- 不同的 tuning 方式的原理及實作

Image Normalization:

利用 Standardization 讓取樣的 dataset 中的平均數(mean)、標準差(std, standard deviation)，經過轉換之後分別趨近於 0 和 1，減少訓練過程中產生偏差，以及訓練的結果被某部分資料支配，接這再用 transforms.Normalization(mean, std)，將圖片進行正規化。

其中平均數(mean)接近 0 是為了讓 unbiased 的 data 更有利於學習，而標準差接近 1 是為了減緩梯度消失(gradient vanishing)和梯度爆炸(gradient explosion)的問題。(Figure 1、2)

```
# 計算normalization需要的mean & std
def get_mean_std(dataset, ratio=0.3):
    # Get mean and std by sample ratio
    dataloader = torch.utils.data.DataLoader(dataset, batch_size=int(len(dataset)*ratio), shuffle=True, num_workers=2)

    data = iter(dataloader).next()[0]          # get the first iteration data
    mean = np.mean(data.numpy(), axis=(0,2,3))
    std = np.std(data.numpy(), axis=(0,2,3))
    return mean, std

train_dataset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transforms.ToTensor())

train_mean, train_std = get_mean_std(train_dataset)

print(train_mean, train_std)
```

Figure 1

```
transform_train = transforms.Compose([
    # data augmentation
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    # transforms.Resize(224),
    # transforms.RandomCrop(192),
    # transforms.RandomHorizontalFlip(),
    # transforms.RandomRotation(0.2),
    # transforms.ColorJitter(brightness=0.5),
    # transforms.ColorJitter(contrast=0.5),
    transforms.ToTensor(),

    # data normalization       # standardization: (image - train_mean) / train_std
    # transforms.Lambda(lambda t: t.expand(3, -1, -1)),
    # transforms.Normalize((0.5,) * 3, (0.5,) * 3)
    transforms.Normalize(train_mean, train_std)
])
```

Figure 2

Data augmentation

在 training sample 固定的情況下，使用 data argumentation 可以增加 training sample 的多樣性或是解決 training sample 不夠多的問題；作法是將被辨識物件(training sample)進行一些調整，例如: 旋轉、縮放、取灰階…等。

實作部分使用了 transforms.RandomCrop(32, padding=4)，將圖片作隨機裁切，輸出尺寸為 32*32 的正方形，上下左右加入 padding 作填充；並

且使用 transforms.RandomHorizontalFlip()將輸入圖片隨機作水平翻轉。(Figure 3)

```
transform_train = transforms.Compose([
        # data augmentation

        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),

        transforms.ToTensor(),
        # data normalization        # standardization: (image - train_mean) / train_std
        transforms.Normalize(train_mean, train_std)

])
```

Figure 3

Learning Rate

使用 SGD 演算法每 30 個 epoch 作一次 learning rate 更新(Figure 4、5、6)

```
def adjust_learning_rate(optim, epoch, lr):
        # define your lr scheduler
        lr = lr * (0.1 ** (epoch //30 ) )
        for param_group in optim.param_groups:        # change the lr to what you define
                param_group['lr'] = lr
```

Figure 4

```
#optimizer = optim.SGD(net.parameters(), lr=lr, momentum=0.9, weight_decay=5e-4)      # momentum-SGD，採用L2正則化（權重衰減）
optimizer = optim.Adam(net.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False)
```

Figure 5

```
adjust_learning_rate(optimizer, epoch, lr)
print("learning rate: ", optimizer.param_groups[0]['lr'])
```

Figure 6

- 比較不同 tuning 方式如何造成影響

加上 SGD、augmentation、normalization 後 vs Baseline model

在 baseline model 中各項表現如下表格(Table 1)，雖然 training 表現不錯，但是在 validation 和 test 中，準確度只有 0.71 左右，而 loss 更是高達 1.08 左右。

train-acc	train-loss	val-acc	val-loss	test-acc	val-loss
0.935	0.184	0.713	1.079	0.714	1.089

