Machine Learning HW3 Report

學號：B06901045 系級：電機三 姓名：曹林熹

1. **請說明你實作的 CNN 模型(best model)，其模型架構、訓練參數量和準確率為何？(1%)**

ANS:

a. 下圖是我的 CNN 模型，主要改的地方有第一層使用 padding = 2，然後用大一點的 mask_size = 5，並且在第一層不使用 pooling 以免之後不夠多層數。而我多加的 convolution_layer 是使用兩個 512 個 filters 的 layer，並且使用兩次 dropout_rate =0.5 的參數。詳細的可以在截圖中觀察。

```
self.cnn = nn.Sequential(
# class torch.nn.Sequential(*args)
# 多个模块按照它们传入构造函数的顺序被加入到网络中去
# 原來有五層
    # nn.Conv2d(3, 64, 3, 1, 1),  # [64, 128, 128] ps.padding 後從 128 => 130，再從 130-3+1 = 128
    nn.Conv2d(3, 64, 5, 1, 2),  # [64, 128, 128] ps.padding 後從 128 => 132，再從 132-5+1 = 128
    nn.BatchNorm2d(64),
    # 2D Normalization
    # class torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True)，
    # 其中num_features 为输入数据的通道数，BatchNorm2d计算的是每个通道上的归一化特征
    nn.ReLU(),
    # nn.MaxPool2d(2, 2, 0),        # [64, 64, 64] 128/2 = 64

    nn.Conv2d(64, 128, 3, 1, 1),  # [128, 128, 128] ps.padding 後從 128 => 130，再從 130-3+1 = 128
    nn.BatchNorm2d(128),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0),        # [128, 64, 64] 128/2 = 64

    nn.Conv2d(128, 256, 3, 1, 1), # [256, 32, 32] ps.padding 後從 64 => 66，再從 66-3+1 = 64
    nn.BatchNorm2d(256),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0),        # [256, 32, 32] 64/2 = 32

    nn.Conv2d(256, 512, 3, 1, 1), # [512, 32, 32] ps.padding 後從 32 => 34，再從 34-3+1 = 32
    nn.BatchNorm2d(512),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0),        # [512, 32, 32] 32/2 = 16

    nn.Conv2d(512, 512, 3, 1, 1), # [512, 16, 16] ps.padding 後從 16 => 18，再從 18-3+1 = 16
    nn.BatchNorm2d(512),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0),        # [512, 4, 4] 16/2= 8

    ############################多加層數##################################

    nn.Conv2d(512, 512, 3, 1, 1), # [512, 4, 4] ps.padding 後從 8 => 10，再從 10-3+1 = 8
    nn.BatchNorm2d(512),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0),       # [512, 2, 2] 8/2= 4

    nn.Conv2d(512, 512, 3, 1, 1), # [512, 4, 4] ps.padding 後從 4 => 6，再從 6-3+1 = 4
    nn.BatchNorm2d(512),
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0)        # [512, 2, 2] 4/2= 2
)
self.fc = nn.Sequential(
    # flatten
    nn.Linear(512*2*2, 1024),
    torch.nn.Dropout(0.5),
    nn.ReLU(),
    nn.Linear(1024, 512),
    torch.nn.Dropout(0.5),
    nn.ReLU(),
    nn.Linear(512, 11)
)
```

