Lab4 Report

Introduction

用 CNN 做腦波圖形分類問題。使用 PyTorch 實作 EEGNet 和 DeepConvNet 這兩種 CNN。在 PyTorch 中，要使用 torch.nn 套件，此套件包含建立 CNN 的模組、可擴充類別和所有必要元件。建立完 model 後，只需要自己定義 forward 函式，PyTorch 會自行進行 backpropagate。

Experiment set up

A. The detail of your model

EEGNet

```
EEGNet(
  (firstconv): Sequential(
    (0): Conv2d(1, 16, kernel_size=(1, 51), stride=(1, 1), padding=(0, 25), bias=False)
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  )
  (depthwiseConv): Sequential(
    (0): Conv2d(16, 32, kernel_size=(2, 1), stride=(1, 1), groups=16, bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 4), stride=(1, 4), padding=0)
    (4): Dropout(p=0.25)
  )
  (separableConv): Sequential(
    (0): Conv2d(32, 32, kernel_size=(1, 15), stride=(1, 1), padding=(0, 7), bias=False)
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ELU(alpha=1.0)
    (3): AvgPool2d(kernel_size=(1, 8), stride=(1, 8), padding=0)
    (4): Dropout(p=0.25)
  )
  (classify): Sequential(
    (0): Linear(in_features=736, out_features=2, bias=True)
  )
)
```

基本上直接照著圖片上實作就好了。

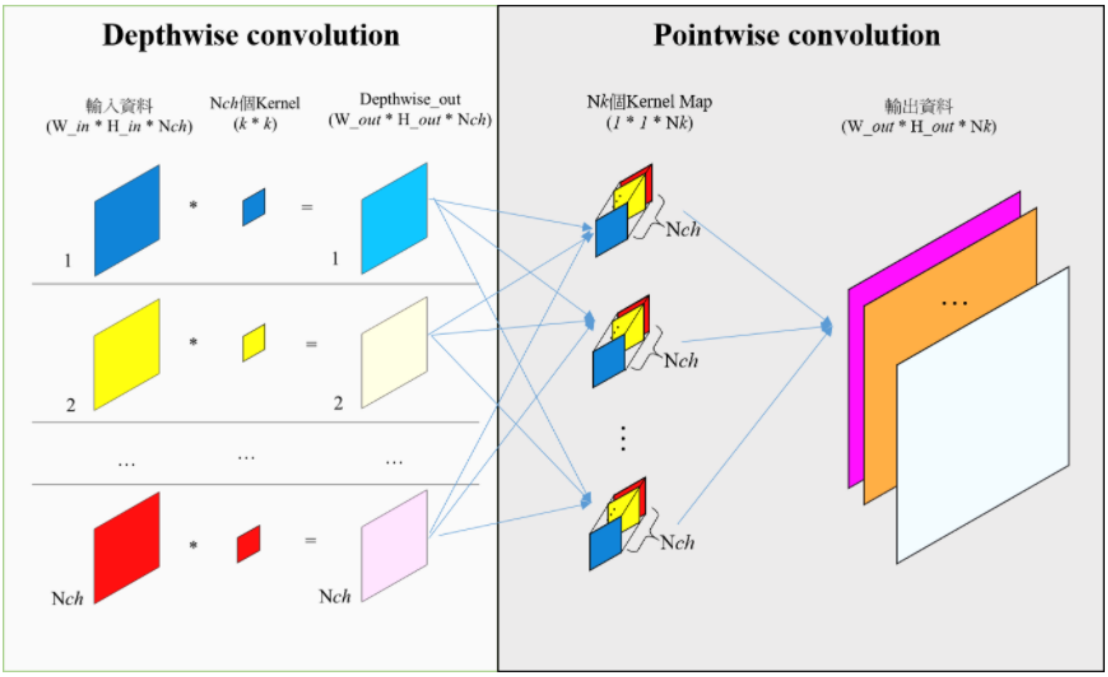
```python
import torch
import torch.nn as nn
import numpy

class EEGNet(nn.Module):
    def __init__(self, act_func, device):
        super(EEGNet, self).__init__()
        self.device = device
        self.firstconv = nn.Sequential(
            nn.Conv2d(1, 16, kernel_size = (1, 51), stride = (1, 1), padding = (0, 25), bias = False),
            nn.BatchNorm2d(16, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True)
        )
        self.depthwiseConv = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size = (2, 1), stride = (1, 1), groups = 16, bias = False),
            nn.BatchNorm2d(32, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            act_func,
            nn.AvgPool2d(kernel_size = (1, 4), stride = (1, 4), padding = 0),
            nn.Dropout(p = 0.25)
        )
        self.seperableConv = nn.Sequential(
            nn.Conv2d(32, 32, kernel_size = (1, 15), stride = (1, 1), padding = (0, 7), bias = False),
            nn.BatchNorm2d(32, eps = 1e-5, momentum = 0.1, affine = True, track_running_stats = True),
            act_func,
            nn.AvgPool2d(kernel_size = (1, 8), stride = (1, 8), padding = 0),
            nn.Dropout(p = 0.25)
        )
        self.classify = nn.Linear(736, 2)

    def forward(self, X):
        out = self.firstconv(X.to(self.device))
        out = self.depthwiseConv(out)
        out = self.seperableConv(out)
        out = out.view(out.shape[0], -1)  #resize
        out = self.classify(out)
        return out
```

