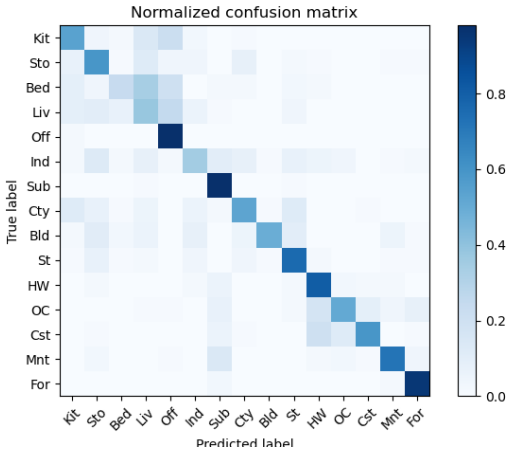
Computer Vision HW2 Report

Student ID: R08222017
Name: 陳韋辰

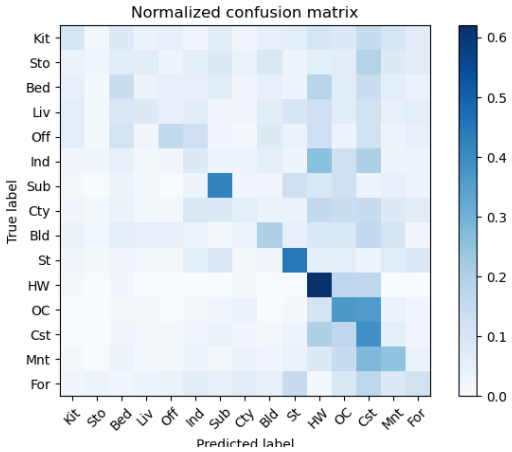
Part 1. (10%)

• Plot confusion matrix of two settings. (i.e. Bag of sift and tiny image representation) (5%)
Ans:

Bag of sift(Acc.=0.628):



Tiny image(Acc.=0.231):



• Compare the results/accuracy of both settings and explain the result. (5%)
Ans:

使用 **Bag of sift 的方法可以獲得較高的 accuracy**，因為在 Bag of sift 中，是使用 SIFT 的演算法將 feature 抽取出來，並與也是從 SIFT 和 kmeans 抓出來的 vocabulary 進行距離分類，統計出柱狀圖後來描述該圖片。這樣的方法可以獲得較有意義區域 (像是角落、變化大的地方等等)的 feature，**有著旋轉、強度、視角等不變性**，並且以統計 edge 方向的方式來表示 feature，另外 DoG detector 及 major rotation 的使用更讓 SIFT 的 feature **有大小和角度的不變性**。相反的 Tiny image 是直接將圖片 resize，來表示該圖片，單一一個 feature 只能表示特定區塊的線性組合，並沒有大小、角度、旋轉、視角的不變性，