b. 參數採用 batch_size = 128，learning_rate = 0.001，epoch = 150

c. 下圖可看到我的 val_accuracy，有不錯的表現

```
[141/150] 42.12 sec(s) Train Acc: 0.977803 Loss: 0.000632 | Val Acc: 0.768805 loss: 0.008778
[142/150] 42.14 sec(s) Train Acc: 0.989864 Loss: 0.000301 | Val Acc: 0.761516 loss: 0.009381
[143/150] 42.02 sec(s) Train Acc: 0.992499 Loss: 0.000188 | Val Acc: 0.766472 loss: 0.009771
[144/150] 42.01 sec(s) Train Acc: 0.992905 Loss: 0.000170 | Val Acc: 0.784548 loss: 0.009401
[145/150] 41.99 sec(s) Train Acc: 0.990776 Loss: 0.000253 | Val Acc: 0.760350 loss: 0.009966
[146/150] 41.88 sec(s) Train Acc: 0.992094 Loss: 0.000232 | Val Acc: 0.741691 loss: 0.013715
[147/150] 41.87 sec(s) Train Acc: 0.971316 Loss: 0.000756 | Val Acc: 0.746064 loss: 0.009834
[148/150] 41.88 sec(s) Train Acc: 0.987229 Loss: 0.000355 | Val Acc: 0.755394 loss: 0.010138
[149/150] 41.99 sec(s) Train Acc: 0.989357 Loss: 0.000289 | Val Acc: 0.746939 loss: 0.010689
[150/150] 41.97 sec(s) Train Acc: 0.990371 Loss: 0.000266 | Val Acc: 0.765306 loss: 0.010459
```

下圖可以看到我的 test_accuracy，準確率有過 strong_baseline

predict_8.csv 0.82008
2 days ago by b06901045_DPGOD
8. best model 重 train, epoch = 120

2. 請實作與第一題接近的參數量，但 CNN 深度（CNN 層數）減半的模型，並說明其模型架構、訓練參數量和準確率為何？(1%)

ANS:

左圖是我的 best_model 參數，可以看到總參數為 11,267,083 個。右邊是我減半的參數，從七層 convolution 變成四層，總參數為 9,979,147

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1        [-1, 64, 128, 128]           4,864
       BatchNorm2d-2        [-1, 64, 128, 128]             128
              ReLU-3        [-1, 64, 128, 128]               0
            Conv2d-4       [-1, 128, 128, 128]          73,856
       BatchNorm2d-5       [-1, 128, 128, 128]             256
              ReLU-6       [-1, 128, 128, 128]               0
         MaxPool2d-7         [-1, 128, 64, 64]               0
            Conv2d-8         [-1, 256, 64, 64]         295,168
       BatchNorm2d-9         [-1, 256, 64, 64]             512
             ReLU-10         [-1, 256, 64, 64]               0
        MaxPool2d-11         [-1, 256, 32, 32]               0
           Conv2d-12         [-1, 512, 32, 32]       1,180,160
      BatchNorm2d-13         [-1, 512, 32, 32]           1,024
             ReLU-14         [-1, 512, 32, 32]               0
        MaxPool2d-15         [-1, 512, 16, 16]               0
           Conv2d-16         [-1, 512, 16, 16]       2,359,808
      BatchNorm2d-17         [-1, 512, 16, 16]           1,024
             ReLU-18         [-1, 512, 16, 16]               0
        MaxPool2d-19           [-1, 512, 8, 8]               0
           Conv2d-20           [-1, 512, 8, 8]       2,359,808
      BatchNorm2d-21           [-1, 512, 8, 8]           1,024
             ReLU-22           [-1, 512, 8, 8]               0
        MaxPool2d-23           [-1, 512, 4, 4]               0
           Conv2d-24           [-1, 512, 4, 4]       2,359,808
      BatchNorm2d-25           [-1, 512, 4, 4]           1,024
             ReLU-26           [-1, 512, 4, 4]               0
        MaxPool2d-27           [-1, 512, 2, 2]               0
           Linear-28                [-1, 1024]       2,098,176
          Dropout-29                [-1, 1024]               0
             ReLU-30                [-1, 1024]               0
           Linear-31                 [-1, 512]         524,800
          Dropout-32                 [-1, 512]               0
             ReLU-33                 [-1, 512]               0
           Linear-34                  [-1, 11]           5,643
================================================================
Total params: 11,267,083
Trainable params: 11,267,083
Non-trainable params: 0
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1        [-1, 64, 128, 128]           4,864
       BatchNorm2d-2        [-1, 64, 128, 128]             128
              ReLU-3        [-1, 64, 128, 128]               0
         MaxPool2d-4          [-1, 64, 64, 64]               0
            Conv2d-5         [-1, 128, 64, 64]          73,856
       BatchNorm2d-6         [-1, 128, 64, 64]             256
              ReLU-7         [-1, 128, 64, 64]               0
         MaxPool2d-8         [-1, 128, 32, 32]               0
            Conv2d-9         [-1, 256, 32, 32]         295,168
      BatchNorm2d-10         [-1, 256, 32, 32]             512
             ReLU-11         [-1, 256, 32, 32]               0
        MaxPool2d-12         [-1, 256, 16, 16]               0
           Conv2d-13         [-1, 512, 16, 16]       1,180,160
      BatchNorm2d-14         [-1, 512, 16, 16]           1,024
             ReLU-15         [-1, 512, 16, 16]               0
        MaxPool2d-16           [-1, 512, 8, 8]               0
           Linear-17                 [-1, 256]       8,388,864
          Dropout-18                 [-1, 256]               0
             ReLU-19                 [-1, 256]               0
           Linear-20                 [-1, 128]          32,896
          Dropout-21                 [-1, 128]               0
             ReLU-22                 [-1, 128]               0
           Linear-23                  [-1, 11]           1,419
================================================================
Total params: 9,979,147
Trainable params: 9,979,147
Non-trainable params: 0
```

下圖放上我的 model，主要改的地方是在第一層也使用了 max_pooling，而最後的三層我直接去除掉，因此從七層變成四層，最後的 size 是 8*8。而 fully connected layer 也更動了參數，改成數字較小的參數，dropout 也少掉了一次減少失去的參數量，使總參數與 best_model 相同。