Table 1

將資料預處理經過 augmentation、normalization，learning rate 以 SGD algorithm 取代固定 learning rate，epoch 40 次(Table 2)和提升至 320 次(Table 3)之後，各項表現如下，不論是 training、validation、test 的準確度皆有顯著提升，而 loss 也有顯著的下降。

可以得知，將資料經過 data argumentation、normalization，再加上使用

SGD algorithm，model 整體表現有顯著的提升。

train-acc	train-loss	val-acc	val-loss	test-acc	val-loss
0.882	0.346	0.830	0.502	0.825	0.507

Table 2

train-acc	train-loss	val-acc	val-loss	test-acc	val-loss
0.972	0.082	0.86	0.49	0.863	0.505

Table 3

不同 learning rate

在 Epoch、data-preprocessing、gradient descent 皆相同的情況下，測試了使用不同 gradient descent 所訓練的 model 差異，Table 3 為使用 SGD algorithm 的各項數據，Table 4 為使用 Adam algorithm 的各項數據，與使用 SGD algorithm 訓練的 mode 相比較，使用 Adam 所訓練的 model，training accuracy 明顯表現較不好，而 validation、test set 的 accuracy 表現也有些微下降；在 loss 的比較上，training loss 有很顯著較差的表現，而 validation、test set 上 loss 則沒有顯著差異。

另外，使用 SGD algorithm 的 model(Figure 7、8)，training set 和 validation/test set 的 accuracy 和 loss 差距較大，而在使用 Adam 的 model(Figure 9、10)上 training set 和 validation/test set 的 accuracy 和 loss 之間的差距相對較小，前期(Epoch 數少)的震盪也比較少。

train-acc	train-loss	val-acc	val-loss	test-acc	val-loss
0.876	0.367	0.815	0.584	0.83	0.549

Table 4

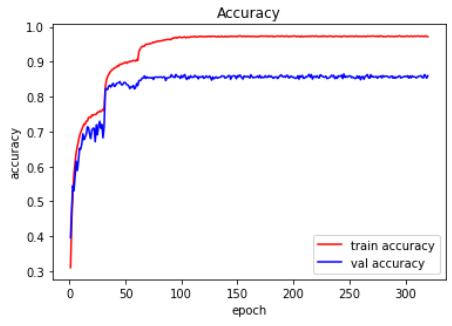


Figure 7

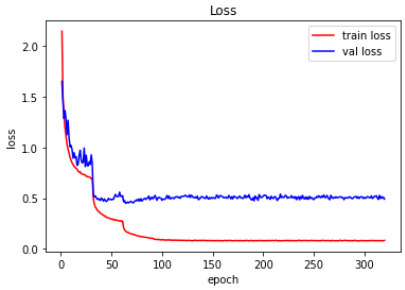


Figure 8

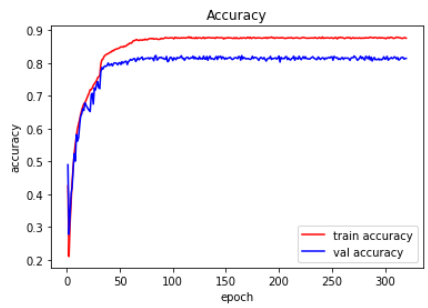


Figure 9

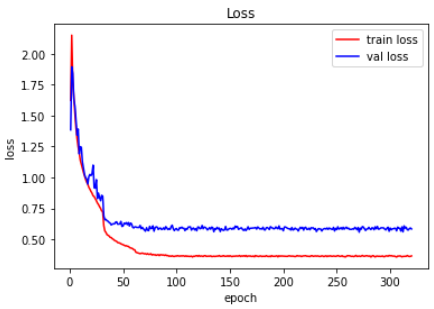


Figure 10

不同 batch size

在 Epoch、data-preprocessing、gradient descent 皆相同的情況下，測試了 batch size 128 和 256 的不同 model，Table 3 為 batch size=128 的各項數據，Table 5 為 batch size=256 的各項數據，可以看出 batch size 大小差距兩倍的情況下，各項數據並沒有很顯著的差異，可能是因為 model 設計上有問題，或是已經達到目前設計 model 的瓶頸，進而沒有顯著差異。

train-acc	train-loss	val-acc	val-loss	test-acc	val-loss
0.972	0.084	0.849	0.548	0.863	0.516