很難針對不同圖片做比較，kNN 的判斷也就較差。

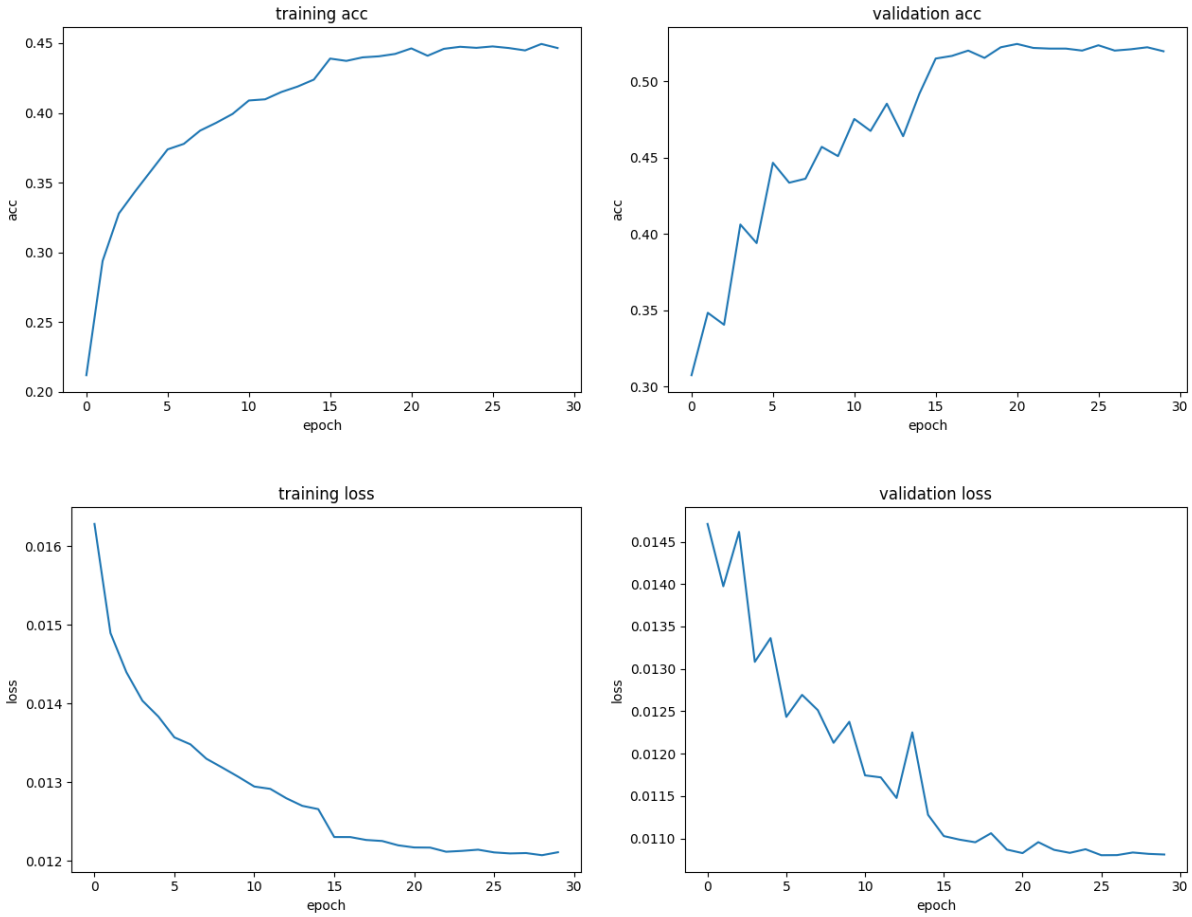
在 Confusion Matrix，對角線代表預測對的位置，而其他區域則代表預測到其他地點，**Bag of sift 的對角線明顯比 Tiny image 的對角線來的清楚，代表預測的結果較好**。其中可以發現 Bag of sift 在對角線附近也有較深的區域，代表 Bag of sift 容易將類似區域的地點認錯，像是 Bedroom-LivingRoom 的顏色很深，因為是很像的場景。Tiny image 的右下角較深，代表 Tiny image 對於室外的場景容易互相搞混，但可以跟室內區分，可能是因為室外照片很多都有大片天空跟地板的相似特性。

Part 2. (35%)

• Compare the performance on residual networks and LeNet. Plot the learning curve (loss and accuracy) on both training and validation sets for both 2 schemes. 8 plots in total. (20%)