```
super(Classifier, self).__init__()
# torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride, padding)
# torch.nn.MaxPool2d(kernel_size, stride, padding)
# input 維度 [3, 128, 128]
self.cnn = nn.Sequential(
# class torch.nn.Sequential(*args)
# 多个模块按照它们传入构造函数的顺序被加入到网络中去
# 原來有五層
    # nn.Conv2d(3, 64, 3, 1, 1),  # [64, 128, 128] ps.padding 後從 128 => 130，再從 130-3+1 = 128
    nn.Conv2d(3, 64, 5, 1, 2),  # [64, 128, 128] ps.padding 後從 128 => 132，再從 132-5+1 = 128
    # (5*5*3+1)*64 = 4,864
    nn.BatchNorm2d(64),
    # 128
    # 2D Normalization
    # class torch.nn.BatchNorm2d(num_features, eps=1e-05, momentum=0.1, affine=True)，
    # 其中num_features 为输入数据的通道数，BatchNorm2d计算的是每个通道上的归一化特征
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0),       # [64, 64, 64] 128/2 = 64

    nn.Conv2d(64, 128, 3, 1, 1),  # [128, 128, 128] ps.padding 後從 64 => 66，再從 66-3+1 = 64
    # (3*3*64+1)*128 = 73856
    nn.BatchNorm2d(128),
    # 256
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0),       # [128, 64, 64] 128/2 = 64

    nn.Conv2d(128, 256, 3, 1, 1), # [256, 32, 32] ps.padding 後從 32 => 34，再從 34-3+1 = 32
    # (3*3*128+1)*256 = 295,168
    nn.BatchNorm2d(256),
    # 512
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0),       # [256, 32, 32] 64/2 = 32

    nn.Conv2d(256, 512, 3, 1, 1), # [512, 32, 32] ps.padding 後從 16 => 18，再從 18-3+1 = 16
    # (3*3*256+1)*512 = 1,180,160
    nn.BatchNorm2d(512),
    # 1,024
    nn.ReLU(),
    nn.MaxPool2d(2, 2, 0),       # [512, 32, 32] 16/2 = 8


)
self.fc = nn.Sequential(
    # flatten
    nn.Linear(512*8*8, 256),
    # (512*8*8+1)*256 = 8,388,864
    # torch.nn.Dropout(0.5),
    nn.ReLU(),
    nn.Linear(256, 128),
    # (256+1)*128 = 32896
    torch.nn.Dropout(0.5),
    nn.ReLU(),
    nn.Linear(128, 11)
    # (128+1)*11 = 1419
)
```

下圖是我用此 model train 出來的 train_accuracy 與 val_accuracy，可以看的出來比之前還要差。

