

  # validation data
  total_loss = 0.0
  total_corr = 0.0; total_eval = 0.0
  for batch_idx in range(len(valid_dataloader)): # validation
    ### Model Input ###
    batch_data = valid_dataloader[batch_idx]
    batch_img, batch_label = batch_data
    batch_img = torch.from_numpy(batch_img)
    pred_label = model.forward(batch_img)

    ### Loss Calculation ###
    batch_label = torch.from_numpy(batch_label.astype(np.float64))
    loss, _ = loss_func(batch_label, pred_label)

    ### Model Evaluation ###
    total_loss += np.sum(loss.cpu().numpy())
```

```
    total_eval += BATCH_SIZE
    total_corr += evaluate(batch_label.cpu().numpy(),
pred_label.cpu().numpy())

  total_corr /= total_eval
  total_loss /= total_eval
  print(f'Validation accuracy: {format(total_corr*100, ".2f")}%,
loss: {format(total_loss, ".4f")}')

  if best_val_loss is None or total_loss < best_val_loss:
    best_val_loss = total_loss
    model.save_weights(MODEL_PATH)
    print('[WEIGHTS SAVED]')
```

而訓練中我們測試以下四種可能性

	EPOCHS = 30	EPOCHS = 50
learning rate = 1e-5	LR = 1e-5 EPOCHS = 30	LR = 1e-5 EPOCHS = 50
learning rate = 1e-6	LR = 1e-6 EPOCHS = 30	LR = 1e-6 EPOCHS = 50

training 的結果(最後一個 epoch 的結果)：

learning rate = 1e-5 EPOCHS = 30	Training accuracy: 48.70%, loss: 1.8531 Validation accuracy: 52.80%, loss: 1.1710
learning rate = 1e-6 EPOCHS = 30	Training accuracy: 50.12%, loss: 0.6621 Validation accuracy: 52.80%, loss: 0.7362
learning rate = 1e-5 EPOCHS = 50	Training accuracy: 53.12%, loss: 1.9188 Validation accuracy: 52.80%, loss: 0.6912
learning rate = 1e-6 EPOCHS = 50	Training accuracy: 50.00%, loss: 0.6608 Validation accuracy: 52.80%, loss: 0.7548

我們可以發現 learning 愈低的話，training loss 會愈小，這是因為太大的學習率可能會使得做 gradient descent 時跨太大步，而導致 loss 變高。

而因為 model 是線性分類的，所以在分類的結果並不是很好，準確率都沒超過 60%，之後若用神經網路來做非線性的分類能使的結果變更好。

而最後，EPOCHS 數愈多照理說準確率會愈高，因為訓練的次數多，但也有可能會有 overfitting 的問題

最後是將 test data 作為 input 送進 model 內做預測

```
# test data
model.load_weights(MODEL_PATH)
test_dataloader = Dataloader(test_dataset)
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
  batch_img = torch.from_numpy(batch_img).to(DEVICE)
  pred_label = model.forward(batch_img)
  ### Model Evaluation ###
  total_eval += 1
  total_corr += evaluate(batch_label, pred_label.cpu().numpy())
print(f'Got {total_corr} correct prediction in {total_eval} test data, accuracy: {format((total_corr*1.0/total_eval)*100, ".2f")}%')
```

test data 的結果：(準確率都一樣)

learning rate = 1e-5, EPOCHS = 30

```
Got 1005 correct prediction in 2000 test data, accuracy: 50.25%
```

learning rate = 1e-6, EPOCHS = 30

```
Got 1005 correct prediction in 2000 test data, accuracy: 50.25%
```

learning rate = 1e-5, EPOCHS = 50

```
Got 1005 correct prediction in 2000 test data, accuracy: 50.25%
```

learning rate = 1e-6, EPOCHS = 50

```
Got 1005 correct prediction in 2000 test data, accuracy: 50.25%
```