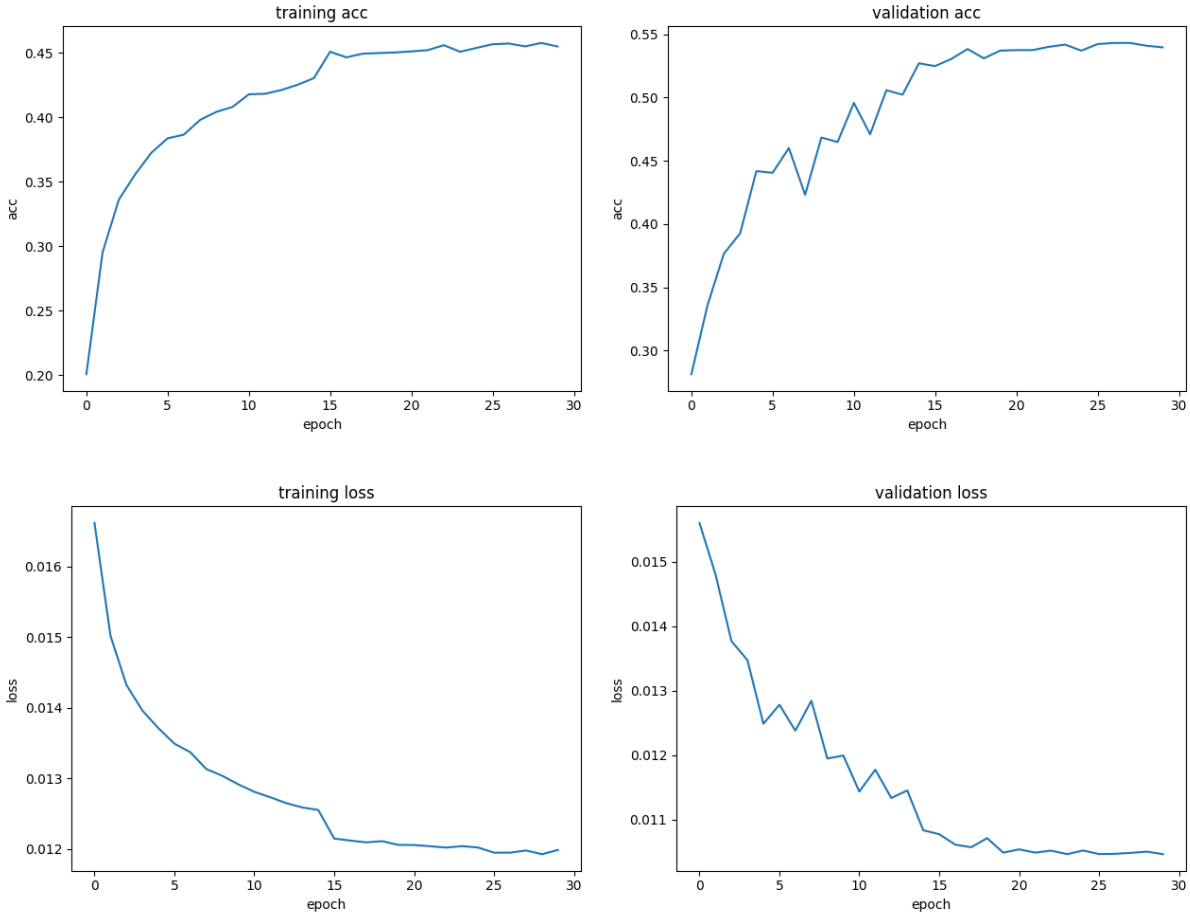
Ans:

LeNet:



```
myLeNet(
  (nrb1): no_residual_block(
    (conv1): Sequential(
      (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (relu): ReLU()
  )
  (conv1): Sequential(
    (0): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (nrb2): no_residual_block(
    (conv1): Sequential(
      (0): Conv2d(6, 6, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (relu): ReLU()
  )
  (conv2): Sequential(
    (0): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (nrb3): no_residual_block(
    (conv1): Sequential(
      (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (relu): ReLU()
  )
  (fc1): Sequential(
    (0): Linear(in_features=400, out_features=120, bias=True)
    (1): ReLU()
  )
  (fc2): Sequential(
    (0): Linear(in_features=120, out_features=84, bias=True)
    (1): ReLU()
  )
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

Resnet:



```
myResnet(
  (rb1): residual_block(
    (conv1): Sequential(
      (0): Conv2d(3, 3, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (bn): BatchNorm2d(3, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU()
  )
  (conv1): Sequential(
    (0): Conv2d(3, 6, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (rb2): residual_block(
    (conv1): Sequential(
      (0): Conv2d(6, 6, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (bn): BatchNorm2d(6, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU()
  )
  (conv2): Sequential(
    (0): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (rb3): residual_block(
    (conv1): Sequential(
      (0): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (bn): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU()
  )
  (fc1): Sequential(
    (0): Linear(in_features=400, out_features=120, bias=True)
    (1): ReLU()
  )
  (fc2): Sequential(
    (0): Linear(in_features=120, out_features=84, bias=True)
    (1): ReLU()
  )
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

在 Resnet 實作上，我先使用兩層 convolutional layer 和三層的 residual block 做 convolution，最後使用三層的 fully connection layer 輸出 10 channel 的 output，作為 Cifar-10 mini 的分類。residual block 是使用一層的 convolutional layer，並在接下一層之前將原本的 input 加上過 convolution 的 output，建立 Residual Network，讓 back propagation 有更短的路徑可用。而為了更有系統的比較，LeNet 的實作也同樣做了類似 residual block 的 layer，但沒有 Residual 的動作。

