Project 2

廖泓博

1. 前言

我上學期有修過電腦視覺與深度學習，其中有作業是用pytorch實作一個模型，因此對電腦視覺相關深度學習 (尤其是 pytorch)已經有一定程度理解。

我常用電腦 (實驗室電腦) 無法使用cuda，而colab有時間限制不能讓我隨便測試，我自己的筆電雖然有cuda，但是平常待在實驗室不常有時間使用，因此整個作業要在實驗室電腦、colab、自己筆電三個不同環境上跑。

下面程式碼片段都是經過簡化的，直接拿去跑會有錯誤。

1. 初始版本

因為連中途會遇到什麼錯誤都不知道，第一個板本主要目標是能做出來就好。

1. 資料前處理

因為第一次用keras，連能不能成功跑起來都不知道，因此只做最小程度的前處理。

前處理包含進行one hot和改變資料集維度，因為我模型忘記做flatten，reshape就變成必要。

Y\_OneHot = utils.to\_categorical(Y, num\_classes=num\_classes);

Y\_Reshape = Y\_OneHot.reshape((-1, 1, 1, num\_classes));

1. 建立模型

使用最基本的Lenet-5

def Lenet(n\_classes: int) -> Model:

    model = keras.Sequential();

    model.add(BatchNormalization());

    model.add(Conv2D(6, (5, 5), activation=tanh));

    model.add(MaxPooling2D((2, 2)));

    model.add(Conv2D(16, (5, 5), activation=tanh));

    model.add(MaxPooling2D((2, 2)));

    model.add(Conv2D(120, (5, 5), activation=tanh));

    model.add(Dense(84, activation=tanh));

    model.add(Dense(n\_classes, activation=softmax));

    return model;

這個模型幾乎與原先路文上的一樣，只有改變輸出節點數、輸出層改為全連接(而非高斯連接)、在卷積層(而非池化層)使用非線性。

因為是分類任務，模型loss採用cross entropy。

模型optimizer採用sgd。根據上學期經驗，sgd很慢、準度很低，但是如果模型或其他地方搞砸 (例如沒初始化權重) 的話sgd至少還能跑出結果 (準度大於平均)，adam 剛跑就會直接跑出NaN。

1. 訓練模型

使用的超參數為: epoch=10, batch\_size=16。

和pytorch (或tensorflow) 很不一樣的是訓練過程不是迴圈，而是把超參數餵進model.fit之後直接跑完全程。

1. 測試結果

我知道測試集不該隨時拿出來看，但是因為訓練過程中驗證集的loss看起來還不錯 (大約0.17)，我還是拿出來看了一下。

結果:

Valid loss = 0.2038 (沒有accuracy，我不知道怎麼叫出來)。

Test loss: 0.2243 (同樣沒有accuracy)。

Inference: 那10張全部判斷成第三個，機率都是0.494。

雖然模型沒做flatten，我事後加上去之後結果仍一樣。

看到這裡我感覺前面這些訓練都是打假球，怎麼會訓練時loss都有降低，結果那10張不只全錯，得到的數值還一模一樣。我目前想到的有三種原因，搞砸某些步驟、前處理不夠多、模型過於簡單。

1. 修復和模組化

這個版本主要修復錯誤和改善可移植性 (畢竟我要在三個不同環境上跑)。

1. 環境設定

我知道這很枝微末節，浪費的時間大概也比省的時間多，但是為了未來的我方便，我還是做個簡單版本控制系統，在每次全部執行的時候一起執行，在最新版本的時候提早結束，並跳過已安裝完成的部分。

這裡不講細節，但是大概念是先看env.txt內的版本內容，如果不是最新就跑安裝套件，並且檢查必要資料夾或檔案是否都存在。如果是Colab的話要再掛載雲端硬碟和複製到本地。

(下面是檢查安裝套件的函式，對於不能輸入的vscode來說很有用)

def install\_package(package\_name: str) -> None:

    if not importlib.util.find\_spec(package\_name):

        command = ["pip", "install", package\_name];

        proc = subprocess.Popen(command, stdin=subprocess.PIPE, stdout=subprocess.PIPE, stderr=subprocess.STDOUT);

        \_ = proc.communicate(input='y'.encode())[0];

