我做過的嘗試與遇到的問題:

1. 起初我想嘗試使用transfer learning的方式把所有有名的model implement進來看看每個model的準確率，所以我試了VGG16, Resnet50, Resnet152, mobilnet_v2, efficientnet_b0等等。

VGG16

```
Accuracy of the network on the 10000 test images: 64.88 %
```

Resnet152

```
Accuracy of the network on the 10000 test images: 55.42 %
```

model size 224M

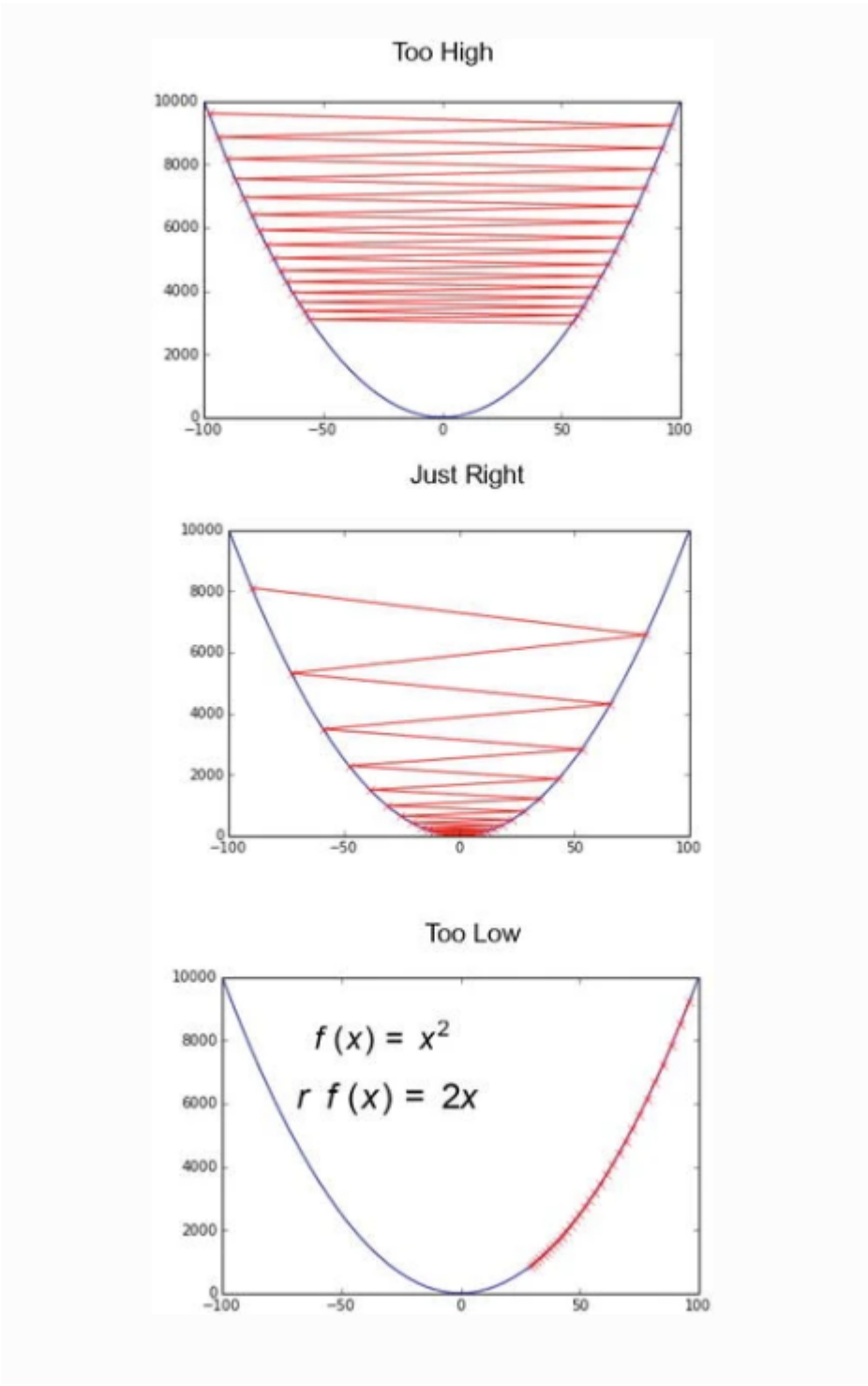
mobilenet_v2

```
Accuracy of the network on the 10000 test images: 55.66 %
```

resnet50

```
Accuracy of the network on the 10000 test images: 62.73 %
```

問題: 在眾多嘗試中，我發現每個model訓練下來的情形都不太理想，唯二最好的兩個就是vgg16跟resnet50，然而我的電腦在訓練vgg16時都會有cuda out of memory的問題，原本使用colab但後來colab說要我購買pro的版本才能繼續使用，所以就沒有再嘗試vgg16的訓練，而resnet50我嘗試了不同的learning rate跟batch size，也參考了很多小批次訓練的方法，包括使用好的optimizer像是Adam、SGD，以及降低learning rate的方式去做訓練但結果也是不超過65%。