```
[141/150] 17.03 sec(s) Train Acc: 0.965741 Loss: 0.000994 | Val Acc: 0.615160 loss: 0.024130
[142/150] 17.04 sec(s) Train Acc: 0.974154 Loss: 0.000705 | Val Acc: 0.685423 loss: 0.017217
[143/150] 17.13 sec(s) Train Acc: 0.980134 Loss: 0.000530 | Val Acc: 0.669679 loss: 0.017779
[144/150] 17.05 sec(s) Train Acc: 0.982060 Loss: 0.000476 | Val Acc: 0.670262 loss: 0.019727
[145/150] 17.06 sec(s) Train Acc: 0.984695 Loss: 0.000373 | Val Acc: 0.655685 loss: 0.021622
[146/150] 17.06 sec(s) Train Acc: 0.988648 Loss: 0.000313 | Val Acc: 0.660641 loss: 0.020383
[147/150] 17.12 sec(s) Train Acc: 0.981553 Loss: 0.000504 | Val Acc: 0.672303 loss: 0.020321
[148/150] 17.05 sec(s) Train Acc: 0.984492 Loss: 0.000410 | Val Acc: 0.660933 loss: 0.021533
[149/150] 17.04 sec(s) Train Acc: 0.980134 Loss: 0.000538 | Val Acc: 0.664431 loss: 0.019381
[150/150] 17.08 sec(s) Train Acc: 0.979424 Loss: 0.000589 | Val Acc: 0.650729 loss: 0.019027
```

3. 請實作與第一題接近的參數量，簡單的 DNN 模型，同時也說明其模型架構、訓練參數和準確率為何？(1%)

ANS:

下圖是我的 DNN 模型，我只使用了一層的 hidden layer，詳細的架構如下左

圖。訓練參數使用 epoch = 150，總 model 參數如下右圖，有 12,600,971 個參數量，與 best_model 差不多。

```python
# DNN
class Classifier(nn.Module):
    def __init__(self):
        super(Classifier, self).__init__()
        self.fc = nn.Sequential(
            # flatten
            nn.Linear(3*128*128, 256),
            torch.nn.Dropout(0.5),
            nn.BatchNorm1d(256),
            nn.ReLU(),
            nn.Linear(256, 64),
            torch.nn.Dropout(0.5),
            nn.BatchNorm1d(64),
            nn.ReLU(),
            nn.Linear(64, 11),
        )

    def forward(self, x):
        out = x
        out = out.view(out.size()[0], -1)
        return self.fc(out)
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1                  [-1, 256]      12,583,168
           Dropout-2                  [-1, 256]               0
       BatchNorm1d-3                  [-1, 256]             512
              ReLU-4                  [-1, 256]               0
            Linear-5                   [-1, 64]          16,448
           Dropout-6                   [-1, 64]               0
       BatchNorm1d-7                   [-1, 64]             128
              ReLU-8                   [-1, 64]               0
            Linear-9                   [-1, 11]             715
================================================================
Total params: 12,600,971
Trainable params: 12,600,971
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 0.01
Params size (MB): 48.07
Estimated Total Size (MB): 48.27
```

可以看到我們的準確率非常地差，Val Acc 落在 25% 左右，甚至連 Train Acc 都只有 30% 正確率， 在這個情況下是個不好的模型。