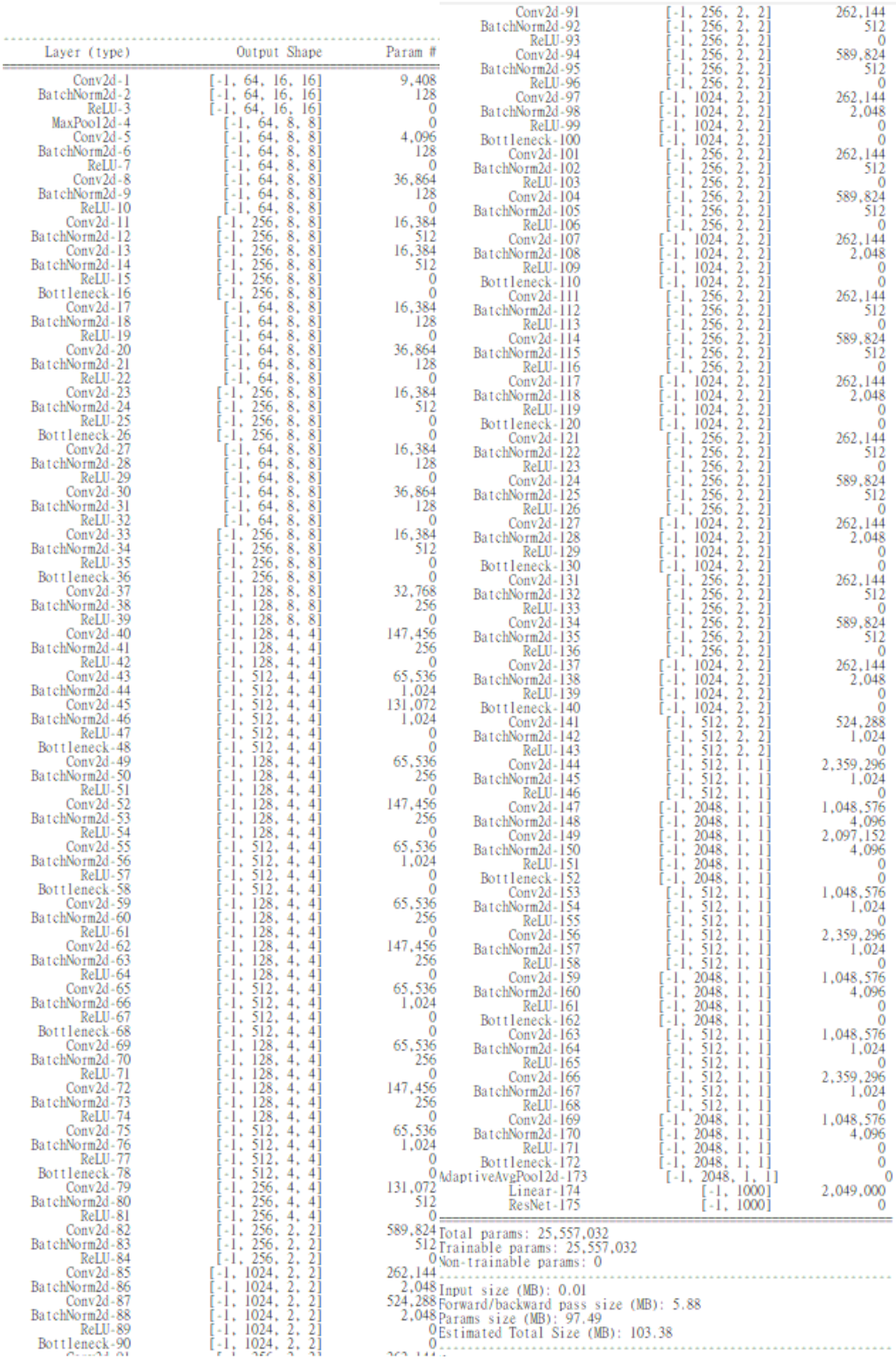
從 training 的數據上，Resnet 在第 9 個 epoch 就達到 40%的正確率，而 LeNet 則在第 11 個 epoch 才達到 40%的正確率，且直到第 30 個 epoch，Resnet 達到了 45.7%的正確率，而 LeNet 只有 44.9%。而在 validation 的數據上也是，Resnet 最高獲得 54.3%的正確率，而 LeNet 最高只有 52.4%的正確率，**正確率表現上 Resnet 都比 LeNet 來的好**。在最終 loss 的計算上也是 LeNet 的 training(0.012 和 0.011)和 vlidation 值(0.0108 和 0.0104)都比