這個版本也把所有方法 (包括main() ) 獨立丟進一個函數，這樣不管轉成 .py 執行或是全部執行都不會有問題，改一個小地方也只需要跑2個block (改的那個和if name==main)。

這個版本也把整個訓練過程包成一個for loop，這樣要加新model或optimizer就很方便，也不用點多個block才能看各自model結果。

1. 資料前處理

雖然用imshow看不出差異，但體感圖片要轉成 0~1比較容易收斂，這邊順便把一些原本放在模型的函式移到預處理。

X\_resize = tf.image.resize(X, input\_size);

X\_resize = tf.image.rgb\_to\_grayscale(X\_resize);

X\_resize = X\_resize.numpy();

X\_resize\_scale = X\_resize / 256;

Y\_OneHot = to\_categorical(Y, num\_classes);

1. 建立模型

因為其他模型比較大，而且遇到一些技術問題，這邊還是用lenet。

大模型的問題是圖片大小和數量太多，無法一次前處理完 (可能要批次)，因此暫時沒碰這塊。

def Lenet(n\_classes: int) -> Model:

    model = keras.Sequential();

    model.add(keras.Input((32, 32, 1)));

model.add(Conv2D(6, (5, 5), activation=sigmoid));

    model.add(AveragePooling2D((2, 2)));

    model.add(Conv2D(16, (5, 5), activation=sigmoid));

    model.add(AveragePooling2D((2, 2)));

    model.add(Flatten());

    model.add(Dense(120, activation=sigmoid));

  model.add(Dense(84, activation=sigmoid));

    model.add(Dense(n\_classes, activation=softmax));

    return model;

上一個版本有一些錯誤 (應該是三層全連接+兩層卷積而不是反過來)。

上一個版本之所以沒有準確率的原因是compile階段少了metric。

這次使遇到的metric是 CategorialAccuracy()，一開始使用 Accuracy()，結果準確率都是0，我到現在還是不知道 Accuracy() 是用來做什麼的。

1. 訓練模型

使用的超參數為: epoch=10, batch\_size=16。

1. 測試結果

上一個版本問題是那10張輸入資料都是全黑 (從load and output images複製貼上範例程式小改一下沒追蹤變數的後果)。因為同組圖片還要跑前處理，這個版本直接大改這段程式碼，基本上就是將原本內部生成的變數移至外部。

def inference(x\_real, y\_real, sign\_names, model):

    prediction = predictModel(model, x\_real);

    prediction\_class = np.argmax(prediction, axis=1);

    prediction\_value = np.max(prediction, axis=1);

    for i in range(gl\_num\_images):

        printfmt = "predicted class = {} ({}), value={:.4f}\nactual class = {} ({}).";

        printstring = printfmt.format(prediction\_class[i],

                                      sign\_names[prediction\_class[i]],

                                      prediction\_value[i],

                                      y\_real[i],

                                      sign\_names[y\_real[i]]);

        print(printstring);

    plt.figure();

    for i in range(gl\_num\_images):

        plt.subplot(2, 5, i+1);

        plt.imshow(x\_real[i]);

    plt.show();

結果:

Valid loss = 0.3507，accuracy = 0.8820。

Inference:

第三張和第六張雖然判斷正確，但是信心值只有59和41%。

第九張把road work 誤判成beware of ice，而且信心值還85%。

1. 改良和擴充
   1. 資料前處理

如果要在每個epoch都隨機，就必須將前處理嵌入資料集中。根據model.fit，能做到的分別是pytorch和tensorflow的dataset。因為預設keras後端是tensorflow，我就直接用tf.data.Dataset。我參考 <https://www.tensorflow.org/tutorials/images/data_augmentation> 方式。因為使用tensorflow，所以容易遇到一堆奇怪問題，像是

ValueError: Dimensions must be equal, but are 4 and 3 for '{{node random\_crop/GreaterEqual}} = GreaterEqual[T=DT\_INT32](random\_crop/Shape, random\_crop/size)' with input shapes: [4], [3].