Table 5

不同 Epoch

在 batch、data-preprocessing、gradient descent 皆相同的情況下，測試了 Epoch 320 和 160 的不同 model，Table 6 為 Epoch =320 的各項數據，Table 5 為 Epoch =160 的各項數據，可以看出 batch size 大小差距兩倍的情況下，各項數據並沒有很顯著的差異，可能是因為 model 設計上有問題，或是已經達到目前設計 model 的瓶頸，進而沒有顯著差異。

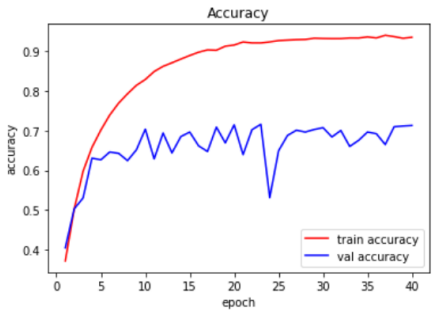
train-acc	train-loss	val-acc	val-loss	test-acc	val-loss
0.973	0.079	0.856	0.503	0.867	0.493

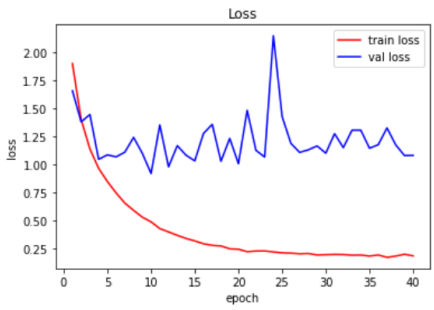
Table 6

- 截圖並說明各項結果

baseline model:

baseline model accuracy	baseline model loss





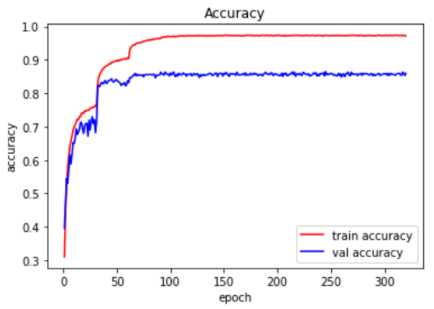
baseline model train-loss, val-loss, train-acc, val-acc

```
learning rate:  0.05
Train loss: 0.184 | Train acc: 0.935
Val loss: 1.079 | Val acc: 0.713
Test loss: 1.089 | Test acc: 0.714
```

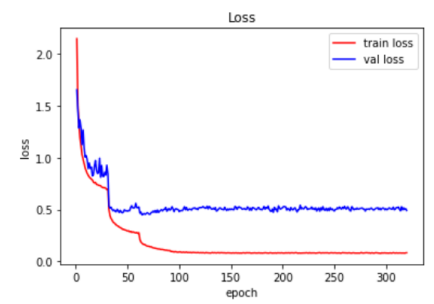
Model configuration:

1. Gradient-descent: SGD
2. Learning rate 更新頻率: 30 epoch
3. Batch size: 128
4. Epoch: 320

model accuracy



baseline model loss



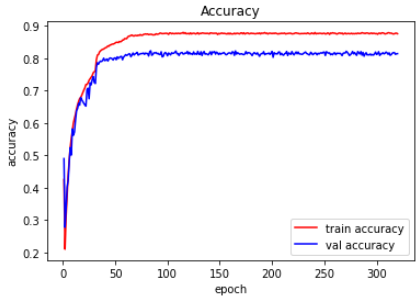
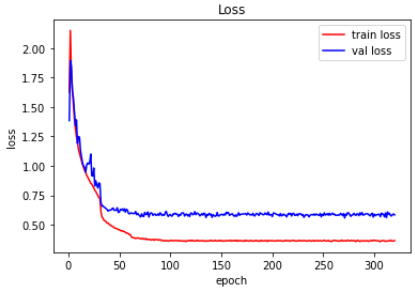
baseline model train-loss, val-loss, train-acc, val-acc

```
learning rate:  1.0000000000000006e-11
Train loss: 0.082 | Train acc: 0.972
Val loss: 0.490 | Val acc: 0.860
Test loss: 0.505 | Test acc: 0.863
```