其中，activation function 和 device 作為可傳入的參數。由 main 執行時傳入 ELU、ReLU 或 LeakyReLU 作為每層的 activation function，而 device 是為了同步資料存取的位置(CPU or GPU)。接下來一一介紹各層細節。

- Sequential 會依序執行括號內的內容。
- Conv2d 為 CNN 的核心，可以偵測影像中的各個 feature。並將輸出通道作為下一層的輸入。
- BatchNorm2d 圖層會在輸入上套用正規化，使其具有零平均值和單位變異數，並提高網路精確度。
- MaxPool 層可協助我們確保影像中的物件位置不會影響神經網路偵測其特定功能的能力。

- Dropout(p = 0.25)，這裡的 0.25 是指該層的神經元在每次迭代訓練時會隨機有 25%的可能性被丟棄。這是為了避免出現 overfitting。
- Linear 圖層是網路中的最後一層，作為 fully connected layer，會計算每個類別的分數。

而 EEGNet 中有用到 Depthwise Separable Convolution。所謂的 Depthwise Separable Convolution 是指針對輸入資料的每一個 Channel 都建立一個 k*k 的 Kernel，然後每一個 Channel 針對對應的 Kernel 都各自(分開)做 convolution。而一般的卷積計算是每個 Kernel Map 都要和所有 channel 都去做 convolution。下圖為計算流程示意圖:



和一般卷積的計算量比較:

$$\frac{\text{Depthwise separable convolution 計算量}}{\text{一般卷積計算量}}$$

$$= \frac{W_{in} * H_{in} * Nch * k * k + Nch * Nk * W\_in * H\_in}{W\_in * H\_in * Nch * k * k * Nk}$$

$$= \frac{1}{Nk} + \frac{1}{k * k}$$

(上述資料及圖來源: https://chih-sheng-huang821.medium.com/%E6%B7%B1%E5%BA%A6%E5%AD%B8%E7%BF%92-mobilenet-depthwise-separable-convolution-f1ed016b3467)

它的作用是能大幅降低 CNN 的計算量，Kernel Map 越大、數量越多，降低的效果就越明顯。在 Pytorch 中，用 groups 參數來實現。

DeepConvNet

Layer	# filters	size	# params	Activation	Options
Input		(C, T)			
Reshape		(1, C, T)			
Conv2D	25	(1, 5)	150	Linear	mode = valid, max norm = 2
Conv2D	25	(C, 1)	25 * 25 * C + 25	Linear	mode = valid, max norm = 2
BatchNorm			2 * 25		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Conv2D	50	(1, 5)	25 * 50 * C + 50	Linear	mode = valid, max norm = 2
BatchNorm			2 * 50		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Conv2D	100	(1, 5)	50 * 100 * C + 100	Linear	mode = valid, max norm = 2
BatchNorm			2 * 100		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Conv2D	200	(1, 5)	100 * 200 * C + 200	Linear	mode = valid, max norm = 2
BatchNorm			2 * 200		epsilon = 1e-05, momentum = 0.1
Activation				ELU	
MaxPool2D		(1, 2)			
Dropout					p = 0.5
Flatten					
Dense	N			softmax	max norm = 0.5