和

Invalid input shape for input Tensor("data:0", shape=(None, 1, 32, 32, 3), dtype=float32). Expected shape (None, 32, 32, 3), but input has incompatible shape (None, 1, 32, 32, 3)

這些錯誤訊息除了知道有錯誤外幾乎無意義，錯誤的地方也不是當初創的，所以追stack trace也沒用。

我把隨機前處理都拔掉，只留 tf.data.Dataset部分。結果第一個epoch跑很慢，後面幾個跑超快，準確率低很多，loss變3.4767，accuracy變0.0578。後來分析Dataset，看起來batch size是max。最後才發現是用tf.data.Dataset輸入錯誤 (應該是from\_tensor\_sices而不是 from\_tensors)。

我發現那個教程有很多錯誤或不適用的地方，例如counter dataset的使用，上面用法完全錯誤，就算我嘗試修正，最後還是碰到無解的型態(參數而非常數)。我原本有嘗試用全域變數模擬隨機種子，不過每個函數需要的維度和數量又不同(有0維和2維的)，而且如果我要用全域變數去當種子，那我不如直接不指定 (stateless\_random\_crop變random\_crop之類，也算是全域變數)。這邊我移除 stateless之後就沒問題了。

def preprocess(image, label, input\_size: tuple[int, int, int]):

    input\_x, input\_y, input\_z = input\_size;

    image = tf.cast(image, tf.float32)

    image = tf.image.resize(image, (input\_x, input\_y));

# image = tf.image.rgb\_to\_grayscale(image);

    image = (image / 255.0);

    return image, label;

def augment(image, label, input\_size: tuple[int, int, int]):

    image, label = preprocess(image, label, input\_size);

    pad\_amt = 10;

    image = tf.image.resize\_with\_pad(image, input\_size[0] + pad\_amt, input\_size[1] + pad\_amt);

    image = tf.image.random\_crop(image, size=input\_size);

    image = tf.image.random\_brightness(image, max\_delta=0.1);

    image = tf.clip\_by\_value(image, 0, 1);

    return image, label;

轉換方法弄好之後接下來是套用到真正的資料集。這邊有幾個重點:

* 一定要進行打亂，keras每個epoch預設都會打亂，但是在看到tf.Dataset的時候keras永遠不會打亂，手動設True也一樣。打亂數字越接近資料集大小越好，而且reshuffle\_each\_iteration 一定要手動設成True，這幾個少一個都會讓準確度大幅降低，(像是驗證集準確度0.05，且準確度在每次epoch開頭都重設，而且驗證集準確度明顯低於測試集，顯示overfitting情況嚴重)。
* 一定要更改batch大小，keras看到tf.Dataset的時候會直接無視該參數。
* 最好進行prefetch，不過這就只是速度問題，雖然可能差很多。

def datasetTransform(dataset, map\_fn, batch\_size, shuffle):

    if shuffle:

        dataset = dataset.shuffle(50000, reshuffle\_each\_iteration=True);

    dataset = dataset.map(map\_fn, num\_parallel\_calls=experimental.AUTOTUNE);

    dataset = dataset.batch(batch\_size);

    return dataset.prefetch(experimental.AUTOTUNE);

* 1. 建立模型

因為Dataset的成功，讓我能用原圖儲存在記憶體，要用的時候再展開，原本記憶體不夠的問題被緩解，我也因此能用更大的模型。

原本我打算用VGG19，但是考量記憶體可能不夠，我就用自創的假的VGG，概念類似，只是輸入變小 (64\*64\*3)，層數也少很多。

def VGGFake(n\_classes: int) -> Model:

    model = keras.Sequential();

    model.add(keras.Input((64, 64, 3)));

    model.add(Conv2D(32, (3, 3), activation=relu));

    model.add(MaxPooling2D((2, 2)));

    model.add(Conv2D(64, (3, 3), activation=relu));

    model.add(Conv2D(64, (3, 3), activation=relu));

    model.add(MaxPooling2D((2, 2)));

    model.add(Conv2D(128, (3, 3), activation=relu));

    model.add(Conv2D(128, (3, 3), activation=relu));

    model.add(MaxPooling2D((2, 2)));

    model.add(Flatten());

    model.add(Dense(1024, activation=relu));

    model.add(Dense(1024, activation=relu));

    model.add(Dense(n\_classes, activation=softmax));

    return model;