Resnet 來的大，說明 **Resnet 可以更有效的計算 gradient，使得 loss function 下降**。

理論上 Residual Network 的使用可以幫助 back propagation 的計算，讓多層的 model 有辦法被訓練到，增加訓練的效果，推測在這邊差距沒有很明顯的原因是因為，這邊所使用的模型較為簡單，所以沒辦法很明顯感受到 Residual Network 的優勢，**如果將模型做多層一點，甚至加入跨多層的 Residual Network，那訓練的結果會出現更大的差距。**

• Attach basic information of the model you use including model architecture and number of the parameters. (5%)
Ans:

在自訂 model 中，我直接使用了 **torchvision.models 中的 resnet50** 作為我的 pre-trained model (pretrained 設為 True，以減少訓練時間，並增加訓練成效)，並沒有使用額外的 network 去連接，**總參數量為 25,557,032，大小為 97.7 MB**，也因為逼近於作業規定的上限，在輸出端我直接捨棄 resnet50 其他的 990 個 output，只用前 10 個 output 訓練，但因為 resnet50 的極高品質，使我的 model 不用做額外的調整也可以達到 Strong baseline，如果在未來要以更高的 baseline 為目標，且沒有 model size 限制情況下，我會選擇在 resnet50 後面加上數層 fully connected layer，讓 output 的 channel 數剛好為 10，加強模型的效益。模型視覺化的結果如下圖所示：



Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 16, 16]	9,408
BatchNorm2d-2	[-1, 64, 16, 16]	128
ReLU-3	[-1, 64, 16, 16]	0
MaxPool2d-4	[-1, 64, 8, 8]	0
Conv2d-5	[-1, 64, 8, 8]	4,096
BatchNorm2d-6	[-1, 64, 8, 8]	128
ReLU-7	[-1, 64, 8, 8]	0
Conv2d-8	[-1, 64, 8, 8]	36,864
BatchNorm2d-9	[-1, 64, 8, 8]	128
ReLU-10	[-1, 64, 8, 8]	0
Conv2d-11	[-1, 256, 8, 8]	16,384
BatchNorm2d-12	[-1, 256, 8, 8]	512
Conv2d-13	[-1, 256, 8, 8]	16,384
BatchNorm2d-14	[-1, 256, 8, 8]	512
ReLU-15	[-1, 256, 8, 8]	0
Bottleneck-16	[-1, 256, 8, 8]	0
Conv2d-17	[-1, 64, 8, 8]	16,384
BatchNorm2d-18	[-1, 64, 8, 8]	128
ReLU-19	[-1, 64, 8, 8]	0
Conv2d-20	[-1, 64, 8, 8]	36,864
BatchNorm2d-21	[-1, 64, 8, 8]	128
ReLU-22	[-1, 64, 8, 8]	0
Conv2d-23	[-1, 256, 8, 8]	16,384
BatchNorm2d-24	[-1, 256, 8, 8]	512
ReLU-25	[-1, 256, 8, 8]	0
Bottleneck-26	[-1, 256, 8, 8]	0
Conv2d-27	[-1, 64, 8, 8]	16,384
BatchNorm2d-28	[-1, 64, 8, 8]	128
ReLU-29	[-1, 64, 8, 8]	0
Conv2d-30	[-1, 64, 8, 8]	36,864
BatchNorm2d-31	[-1, 64, 8, 8]	128
ReLU-32	[-1, 64, 8, 8]	0
Conv2d-33	[-1, 256, 8, 8]	16,384
BatchNorm2d-34	[-1, 256, 8, 8]	512
ReLU-35	[-1, 256, 8, 8]	0
Bottleneck-36	[-1, 256, 8, 8]	0
Conv2d-37	[-1, 128, 8, 8]	32,768
BatchNorm2d-38	[-1, 128, 8, 8]	256
ReLU-39	[-1, 128, 8, 8]	0
Conv2d-40	[-1, 128, 4, 4]	147,456
BatchNorm2d-41	[-1, 128, 4, 4]	256
ReLU-42	[-1, 128, 4, 4]	0
Conv2d-43	[-1, 512, 4, 4]	65,536
BatchNorm2d-44	[-1, 512, 4, 4]	1,024
Conv2d-45	[-1, 512, 4, 4]	131,072
BatchNorm2d-46	[-1, 512, 4, 4]	1,024
ReLU-47	[-1, 512, 4, 4]	0
Bottleneck-48	[-1, 512, 4, 4]	0
Conv2d-49	[-1, 128, 4, 4]	65,536
BatchNorm2d-50	[-1, 128, 4, 4]	256
ReLU-51	[-1, 128, 4, 4]	0
Conv2d-52	[-1, 128, 4, 4]	147,456
BatchNorm2d-53	[-1, 128, 4, 4]	256
ReLU-54	[-1, 128, 4, 4]	0
Conv2d-55	[-1, 512, 4, 4]	65,536
BatchNorm2d-56	[-1, 512, 4, 4]	1,024
ReLU-57	[-1, 512, 4, 4]	0
Bottleneck-58	[-1, 512, 4, 4]	0
Conv2d-59	[-1, 128, 4, 4]	65,536
BatchNorm2d-60	[-1, 128, 4, 4]	256
ReLU-61	[-1, 128, 4, 4]	0
Conv2d-62	[-1, 128, 4, 4]	147,456
BatchNorm2d-63	[-1, 128, 4, 4]	256
ReLU-64	[-1, 128, 4, 4]	0
Conv2d-65	[-1, 512, 4, 4]	65,536
BatchNorm2d-66	[-1, 512, 4, 4]	1,024
ReLU-67	[-1, 512, 4, 4]	0
Bottleneck-68	[-1, 512, 4, 4]	0
Conv2d-69	[-1, 128, 4, 4]	65,536
BatchNorm2d-70	[-1, 128, 4, 4]	256
ReLU-71	[-1, 128, 4, 4]	0
Conv2d-72	[-1, 128, 4, 4]	147,456
BatchNorm2d-73	[-1, 128, 4, 4]	256
ReLU-74	[-1, 128, 4, 4]	0
Conv2d-75	[-1, 512, 4, 4]	65,536
BatchNorm2d-76	[-1, 512, 4, 4]	1,024
ReLU-77	[-1, 512, 4, 4]	0
Bottleneck-78	[-1, 512, 4, 4]	0
Conv2d-79	[-1, 256, 4, 4]	131,072
BatchNorm2d-80	[-1, 256, 4, 4]	512
ReLU-81	[-1, 256, 4, 4]	0
Conv2d-82	[-1, 256, 2, 2]	589,824
BatchNorm2d-83	[-1, 256, 2, 2]	512
ReLU-84	[-1, 256, 2, 2]	0
Conv2d-85	[-1, 1024, 2, 2]	262,144
BatchNorm2d-86	[-1, 1024, 2, 2]	2,048
Conv2d-87	[-1, 1024, 2, 2]	524,288
BatchNorm2d-88	[-1, 1024, 2, 2]	2,048
ReLU-89	[-1, 1024, 2, 2]	0
Bottleneck-90	[-1, 1024, 2, 2]	0