(fig 8-5 from Stochastic Gradient Descent Nikhil Ketkar)

2. 另外我嘗試了各種不同的data augumentation，像是Randomcrop、Randomrotation等等，結果訓練出來的結果也很慘，loss反而更加降不下去了，我才理解到，訓練model不是隨便亂加這些就可以讓他準確率提高，有時候反而會成為不好的noise導致model無法收斂。

3. 最後參考了著名的efficientnet的paper，在降低model的大小的情況下準確率還可以很高，但我自己使用models.efficientnet_b0()的方式來取得model訓練後loss一直停在3左右降不下去，原因我也不清楚。因此到這邊我決定要來自己查詢網路有沒有人自己架設一個model來訓練同時訓練成效也不錯的。

What I have done

我參考這篇文章來架設model
https://www.kaggle.com/code/yiweiwangau/cifar-100-resnet-pytorch-75-17-accuracy?scriptVersionId=79050478
我所做的調整包括

1. 在data augumentation加入
transforms.RandomCrop(32,padding=4,padding_mode='reflect')

```
mean = (0.5071, 0.4867, 0.4408)
std = (0.2675, 0.2565, 0.2761)
train_transform = transforms.Compose(
    [transforms.RandomCrop(32, padding=4,padding_mode='reflect'),
     transforms.RandomHorizontalFlip(p=0.5),
     transforms.ToTensor(),
     transforms.Normalize(mean, std)]) # calculte yourself

test_transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize(mean, std)]) # calculte yourself

batch_size = 256
num_classes = 100    # check
```

加入這個除了是參考了上面kaggle的作法外，也視察了很多不同人的做法發現很多人都會加入這個noise，讓model可以辨識更多實際可能發生的情形。減少overfitting的可能性。

2. 另外在

```
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                                          shuffle=True, num_workers=2, pin_memory=True)
```

加入了 pin_memory = True 目的是讓系統可以跑得快一點

3. 接著是架設自己的resnet

```python
class BaseModel(nn.Module):
    def training_step(self,batch):
        images,labels = batch
        out = self(images)
        loss = F.cross_entropy(out,labels)
        return loss

    def validation_step(self,batch):
        images,labels = batch
        out = self(images)
        loss = F.cross_entropy(out,labels)
        acc = accuracy(out,labels)
        return {"val_loss":loss.detach(),"val_acc":acc}

    def validation_epoch_end(self,outputs):
        batch_losses = [loss["val_loss"] for loss in outputs]
        loss = torch.stack(batch_losses).mean()
        batch_accuracy = [accuracy["val_acc"] for accuracy in outputs]
        acc = torch.stack(batch_accuracy).mean()
        return {"val_loss":loss.item(),"val_acc":acc.item()}

    def epoch_end(self, epoch, result):
        print("Epoch [{}], last_lr: {:.5f}, train_loss: {:.4f}, val_loss: {:.4f}, val_acc: {:.4f}".format(
            epoch, result['lrs'][-1], result['train_loss'], result['val_loss'], result['val_acc']))
```

```python
def conv_block(in_channels, out_channels, pool=False):
    layers = [nn.Conv2d(in_channels, out_channels, kernel_size=3, padding=1),
              nn.BatchNorm2d(out_channels),
              nn.ReLU(inplace=True)]
    if pool: layers.append(nn.MaxPool2d(2))
    return nn.Sequential(*layers)

class ResNet9(BaseModel):
    def __init__(self, in_channels, num_classes):
        super().__init__()

        self.conv1 = conv_block(in_channels, 32)
        self.conv2 = conv_block(32, 64, pool=True)
        self.res1 = nn.Sequential(conv_block(64, 64), conv_block(64, 64))

        self.conv3 = conv_block(64, 128, pool=True)
        self.conv4 = conv_block(128, 256, pool=True)
        self.res2 = nn.Sequential(conv_block(256, 256), conv_block(256, 256))
        self.conv5 = conv_block(256, 512, pool=True)
        self.res3 = nn.Sequential(conv_block(512, 512), conv_block(512, 512))

        self.classifier = nn.Sequential(nn.MaxPool2d(2), # 1028 x 1 x 1
                                        nn.Flatten(), # 1028
                                        nn.Linear(512, num_classes)) # 1028 -> 100

    def forward(self, xb):
        out = self.conv1(xb)
        out = self.conv2(out)
        out = self.res1(out) + out
        out = self.conv3(out)
        out = self.conv4(out)
        out = self.res2(out) + out
        out = self.conv5(out)
        out = self.res3(out) + out
        out = self.classifier(out)
        return out
```