```
[141/150] 5.91 sec(s) Train Acc: 0.300426 Loss: 0.015558 | Val Acc: 0.243732 loss: 0.016376
[142/150] 5.93 sec(s) Train Acc: 0.300527 Loss: 0.015587 | Val Acc: 0.246356 loss: 0.016805
[143/150] 5.94 sec(s) Train Acc: 0.299108 Loss: 0.015574 | Val Acc: 0.237026 loss: 0.016647
[144/150] 6.03 sec(s) Train Acc: 0.297993 Loss: 0.015582 | Val Acc: 0.233819 loss: 0.016572
[145/150] 6.05 sec(s) Train Acc: 0.297588 Loss: 0.015590 | Val Acc: 0.264431 loss: 0.016470
[146/150] 5.94 sec(s) Train Acc: 0.301642 Loss: 0.015582 | Val Acc: 0.233236 loss: 0.016621
[147/150] 5.95 sec(s) Train Acc: 0.301237 Loss: 0.015486 | Val Acc: 0.271137 loss: 0.016290
[148/150] 5.90 sec(s) Train Acc: 0.307521 Loss: 0.015428 | Val Acc: 0.250437 loss: 0.016610
[149/150] 5.91 sec(s) Train Acc: 0.302047 Loss: 0.015419 | Val Acc: 0.248105 loss: 0.016625
[150/150] 5.92 sec(s) Train Acc: 0.302757 Loss: 0.015421 | Val Acc: 0.241108 loss: 0.016592
```

4. 請說明由 1 ~ 3 題的實驗中你觀察到了什麼？(1%)

根據我的觀察，我發現在相同參數量下，layer 越多，得到的準確率就越高。每次 train 都採用 epoch = 150，除了第三題外，Train Acc 都逼近 100%，因此 Val Acc 會接近我們的 Test Acc。除非要重新大幅更改第二題的架構，不然單純改變參數很難贏過第一題的正確率。然而使用了第三題的 DNN 模型，可以看到即使將 epoch 調很多次也很難將 Train Acc 繼續往上升，我認為原因在於資料一進到訓練模型後就將圖片 flatten 了，變成一維，因此很難判斷資料庫圖片。

5. 請嘗試 data normalization 及 data augmentation，說明實作方法並且說明實行前後對準確率有什麼樣的影響？(1%)

ANS:

a. data augmentation

下圖是我先使用 data augmentation 的方法，並沒有使用 normalization。我將 Transform = None 改成 Transform = True，使其進行 data augmentation，採用的有以下兩種加強方式：

transforms.RandomHorizontalFlip() # 隨機將圖片水平翻轉
transforms.RandomRotation(15) # 隨機在（-15，+15）旋轉圖片

可以看到在 Val Acc 都介於 0.75 ~ 0.78 之間，比起第一題沒有做加強的

data set 較為穩定且準確（第一題仍有 74% 左右的正確率）

```
[140/150] 41.77 sec(s) Train Acc: 0.982060 Loss: 0.000489 | Val Acc: 0.774344 loss: 0.008477
[141/150] 41.77 sec(s) Train Acc: 0.986925 Loss: 0.000327 | Val Acc: 0.775802 loss: 0.009102
[142/150] 41.80 sec(s) Train Acc: 0.989155 Loss: 0.000302 | Val Acc: 0.765306 loss: 0.009590
[143/150] 41.78 sec(s) Train Acc: 0.990168 Loss: 0.000340 | Val Acc: 0.772012 loss: 0.009959
[144/150] 41.78 sec(s) Train Acc: 0.975674 Loss: 0.000741 | Val Acc: 0.769388 loss: 0.009717
[145/150] 41.78 sec(s) Train Acc: 0.966957 Loss: 0.000936 | Val Acc: 0.771429 loss: 0.008438
[146/150] 41.76 sec(s) Train Acc: 0.982465 Loss: 0.000472 | Val Acc: 0.768513 loss: 0.009188
[147/150] 41.76 sec(s) Train Acc: 0.978715 Loss: 0.000693 | Val Acc: 0.761808 loss: 0.008613
[148/150] 41.77 sec(s) Train Acc: 0.980641 Loss: 0.000493 | Val Acc: 0.755394 loss: 0.008797
[149/150] 41.79 sec(s) Train Acc: 0.988040 Loss: 0.000316 | Val Acc: 0.779592 loss: 0.009112
```

b. normalization

接著我們進行對已經加強過的 data 再採用 normalization 的方式，採用方法為

transforms.Normalize(mean = (0.5, 0.5, 0.5), std = (0.5, 0.5, 0.5)) # 歸一化到 [-1, 1]

可以看到經過 normalization 後並沒有對準確度有太大影響。有將此 model 預測出的 test 上傳到 kaggle，沒想到只有 0.36102 準確度… 。想了想一下，我將 testing_set 也做了 augmentation 以及 normalization，重新預測一次得到 0.83801 準確度！或許連 testing_set 一起做 augmentation 後能夠讓機器更感受到重要的部分。