Conv2d-91	[-1, 256, 2, 2]	262,144
BatchNorm2d-92	[-1, 256, 2, 2]	512
ReLU-93	[-1, 256, 2, 2]	0
Conv2d-94	[-1, 256, 2, 2]	589,824
BatchNorm2d-95	[-1, 256, 2, 2]	512
ReLU-96	[-1, 256, 2, 2]	0
Conv2d-97	[-1, 1024, 2, 2]	262,144
BatchNorm2d-98	[-1, 1024, 2, 2]	2,048
ReLU-99	[-1, 1024, 2, 2]	0
Bottleneck-100	[-1, 1024, 2, 2]	0
Conv2d-101	[-1, 256, 2, 2]	262,144
BatchNorm2d-102	[-1, 256, 2, 2]	512
ReLU-103	[-1, 256, 2, 2]	0
Conv2d-104	[-1, 256, 2, 2]	589,824
BatchNorm2d-105	[-1, 256, 2, 2]	512
ReLU-106	[-1, 256, 2, 2]	0
Conv2d-107	[-1, 1024, 2, 2]	262,144
BatchNorm2d-108	[-1, 1024, 2, 2]	2,048
ReLU-109	[-1, 1024, 2, 2]	0
Bottleneck-110	[-1, 1024, 2, 2]	0
Conv2d-111	[-1, 256, 2, 2]	262,144
BatchNorm2d-112	[-1, 256, 2, 2]	512
ReLU-113	[-1, 256, 2, 2]	0
Conv2d-114	[-1, 256, 2, 2]	589,824
BatchNorm2d-115	[-1, 256, 2, 2]	512
ReLU-116	[-1, 256, 2, 2]	0
Conv2d-117	[-1, 1024, 2, 2]	262,144
BatchNorm2d-118	[-1, 1024, 2, 2]	2,048
ReLU-119	[-1, 1024, 2, 2]	0
Bottleneck-120	[-1, 1024, 2, 2]	0
Conv2d-121	[-1, 256, 2, 2]	262,144
BatchNorm2d-122	[-1, 256, 2, 2]	512
ReLU-123	[-1, 256, 2, 2]	0
Conv2d-124	[-1, 256, 2, 2]	589,824
BatchNorm2d-125	[-1, 256, 2, 2]	512
ReLU-126	[-1, 256, 2, 2]	0
Conv2d-127	[-1, 1024, 2, 2]	262,144
BatchNorm2d-128	[-1, 1024, 2, 2]	2,048
ReLU-129	[-1, 1024, 2, 2]	0
Bottleneck-130	[-1, 1024, 2, 2]	0
Conv2d-131	[-1, 256, 2, 2]	262,144
BatchNorm2d-132	[-1, 256, 2, 2]	512
ReLU-133	[-1, 256, 2, 2]	0
Conv2d-134	[-1, 256, 2, 2]	589,824
BatchNorm2d-135	[-1, 256, 2, 2]	512
ReLU-136	[-1, 256, 2, 2]	0
Conv2d-137	[-1, 1024, 2, 2]	262,144
BatchNorm2d-138	[-1, 1024, 2, 2]	2,048
ReLU-139	[-1, 1024, 2, 2]	0
Bottleneck-140	[-1, 1024, 2, 2]	0
Conv2d-141	[-1, 512, 2, 2]	524,288
BatchNorm2d-142	[-1, 512, 2, 2]	1,024
ReLU-143	[-1, 512, 2, 2]	0
Conv2d-144	[-1, 512, 1, 1]	2,359,296
BatchNorm2d-145	[-1, 512, 1, 1]	1,024
ReLU-146	[-1, 512, 1, 1]	0
Conv2d-147	[-1, 2048, 1, 1]	1,048,576
BatchNorm2d-148	[-1, 2048, 1, 1]	4,096
Conv2d-149	[-1, 2048, 1, 1]	2,097,152
BatchNorm2d-150	[-1, 2048, 1, 1]	4,096
ReLU-151	[-1, 2048, 1, 1]	0
Bottleneck-152	[-1, 2048, 1, 1]	0
Conv2d-153	[-1, 512, 1, 1]	1,048,576
BatchNorm2d-154	[-1, 512, 1, 1]	1,024
ReLU-155	[-1, 512, 1, 1]	0
Conv2d-156	[-1, 512, 1, 1]	2,359,296
BatchNorm2d-157	[-1, 512, 1, 1]	1,024
ReLU-158	[-1, 512, 1, 1]	0
Conv2d-159	[-1, 2048, 1, 1]	1,048,576
BatchNorm2d-160	[-1, 2048, 1, 1]	4,096
ReLU-161	[-1, 2048, 1, 1]	0
Bottleneck-162	[-1, 2048, 1, 1]	0
Conv2d-163	[-1, 512, 1, 1]	1,048,576
BatchNorm2d-164	[-1, 512, 1, 1]	1,024
ReLU-165	[-1, 512, 1, 1]	0
Conv2d-166	[-1, 512, 1, 1]	2,359,296
BatchNorm2d-167	[-1, 512, 1, 1]	1,024
ReLU-168	[-1, 512, 1, 1]	0
Conv2d-169	[-1, 2048, 1, 1]	1,048,576
BatchNorm2d-170	[-1, 2048, 1, 1]	4,096
ReLU-171	[-1, 2048, 1, 1]	0
Bottleneck-172	[-1, 2048, 1, 1]	0
AdaptiveAvgPool2d-173	[-1, 2048, 1, 1]	0
Linear-174	[-1, 1000]	2,049,000
ResNet-175	[-1, 1000]	0