那這邊基本上是參照kaggle上面的做法去做一模一樣的model，但不同的是，我遇到了一個問題這個model本身的大小超過了100MB，我常是增加幾層layer跟減少layer，而我發現最有效率減少model的大小的方式，是把裡面的parameters減少，因此我把所有的parameters大約減半做訓練，做出來的成效跟原本的準確率大概差了4%左右。而最後的size也從117M減少到29M

```
# see size of saved model
! du -h model.pth
```

```
29M     model.pth
```

另外kernal size跟padding的部分，我參照了網路上一些有經驗的人的建議，大家幾乎都是一致的將kernal size調成3，padding調成1。

4. 這是我model的架構

```
ResNet9(
  (conv1): Sequential(
    (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
  )
  (conv2): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res1): Sequential(
    (0): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (conv3): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv4): Sequential(
    (0): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res2): Sequential(
    (0): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
```

```
  (conv5): Sequential(
    (0): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (res3): Sequential(
    (0): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
    (1): Sequential(
      (0): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (2): ReLU(inplace=True)
    )
  )
  (classifier): Sequential(
    (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (1): Flatten(start_dim=1, end_dim=-1)
    (2): Linear(in_features=512, out_features=100, bias=True)
  )
)
```

5. 這是我訓練的參數

```
epochs = 30
optimizer =  optim.Adam
max_lr=0.001
grad_clip = 0.01
weight_decay = 1e-3
scheduler = optim.lr_scheduler.OneCycleLR
```

比較特別的是grad_clip跟scheduler的使用，grad_clip我查詢網路上的說明，是可以防止梯度爆炸的東西，可以將梯度約束在某一個區間。
而scheduler我認為可能是造成這次訓練結果還不錯的關鍵原因(但我不太確定)，他可以讓learning rate從小到大再從大到小，並且我發現在這個訓練過程中當learning rate由小到大跟大到小的過程中往往是loss下降最快的時候。

```
Epoch [0], last_lr: 0.00007, train_loss: 3.8653, val_loss: 3.3510, val_acc: 0.2034
Epoch [1], last_lr: 0.00015, train_loss: 3.1356, val_loss: 2.8628, val_acc: 0.2897
Epoch [2], last_lr: 0.00028, train_loss: 2.7541, val_loss: 2.7631, val_acc: 0.3056
Epoch [3], last_lr: 0.00044, train_loss: 2.4952, val_loss: 2.5043, val_acc: 0.3513
Epoch [4], last_lr: 0.00060, train_loss: 2.2797, val_loss: 2.3438, val_acc: 0.3842
Epoch [5], last_lr: 0.00076, train_loss: 2.0712, val_loss: 2.3189, val_acc: 0.3983
Epoch [6], last_lr: 0.00089, train_loss: 1.9080, val_loss: 2.1942, val_acc: 0.4285
Epoch [7], last_lr: 0.00097, train_loss: 1.7708, val_loss: 2.0920, val_acc: 0.4437
Epoch [8], last_lr: 0.00100, train_loss: 1.6419, val_loss: 1.8849, val_acc: 0.5021
Epoch [9], last_lr: 0.00099, train_loss: 1.5183, val_loss: 2.2129, val_acc: 0.4464
Epoch [10], last_lr: 0.00098, train_loss: 1.4333, val_loss: 1.8605, val_acc: 0.5071
Epoch [11], last_lr: 0.00095, train_loss: 1.3424, val_loss: 1.6138, val_acc: 0.5564
Epoch [12], last_lr: 0.00091, train_loss: 1.2619, val_loss: 1.5546, val_acc: 0.5673
Epoch [13], last_lr: 0.00087, train_loss: 1.2032, val_loss: 1.5822, val_acc: 0.5696
Epoch [14], last_lr: 0.00081, train_loss: 1.1352, val_loss: 1.5486, val_acc: 0.5697
Epoch [15], last_lr: 0.00075, train_loss: 1.0717, val_loss: 1.4003, val_acc: 0.6059
Epoch [16], last_lr: 0.00068, train_loss: 1.0044, val_loss: 1.4915, val_acc: 0.5881
Epoch [17], last_lr: 0.00061, train_loss: 0.9346, val_loss: 1.3578, val_acc: 0.6155
Epoch [18], last_lr: 0.00054, train_loss: 0.8658, val_loss: 1.3333, val_acc: 0.6302
Epoch [19], last_lr: 0.00046, train_loss: 0.7938, val_loss: 1.3371, val_acc: 0.6325
Epoch [20], last_lr: 0.00039, train_loss: 0.7077, val_loss: 1.2301, val_acc: 0.6633
Epoch [21], last_lr: 0.00032, train_loss: 0.6254, val_loss: 1.2047, val_acc: 0.6652
Epoch [22], last_lr: 0.00025, train_loss: 0.5376, val_loss: 1.1349, val_acc: 0.6849
Epoch [23], last_lr: 0.00019, train_loss: 0.4481, val_loss: 1.0925, val_acc: 0.6979
Epoch [24], last_lr: 0.00013, train_loss: 0.3644, val_loss: 1.0806, val_acc: 0.7064
Epoch [25], last_lr: 0.00009, train_loss: 0.2929, val_loss: 1.0695, val_acc: 0.7114
Epoch [26], last_lr: 0.00005, train_loss: 0.2433, val_loss: 1.0622, val_acc: 0.7135
Epoch [27], last_lr: 0.00002, train_loss: 0.2066, val_loss: 1.0561, val_acc: 0.7195
Epoch [28], last_lr: 0.00001, train_loss: 0.1845, val_loss: 1.0524, val_acc: 0.7200
Epoch [29], last_lr: 0.00000, train_loss: 0.1743, val_loss: 1.0523, val_acc: 0.7204
```