依上圖進行實作。其中 C = 2。

```python
class DeepConvNet(nn.Module):
    def __init__(self, act_func, device):
        super(DeepConvNet, self).__init__()
        self.device = device
        self.conv0 = nn.Conv2d(1, 25, kernel_size = (1, 5))
        self.conv1 = nn.Sequential(
            nn.Conv2d(25, 25, kernel_size = (2, 1)),
            nn.BatchNorm2d(25, eps = 1e-5, momentum = 0.1),
            act_func,
            nn.MaxPool2d(kernel_size = (1, 2)),
            nn.Dropout(p = 0.5)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(25, 50, kernel_size = (1, 5)),
            nn.BatchNorm2d(50, eps = 1e-5, momentum = 0.1),
            act_func,
            nn.MaxPool2d(kernel_size = (1, 2)),
            nn.Dropout(p = 0.5)
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(50, 100, kernel_size = (1, 5)),
            nn.BatchNorm2d(100, eps = 1e-5, momentum = 0.1),
            act_func,
            nn.MaxPool2d(kernel_size = (1, 2)),
            nn.Dropout(p = 0.5)
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(100, 200, kernel_size = (1, 5)),
            nn.BatchNorm2d(200, eps = 1e-5, momentum = 0.1),
            act_func,
            nn.MaxPool2d(kernel_size = (1, 2)),
            nn.Dropout(p = 0.5)
        )

        self.classify = nn.Linear(8600, 2)
```

```python
def forward(self, X):
    out = self.conv0(X.to(self.device))
    out = self.conv1(out)
    out = self.conv2(out)
    out = self.conv3(out)
    out = self.conv4(out)
    out = out.view(out.shape[0], -1)  #flatten
    out = self.classify(out)
    #out = nn.functional.softmax(out, dim = 0)
    return out
```

而這個就是普通的 CNN 架構。其中要注意的是，我們在 main 中用了 nn.CrossEntropyLoss，這之中就包含了 softmax 運算，因此這裡不

需要再呼叫 softmax。

main.py

main 檔為主要執行訓練的檔案。首先是 Hyper Parameters:

```
#model = EEGNet(a, device)
model = DeepConvNet(a, device)
model.to(device)
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001, weight_decay = 0.01)
loss_func = nn.CrossEntropyLoss()
num_epochs = 500
```

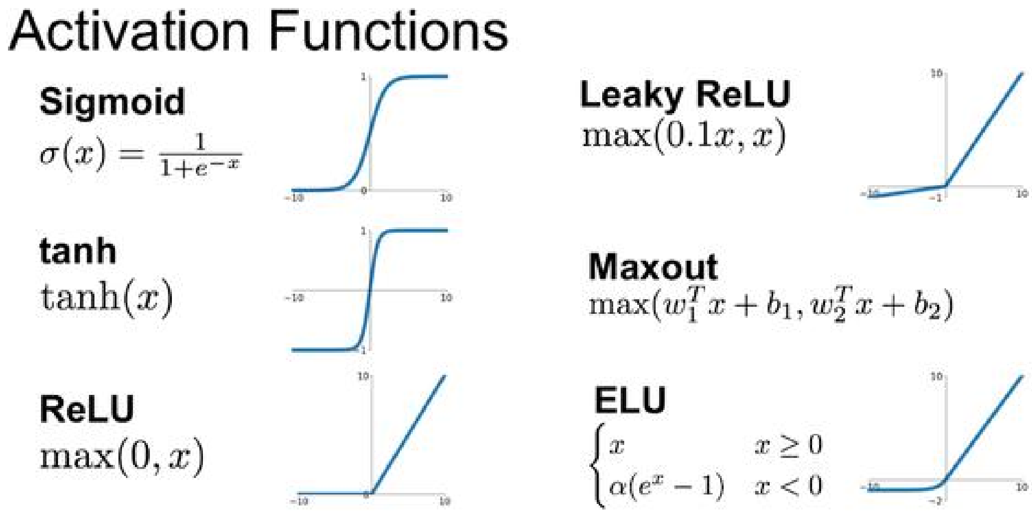
先選擇使用 EEGNet 或是 DeepConvNet，接著使用 Adam 優化器進行 gradient descent update。learning rate 設為 0.001，weight decay 是指在更新參數時加上一個懲罰，避免更新太多導致 overfitting，這裡設為 0.01。loss function 用 Cross Entropy，epochs 設為 500 個。