不同 learning rate 對訓練的影響

Model configuration:

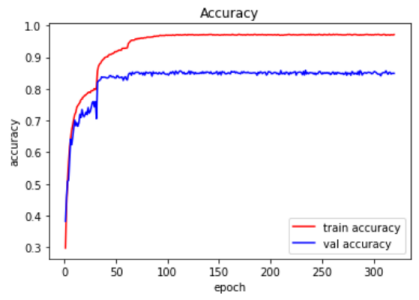
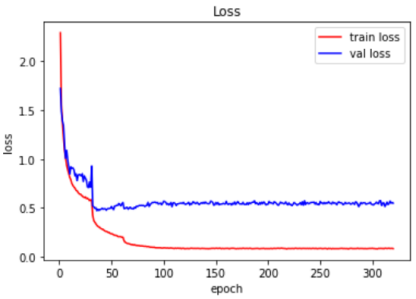
1. Gradient-descent: Adam
2. Learning rate 更新頻率: 30 epoch
3. Batch size: 128
4. Epoch: 320

model accuracy	baseline model loss
	
baseline model train-loss, val-loss, train-acc, val-acc Epoch: 320 learning rate: 1.0000000000000006e-11 Train loss: 0.367 | Train acc: 0.876 Val loss: 0.584 | Val acc: 0.815 Test loss: 0.549 | Test acc: 0.830	EPOCH = 320 BATCH_SIZE = 128 lr = 0.1 device = torch.device("cuda") optim.Adam(net.parameters(),

Batch size 增加對訓練的影響

Model configuration:

1. Gradient-descent: SGD
2. Learning rate 更新頻率: 30 epoch
3. Batch size: 256
4. Epoch: 320

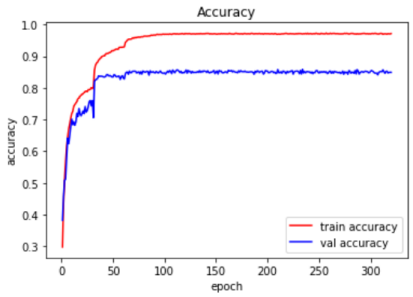
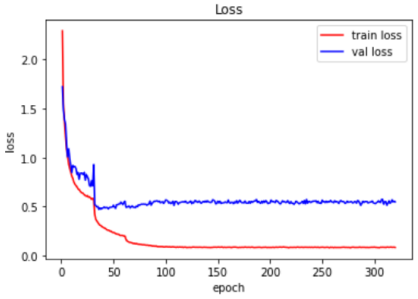
model accuracy	baseline model loss
	

baseline model
train-loss, val-loss, train-acc, val-acc
learning rate: 1.0000000000000006e-11
Train loss: 0.084 | Train acc: 0.972
Val loss: 0.548 | Val acc: 0.849
Test loss: 0.516 | Test acc: 0.863
	# setting parameter
	EPOCH = 320
	BATCH_SIZE = 256
	lr = 0.1
	device = torch.device("cuda")

Epoch 減少對訓練的影響

Model configuration:

1. Gradient-descent: SGD
2. Learning rate 更新頻率: 30 epoch
3. Batch size: 128
4. Epoch: 160

<table>
<tr>
<td>model accuracy
 
</td>
<td>baseline model loss
 
</td>
</tr>
<tr>
<td>baseline model
train-loss, val-loss, train-acc, val-acc
learning rate: 1.0000000000000004e-06
Train loss: 0.079 | Train acc: 0.973
Val loss: 0.503 | Val acc: 0.856
Test loss: 0.493 | Test acc: 0.867</td>
<td>

```
# setting parameter
EPOCH = 160
BATCH_SIZE = 128
lr = 0.1
device = torch.device("cuda")
```

</td>
</tr>
</table>