因此我認為是這個scheduler的方式讓這個model在訓練過程優化了不少。

6. 最後是這個model的testing結果

In [3]:
```python
# load trained model
from model import ResNet9
model = torch.load("./model_Resnet9final.pth")
model.to(device)

# fixed testing process
correct = 0
total = 0
# since we're not training, we don't need to calculate the gradients for our outputs
with torch.no_grad():
    for data in testloader:
        images, labels = data
        images = images.to(device)
        labels = labels.to(device)
        # calculate outputs by running images through the network
        outputs = model(images)
        # the class with the highest energy is what we choose as prediction
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
print(f'Accuracy of the network on the 10000 test images: {100 * correct / total:.2f} %')
```

```
Accuracy of the network on the 10000 test images: 71.82 %
```

調整完大小的model跟kaggle上的範例大約差了3-4%的accuracy。

What I have leaned

1. 這次的經驗我覺得十分難能可貴，因為以前做過ML的經驗常常是參考他人的code然後一模一樣的去使用他，最後達到一個結果，但這樣的經驗永遠無法讓我理解這些code中間的意義，以及當我要完全自己寫出來時該注意甚麼重要的東西。因此這次的經驗，讓我從處理data一直到架設model上都有了很多概念，也發現有時候不是隨便亂調就可以讓model變得更好，現在已經有很多有經驗的人撰寫的paper寫了很多能讓model更有效率的調整方式，應該多加查詢並且嘗試，才可以站在他人的經驗上進步。

2. 這次很慶幸我最後選擇了架設一個model出來，而不是單純使用transfer learning的方式使用別人的model，因為這幫助了我更加理解架設model時每一層layer的重要性，以及層層input與output的環環相扣，另外我也理解了其實架設一層層的layer沒有想像中那麼難。雖然現在還是要參考別人的code才能做更改寫出來。

3. 最後特別的是自己架設的model我不僅可以將batch size設到256都不會有cuda out of memory的問題，而且準確率還比其他model高，原因我也不知道為甚麼，可能是架構簡單不會太複雜，反而可以達到不錯的訓練成效。

Reference

https://blog.jovian.ai/image-classification-of-cifar100-dataset-using-pytorch-8b7145242df1
(**Image Classification of CIFAR100 Dataset Using PyTorch**)
https://link.springer.com/chapter/10.1007/978-1-4842-2766-4_8
(Stochastic Gradient Descent Nikhil Ketkar)
https://www.kaggle.com/code/yiweiwangau/cifar-100-resnet-pytorch-75-17-accuracy?scriptVersionId=79050478
(**CIFAR-100 Resnet PyTorch 75.17% Accuracy**)
https://medium.com/%E8%BB%9F%E9%AB%94%E4%B9%8B%E5%BF%83/efficientnet-v2%E7%9A%84%E8%83%8C%E5%BE%8C-%E9%87%8B%E6%94%BEmobilenet%E5%9C%A8gpu-tpu%E4%B8%8A%E7%9A%84%E6%95%88%E7%8E%87-f19abde55b05
(**EfficientNet V2**的背後**:** 釋放**MobileNet**在**GPU/TPU**上的效率)
https://www.twblogs.net/a/5cfff8b9bd9eee14029fb25f
(**pytorch** 學習 **|**梯度截斷 **gradient clip** 的簡單實現)
https://zhuanlan.zhihu.com/p/387162205
(**OneCycleLR**学习率的原理与使用)

EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks

Mingxing Tan, Quoc V. Le

International Conference on Machine Learning, 2019