接著是 training 的部分:

```
total_train_accuracy = []
total_test_accuracy = []
for epoch in range(num_epochs):
    # training
    correct_train = 0
    total_train = 0
    total_loss = 0
    for i, (data, labels) in enumerate(train_loader):
        data = data.to(device, dtype = torch.float)
        labels = labels.to(device, dtype = torch.long)

        # Clear gradients
        optimizer.zero_grad()

        # Forward
        outputs = model(data)

        # Calculate cross entropy loss
        train_loss = loss_func(outputs, labels)
        total_loss += train_loss

        # Get predictions from the maximum value
        predicted = torch.max(outputs.data, 1)[1]

        # Total number of labels
        total_train += len(labels)

        # Total correct predictions
        correct_train += (predicted == labels).float().sum()

        # Calculate gradients
        train_loss.backward()

        # Update parameters
        optimizer.step()

    train_accuracy = 100 * (correct_train / total_train)
```

步驟都寫在圖中的註解上了。大致上就是重複清空 optimizer 梯度、Forwarding、計算 loss、呼叫 backward 計算 gradients、呼叫 step 進行 gradient descent update 等步驟。而 testing 也是一樣，只是不用計算 gradient 及 update 而已。

B. Explain the activation function (ReLU, Leaky ReLU, ELU)



上圖是各個 activation function 的圖形。ELU 和 Leaky ReLU 相較於 ReLU 的差異在於，當 value 小於 0 時前者依舊有梯度存在，而後者的梯度會直接為 0。導致在 backpropagate 時，ELU 和 Leaky ReLU 會比較容易進行更新。

Experimental results

A. The highest testing accuracy

learning rate = 0.001

weight decay = 0.01

epochs = 500

EEGNet:

```
epoch 490:
trainig accuracy: tensor(97.7778, device='cuda:0')  loss: tensor(0.4665, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(84.3519, device='cuda:0')

epoch 500:
trainig accuracy: tensor(97.0370, device='cuda:0')  loss: tensor(0.5123, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(84.6296, device='cuda:0')

LeakyReLU has max accuracy 87.22222137451172% at epoch 491

ELU has max accuracy 82.96295928955078%
ReLU has max accuracy 87.31481170654297%
LeakyReLU has max accuracy 87.22222137451172%
```

DeepConvNet:

```
epoch 490:
trainig accuracy: tensor(94.2593, device='cuda:0')  loss: tensor(0.7593, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(77.9630, device='cuda:0')

epoch 500:
trainig accuracy: tensor(95.4630, device='cuda:0')  loss: tensor(0.6326, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(80.7407, device='cuda:0')

LeakyReLU has max accuracy 82.59259033203125% at epoch 239

ELU has max accuracy 80.74073791503906%
ReLU has max accuracy 82.96295928955078%
LeakyReLU has max accuracy 82.59259033203125%
```