Total params: 25,557,032
Trainable params: 25,557,032
Non-trainable params: 0

Input size (MB): 0.01
Forward/backward pass size (MB): 5.88
Params size (MB): 97.49
Estimated Total Size (MB): 103.38

• Briefly describe what method do you apply? (e.g. data augmentation, model architecture, loss function, semi-supervised etc.) (10%)
Ans:

在 Loss Function 上我使用 **torch.nn 的 CrossEntropyLoss** 作為我的基準，並使用

torch.optim 的 SGD 作為我的優化器。除了使用上述提到的 **pre-trained model**，提升訓

練效率外，我還使用了 **data augmentation** 和 **data cleaning** 的手法，提升訓練成果，實驗如下：

1. data augmentation (以下實驗 epoch=4，batch size=256，無 data cleaning)

Type of transforms in torchvision.transforms	Training acc.	Validation acc.
None	0.81	0.687
transforms.RandomAffine(degrees=3, translate=(0.07,0.07), scale=(0.9,1.1), shear=N	0.7275	0.707
transforms.RandomPerspective(distortion_scale=0.5, p=0.07),	0.7792	0.7061
transforms.RandomAdjustSharpness(sharpness_factor=3, p=0.05),	0.8129	0.703
transforms.RandomCrop(25)	0.6943	0.643
ransforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0),	0.8073	0.7009
extra_transforms_set	0.6814	0.7247

使用以上**橘色區間的 transform 可以避免 over training**，增加 validation 的正確率，最終我使用了合成的 transforms 將 validation 的正確率提升到了最高，其 transforms 模型如下，測試下來最好的成果(Validation accuracy = 0.7247)：

Type of transforms in torchvision.transforms	proportion of total data
transforms.RandomHorizontalFlip(p=0.5), transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0), transforms.RandomAffine(degrees=5, translate=(0.1,0.1), scale=(0.9,1.1), shear=None), transforms.RandomAdjustSharpness(sharpness_factor=3, p=0.05),	1
transforms.RandomHorizontalFlip(p=0.5), transforms.ColorJitter(brightness=0.1, contrast=0.1, saturation=0.1, hue=0), transforms.GaussianBlur(kernel_size=(7, 7), sigma=(0.1, 3)), transforms.RandomAffine(degrees=70, translate=(0.2,0.2), scale=(0.7,1.3), shear=None),	0.3
transforms.RandomAffine(degrees=50, translate=(0.2,0.2), scale=(0.7,1.3), shear=None), transforms.GaussianBlur(kernel_size=(7, 7), sigma=(0.1, 3)), transforms.RandomHorizontalFlip(p=0.5),	0.3
	1.6

2. data cleaning (以下實驗 epoch=4，batch size=256，無 data augmentation)

在作業的數據中有 3000 張是 dirty images，占總 data 數為 13%，會對訓練成果造成很

大的影響，因此必須去除，我使用 torchvision.models 中的 resnet50 作為我的 classifying model，**只訓練 6 個 epoch，並將 pretrained 設為 False，避免變成 strong model，造成 overtraining**，最終取最大 probability 最小的 4000 個 data 刪除，代表可能是不容易被分類的 dirty images，實驗如下：

	Training acc.	Validation acc.
None	0.7179	0.7013
clean 1000 data	0.7342	0.7186
clean 2000 data	0.7328	0.7262
clean 4000 data	0.7486	0.7337

發現在取 4000 比 data 時，可以獲得最好的正確率增加。

•實驗結果：

最終使用 epoch=40，batch size=32，learning rate=0.01，milestones= [15, 25, 30]，gamma=0.1，以及上述的方法，達到在 public testing data 上有 82.72%的正確率，學習曲線如下：