```
[141/150] 41.67 sec(s) Train Acc: 0.967768 Loss: 0.000996 | Val Acc: 0.774052 loss: 0.008021
[142/150] 41.68 sec(s) Train Acc: 0.980945 Loss: 0.000522 | Val Acc: 0.775510 loss: 0.008022
[143/150] 41.67 sec(s) Train Acc: 0.989966 Loss: 0.000250 | Val Acc: 0.783090 loss: 0.008665
[144/150] 41.68 sec(s) Train Acc: 0.991891 Loss: 0.000275 | Val Acc: 0.771429 loss: 0.009215
[145/150] 41.65 sec(s) Train Acc: 0.969694 Loss: 0.000886 | Val Acc: 0.757434 loss: 0.008925
[146/150] 41.68 sec(s) Train Acc: 0.989053 Loss: 0.000348 | Val Acc: 0.749854 loss: 0.010212
[147/150] 41.68 sec(s) Train Acc: 0.979526 Loss: 0.000574 | Val Acc: 0.750146 loss: 0.011114
[148/150] 41.68 sec(s) Train Acc: 0.972735 Loss: 0.000753 | Val Acc: 0.758892 loss: 0.010462
[149/150] 41.67 sec(s) Train Acc: 0.982870 Loss: 0.000452 | Val Acc: 0.765598 loss: 0.009575
[150/150] 41.68 sec(s) Train Acc: 0.987736 Loss: 0.000341 | Val Acc: 0.772886 loss: 0.009431
```

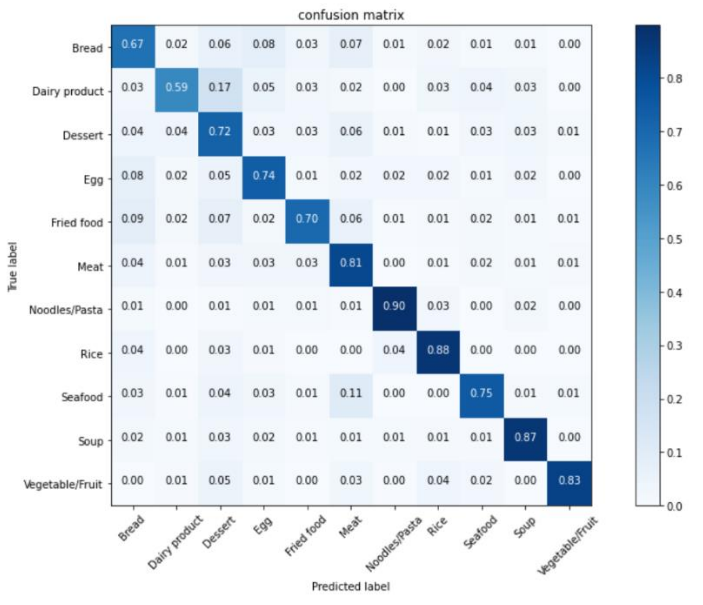
6. 觀察答錯的圖片中，哪些 class 彼此間容易用混？[繪出 confusion matrix 分析](1%)

ANS:

我們觀察以下矩陣，建立一個 prefalse_list 包含（X, Y）表示 True_label 為 X，然而我們錯誤預測此 Predicted_label 為 Y，且此機率發生為 0.05 以上。容易混淆的原因可能是 feature 太相同，因此造成錯誤判斷。

Prefalse_list：

1. (Bread, Dessert)/(Bread, Egg)/(Bread, Meat)
2. (Dairy product, Dessert) #奶製品與甜點相似度確實高，錯誤率最高/(Dairy product, Egg)
3. (Dessert, Meat) #令人意外的結果，因為兩者相似度直覺上不高
4. (Egg, Bread)/(Egg, Dessert)
5. (Fried food, Bread)/(Fried food, Dessert)/(Fried food, Meat)
6. (Vegetable/Fruit, Dessert)



附註：最後一題由於只使用 train_set 得到我們的 confusion_matrix，但是此模型我並沒有存下來，不過我接下來使用的模型、參數都是一樣的，因此繪製 confusion_matrix 時不會與這張圖差太多。