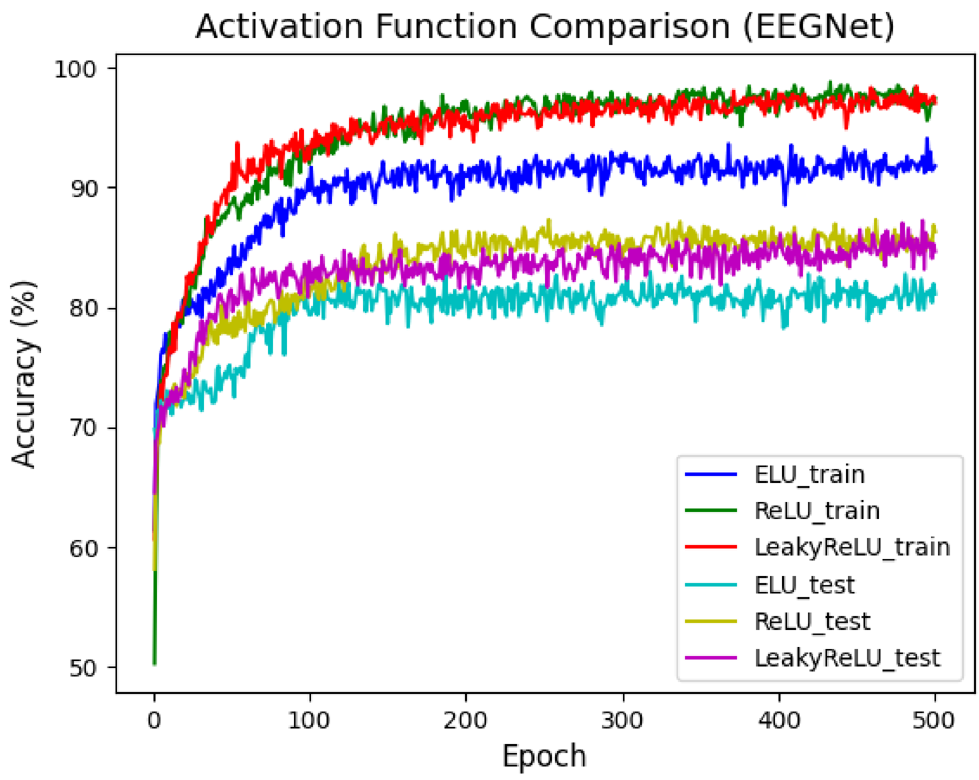
Testing accuracy comparison:

	ELU	ReLU	LeakyReLU
EEGNet	82.96%	87.31%	87.22%
DeepConvNet	80.74%	82.96%	82.59%

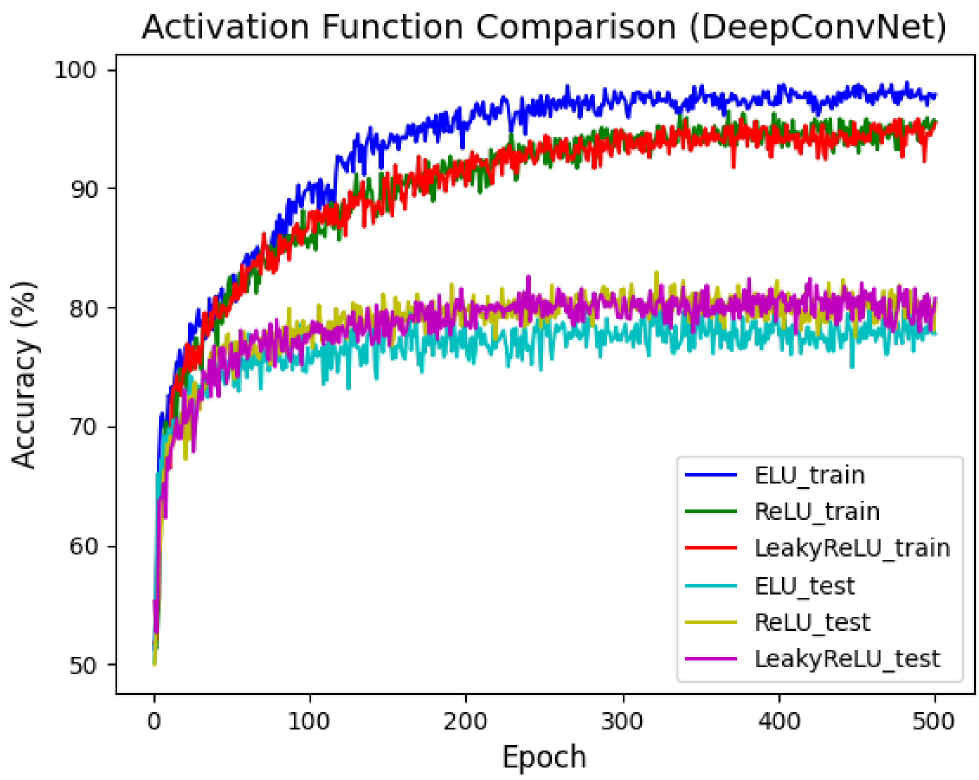
使用 EEGNet 和 ReLU 及 LeakyReLU 都讓 testing accuracy 超過 87%。