這裡還測試dropout的有效性，一般dropout比例應該會更高就是了。

def VGGFakePlus(n\_classes: int) -> Model:

    model = keras.Sequential();

    model.add(keras.Input((64, 64, 3)));

    model.add(layers.Conv2D(32, (3, 3), activation=relu));

    model.add(layers.MaxPooling2D((2, 2)));

    model.add(layers.Conv2D(64, (3, 3), activation=relu));

    model.add(layers.Conv2D(64, (3, 3), activation=relu));

    model.add(layers.MaxPooling2D((2, 2)));

    model.add(layers.Conv2D(128, (3, 3), activation=relu));

    model.add(layers.Conv2D(128, (3, 3), activation=relu));

    model.add(layers.MaxPooling2D((2, 2)));

    model.add(layers.Flatten());

  model.add(layers.Dense(1024, activation=relu));

    model.add(layers.Dropout(0.2));

    model.add(layers.Dense(1024, activation=relu));

    model.add(layers.Dropout(0.2));

    model.add(layers.Dense(n\_classes, activation=softmax));

    return model;

在這之上，我再加入Batch normalizaton。Batch normalizaton 因為要加在relu前，relu不能直接寫同一行，所以看起來行數特別多。

def VGGFakeBN(n\_classes: int) -> Model:

    model = keras.Sequential();

    model.add(keras.Input((64, 64, 3)));

    model.add(layers.Conv2D(32, (3, 3)));

    model.add(layers.BatchNormalization());

    model.add(layers.ReLU());

    model.add(layers.MaxPooling2D((2, 2)));

    model.add(layers.Conv2D(64, (3, 3)));

    model.add(layers.BatchNormalization());

    model.add(layers.ReLU());

    model.add(layers.Conv2D(64, (3, 3)));

    model.add(layers.BatchNormalization());

    model.add(layers.ReLU());

    model.add(layers.MaxPooling2D((2, 2)));

    model.add(layers.Conv2D(128, (3, 3)));

    model.add(layers.BatchNormalization());

    model.add(layers.ReLU());

    model.add(layers.Conv2D(128, (3, 3)));

    model.add(layers.BatchNormalization());

    model.add(layers.ReLU());

    model.add(layers.MaxPooling2D((2, 2)));

    model.add(layers.Flatten());

    model.add(layers.Dense(1024));

    model.add(layers.BatchNormalization());

    model.add(layers.ReLU());

    model.add(layers.Dropout(0.2));

    model.add(layers.Dense(1024));

    model.add(layers.BatchNormalization());

    model.add(layers.ReLU());

    model.add(layers.Dropout(0.2));

    model.add(layers.Dense(n\_classes, activation=softmax));

    return model;

* 1. 測試結果

Loss和accuracy為validation的，Correct/incorrect為inference的

10 epoch:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| network | Lenet | VGGFake | VGGFakeDrop | VGGFakeBN |
| loss | 0.5404 | 0.1660 | 0.1397 | 0.0725 |
| accuracy | 83.61% | 96.60% | 97.26% | 97.89% |
| Correct avg | 99.5% | 100% | 100% | 100% |
| Correct low | 2, 71.9% | - | 9, 95.7 | - |
| Incorect 1 | 3, 29.9% | 9, 89.5% | - | 9, 99.2% |
| Incorect 2 | 9, 52.8% | - | - | - |

30 epoch:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| network | Lenet | VGGFake | VGGFakeDrop | VGGFakeBN |
| loss | 0.4532 | 0.2808 | 0.0867 | 0.0817 |
| accuracy | 87.41% | 97.41% | 97.64% | 98.59% |
| Correct avg | 99.0% | 100% | 100% | 100% |
| Correct low | 3, 69.33% | 9, 82.79% | - | 80.23% |
| Incorect 1 | 9, 38.59% | - | 9, 0.5467 | 9, 71.56% |
| Incorect 2 | - | - | - | - |
| Test  loss | 0.5118 | 0.3712 | 0.2510 | 0.0964 |
| Test  accuracy | 88.04% | 95.72% | 95.95% | 98.15% |

這幾個model大概在20 epochs左右就收斂了，在這之後的validation數值幾乎就是隨機跳動。