B. Comparison figures

EEGNet:



DeepConvNet:



Discussion

weight decay 的使用:

前面有提到，weight decay 是為了避免參數一次更新太多，造成對 training data overfitting。也可以說是對 update 的量做一個「懲罰」。若沒有使用 weight decay 這個參數，testing 的準確率則會發生下降，如下:

EEGNet without weight decay:

```
epoch 490:
trainig accuracy: tensor(98.3333, device='cuda:0')  loss: tensor(0.2108, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(83.5185, device='cuda:0')

epoch 500:
trainig accuracy: tensor(98.2407, device='cuda:0')  loss: tensor(0.2288, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(83.1481, device='cuda:0')

LeakyReLU has max accuracy 85.09259033203125% at epoch 257

ELU has max accuracy 83.6111068725586%
ReLU has max accuracy 86.5740737915039%
LeakyReLU has max accuracy 85.09259033203125%
```

DeepConvNet without weight decay:

```
epoch 490:
trainig accuracy: tensor(96.2963, device='cuda:0')  loss: tensor(0.5426, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(77.7778, device='cuda:0')

epoch 500:
trainig accuracy: tensor(96.0185, device='cuda:0')  loss: tensor(0.4785, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(77.2222, device='cuda:0')

LeakyReLU has max accuracy 79.62963104248047% at epoch 453

ELU has max accuracy 80.18518066040625%
ReLU has max accuracy 81.48148345947266%
LeakyReLU has max accuracy 79.62963104248047%
```

可以看到，和原本大約差 1%至 2%，雖然差距不大，卻足以讓其通過要求的 87%標準。

CPU vs GPU:

我們可以用下面這段 code 測量執行時間。在 CPU 或 GPU 都能適用。

```
start = torch.cuda.Event(enable_timing=True)
end = torch.cuda.Event(enable_timing=True)

start.record()
# operations you want to measure
end.record()

# Waits for everything to finish running
torch.cuda.synchronize()

print('exe time: ' + str(start.elapsed_time(end) / 1000) + ' s')
```

測量環境:

model: EEGNet
activation function: LeakyReLU
using CPU:

```
epoch 490:
trainig accuracy: tensor(96.4815)  loss: tensor(0.5066, grad_fn=<AddBackward0>)
testing accuracy: tensor(84.8148)

epoch 500:
trainig accuracy: tensor(98.0556)  loss: tensor(0.4361, grad_fn=<AddBackward0>)
testing accuracy: tensor(85.6481)

LeakyReLU has max accuracy 87.31481170654297% at epoch 316
exe time: 357.27465625 s
```

using GPU:

```
epoch 490:
trainig accuracy: tensor(96.9444, device='cuda:0')  loss: tensor(0.5316, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(85.8333, device='cuda:0')

epoch 500:
trainig accuracy: tensor(96.9444, device='cuda:0')  loss: tensor(0.4499, device='cuda:0', grad_fn=<AddBackward0>)
testing accuracy: tensor(87.0370, device='cuda:0')

LeakyReLU has max accuracy 87.87036895751953% at epoch 420
exe time: 38.46273046875 s
```

從上兩張圖的比較可以看出，若是用 GPU，則 tensor 型態中會多顯示一個 device = cuda:0，用 CPU 則不會額外顯示。且 GPU 執行約 38 秒，比 CPU 的 357 秒快了 9 倍多。這可以證明 GPU 的平行運算在 CNN 的加速效果是非常明顯的。